Yang, J. C. et al. (Eds.) (2018). Proceedings of the 26th International Conference on Computers in Education. Philippines: Asia-Pacific Society for Computers in Education

Supporting Novices and Advanced Students in Acquiring Multiple Coding Skills

Geela Venise Firmalo FABIC[a*], Antonija MITROVIC[a] & Kourosh NESHATIAN[a]
[a] *Computer Science and Software Engineering, University of Canterbury, New Zealand*
*geela.fabic@pg.canterbury.ac.nz

Abstract: We present our study on PyKinetic with various activities to target several skills: code tracing, debugging, and code writing. Half of the participants (control group) received the problems in a fixed order, while for the other half (experimental group) problems were selected adaptively, based on their performance. In a previous paper, we discussed the general findings from the study. In this paper we present further analyses and focus on differences between low performing students and students with higher pre-existing knowledge. We hypothesized that: (H1) novices will benefit more than advanced students, and (H2) advanced students in the experimental group will benefit more than those in the control group. The results confirmed H1 and revealed that this version of PyKinetic was more beneficial for novice learners. Moreover, novices showed evidence of learning multiple skills: code writing, debugging and code tracing. However, we did not have enough evidence for hypothesis H2.

Keywords: mobile Python tutor, Parsons problems, adaptive problem selection, code debugging, code tracing, code writing

1. Introduction

In 2015, we started developing PyKinetic, a mobile Python tutor for novices designed for Android smartphones where all activities are designed to be completed without typing, and only require tap and long-tap interactions (Fabic, Mitrovic & Neshatian, 2016). Our motivation was to develop a mobile tutor as a supplement to lectures and labs. The current version of PyKinetic targets multiple coding skills via five types of activities: regular Parsons problems, Parsons problems with incomplete lines of code (LOCs), identifying erroneous LOCs, fixing erroneous LOCs, and output prediction. We conducted a study comparing two versions of PyKinetic: a version with the fixed sequence of problems, and a version which selected problems for students adaptively, based on their performance. In a previous paper (Fabic, Mitrovic & Neshatian, 2018) we presented the overall results of that study, focusing on the differences between experimental and control groups and showed that the adaptive version was more beneficial for learning in comparison to the version with a fixed order of problems. In this paper, we delve deeper in the same study and present further analyses for novices and advanced students. Our hypothesis is that novices will benefit more than advanced students in using this version of PyKinetic (H1). We also hypothesize that advanced students in the experimental group will benefit more than advanced students in the control group (H2). Based on results from our previous study, advanced students benefit more with debugging activities (Fabic, Mitrovic & Neshatian 2017a). Therefore, the adaptive version may provide more suitable problems for advanced students and result in better-quality learning.

A variety of skills necessary for programming have been discussed in the literature. Researchers found that code tracing must be learned before code writing (Lopez et al., 2008; Thompson et al., 2008; Harrington & Cheng, 2018). Further evidence proves existence of relationships between code tracing, code writing and code explaining (Lister et al., 2009; Venables, Tan & Lister, 2009). A strong positive correlation was found between code tracing and code writing (Lister et al., 2010). Harrington & Cheng (2018) conducted a study and found that regardless of whether the student is better in either code tracing or code writing, a large gap between the two skills is most likely due to students struggling with understanding core programming concepts. Ahmadzadeh, Elliman & Higgins (2005) conducted a study on the debugging patterns of novices

and found that most learners competent in debugging were advanced programmers (66%). However, only 39% of advanced programmers were also competent in debugging. These findings provide indications that debugging and tracing someone else's program requires a higher order of skill than code writing.

Parsons problems are exercises that aid learners in recognizing the sequential and non-sequential aspects of programming. These problems were originally developed as a fun way to learn Turbo Pascal (Parsons & Haden, 2006) to improve syntactic skills. These problems are suitable for novices as they contain syntactically correct code that only needs to be arranged in the right order. There are variations of Parsons problems implemented such as Parsons problems with distractors (extra LOCs) (Fabic, Mitrovic & Neshatian 2016; Kumar, 2018), and Parsons problems with incomplete LOCs (Fabic, Mitrovic & Neshatian 2017a; Ihantola, Helminen & Karavirta, 2013). More variants are considered by Denny et al. (2008) and (Cheng & Harrington, 2017). To the best of our knowledge, most work in Parsons problems contains fragments of code containing multiple lines for each fragment. However, in our work, all versions of PyKinetic containing Parsons problems all had code fragments containing exactly one line per fragment, which requires more effort (moves) for a problem to be solved. Similarly, Kumar (2018) also implemented Parsons problems with fragments containing single LOCs.

Self-explanation (SE) is a learning activity which promotes deeper learning, by producing inference rules and justifications which are not directly presented by the material (Chi et al., 1989). SE prompts were first introduced as open-ended questions which encourage learners to think without any set limitations. Over the years, SE activities have been proven useful in various domains such as geometry (Aleven, Koedinger & Cross, 1999), probability (Atkinson, Renkl & Merrill, 2003; Berthold, Eysink & Renkl, 2009), data normalization (Mitrovic, 2005), database modelling (Weerasinghe & Mitrovic, 2006), electrical circuits (Johnson & Mayer, 2010), and chemistry (McLaren et al., 2016). There were several studies comparing different forms of self-explanation (Wylie & Chi, 2014). In PyKinetic, we have used *menu-based SE prompts*, which provide choices from a menu, instead of traditional open-ended questions proven to be effective in our previous study (Fabic, Mitrovic & Neshatian, 2017a; 2017b).

2. Types of Problems in PyKinetic

A PyKinetic problem contains a description, a code snippet, and one or more activities. Activities can be Parsons problems, completing missing elements in LOCs, output prediction, identifying incorrect LOCs and fixing them. We defined seven types of problems (see Table 1, second column). Problem types 1–4 consist of a single activity each, whereas other types (5–7) are combinations of two or more activities. The problem types are ordered by the complexity and the number of coding skills involved. The code provided for problems with debugging activities contain 1–3 incorrect LOCs, whereas other problem types contain error-free code. PyKinetic covers six Python topics: string manipulation, conditional statements, while loops, for loops, lists, and tuples.

The simplest problem type (**Level 1**) is a regular Parsons problem (*Reg_Pars*) which only requires LOCs to be rearranged by dragging and dropping. Correct indentations are provided for all LOCs as scaffolding. For each Reg_Pars, there is one predefined hint. Subsequent incorrect attempts result in alternating simple feedback and a predefined hint. When the learner successfully reorders the LOCs in the correct order, PyKinetic provides positive feedback.

On **Level 2** (*Out*), the student is given one or more test cases, and needs to predict the output of the given code. For each Out activity, there are three incorrect and one correct choice. One predefined hint is provided if the learner selects an incorrect choice. After an incorrect attempt, a choice cannot be made without first closing the output prediction dialog box to encourage learners to review the code.

Level 3 problems consist of a single debugging activity (*Dbg*), requiring the student to identify *n* erroneous LOCs, where *n* is given. In Dbg activities, the learner is given the problem description and some test cases with the actual output the code produces. If the solution is correct, the student receives positive feedback. When the solution is incorrect, feedback is firstly given if the student selects too many or too few incorrect LOCs. If the learner selects exactly *n* lines, but the selections are all incorrect, this would result in alternating simple feedback and a predefined hint

like given in Reg_Pars (Level 1). Also, like Reg_Pars, there is only one predefined hint given for each Dbg activity. Moreover, the learner is notified when their solution was partially correct.

Level 4 contains Parsons problems with incomplete LOCs and SE prompts (*Pars_Inc*). Initially, the student is given the description of the problem and the expected output. Each Pars_Inc problem contains up to three incomplete lines. An incomplete LOC contains a blank line, which may require one or more keywords. The length of the blank line is indicative of keywords needed. To complete a LOC, an answer is chosen by tapping between provided options instead of typing like work of Ihantola, Helminen, & Karavirta, (2013). When the learner selects the correct option, the learner next gets the SE prompt, which is associated to the line just completed. The learner is only allowed to attempt the SE question once to avoid guessing and is not allowed to avoid it.

Level 5 problem is a combination of debugging and output prediction (*Dbg –> Out*). *Dbg –> Fix* (**Level 6**) is a combination of debugging and fixing activities. Lastly, the most complex problem type is *Dbg –> Out –> Fix* (**Level 7**), which is a combination of three activities: identifying erroneous LOCs, predicting the output for the same erroneous code, and fixing the identified errors.

3. Experimental Design and Adaptive Problem Selection

We recruited 30 participants from an introductory programming course at the University of Canterbury. The participants learnt all Python topics covered in PyKinetic before the study. The study was approved by the Human Ethics committee of the University of Canterbury. The participants were randomly assigned into control or experimental group. The session length was two hours. The participants were first given a brief introduction and their consent was obtained, followed by a pre-test on computers (18 minutes). Next, a briefing paper was given for PyKinetic, together with an Android phone with the app installed. For the experimental group, we entered the pre-test scores into PyKinetic, so that the adaptive strategy could select the first problem based on each learners' pre-test score. Both groups had 14 problems to solve. A learner must complete a problem to proceed to the next one. Participants interacted with PyKinetic for an hour, then a post-test was given, with the same constraints as the pre-test. There were two tests of similar complexity that were alternatively used as the pre-test for half of the participants. Both tests had six questions of the same types as in PyKinetic. However, instead of having two variants Parsons problems, there was only one Parsons problem with three extra LOCs (distractors) completed by drag and drop. Moreover, a code writing question was included which required the learners to type their code without being able to run it. Other questions were answered by multiple choice and drop-down list boxes. Each question was worth 1 mark for each problem it required, apart from the Parsons problem with distractors (2 marks), and the code writing question (5 marks).

The control group received problems in the fixed order, as shown in Table 1. The problems given to the experimental group were selected adaptively based on each student's performance. In steps 1–7, the participants could receive a regular Parsons problem (Reg_Pars), output prediction problem (Out) or be asked to identify erroneous LOCs (Dbg). Steps 8–14 were composed of more difficult problems (levels 4–7). The 14 problems that were given to the control group correspond to 42 problems for the experimental group, to provide three difficulty levels. For example, problem 1 was a Reg_Pars for the control group; for the experimental group, the same problem was given either as a Reg_Pars, Out or Dbg. For steps 2–14, the adaptive strategy selected the problem type based on the student's score on the previous problem.

For the experimental group, the first problem was selected based on the participant's pre-test score. If the participant scored below 50%, a Reg_Pars was given. Participants who achieved at least 50% but less than 75% were given an Out (level 2) problem; however, a Dbg (level 3) problem was given if the learner performed well on Out or performed similarly compared to Dbg. For participants who scored more than 75%, a Dbg problem was given.

After the first problem, the adaptive strategy used the performance on the previous step to select the next problem. If the score is less than 50%, a problem of difficulty level 1 is selected. A difficulty level 2 problem is selected for scores more than 50% but less than 75%, and a difficulty level 3 if the score is at least 75%. PyKinetic calculates the score for each activity in a problem separately, and the problem score is the average of the activities scores. The score for an activity depends on the time taken (*TimeScore*) and the number of attempts (*AttemptScore*). The ideal

number of attempts for an activity is the minimum number of submissions needed to complete. *TimeScore* is the quotient of ideal time and the actual time the student took. *AttemptScore* is calculated as the quotient of the ideal and the actual number of submissions the student made. Both scores are then combined to compute the score for the activity: ActivityScore = (0.5 * *TimeScore* + 0.5 * *AttemptScore*) – *Penalty*. Furthermore, a penalty is applied if the time per attempt is less than 10 seconds (0.17 min – *AttemptTime*). The penalty is calculated based on the time taken per attempt (*AttemptTime*) and on 10 seconds threshold. The shorter the *AttemptTime*, the bigger the penalty.

Table 1

Problems for each step: Fixed (Control group) vs. Adaptive (Experimental group)

Step	Fixed (Control)	Adaptive (Experimental)
1	Regular Parsons problem (Reg_Pars)	Difficulty level 1: Reg_Pars Difficulty level 2: Out Difficulty level 3: Dbg
2		
3	Output prediction (Out)	
4		
5	Identifying erroneous Lines of Code (Dbg)	
6		
7	Parsons Problem with incomplete Lines of Code and Menu-based SE (Pars_Inc)	Difficulty level 1: Pars_Inc Difficulty level 2: Dbg –> Out Difficulty level 3: Dbg –> Fix
8		
9	Dbg –> Out	
10		
11	Dbg –> Fixing erroneous Lines of Code (Fix)	Difficulty level 1: Pars_Inc Difficulty level 2: Dbg –> Out Difficulty level 3: Dbg –> Out –> Fix
12		
13	Dbg –> Out –> Fix	
14		

4. Findings and Conclusions

We eliminated one outlier from the control group and present the results for the remaining 29 participants (15 in experimental and 14 in control). Due to the fixed session length, only 12 participants (41%) finished all 14 problems (6 from each group). On average, the participants completed 89% of the problems (12.52, s = 1.6). We divided the participants based on the pre-test scores: the participants who scored less than the median (72.92%) were labelled as novices, while the rest were considered as advanced participants. Due to random allocation of participants, we discovered that numbers of novices vs. advanced students were unbalanced in both groups. There were 15 novices (ten from experimental and five from control), and 14 advanced students (five from experimental and nine from control). Table 2 reports the results of pre/post-test scores of novices and advanced students.

Using the Wilcoxon Signed Ranks test, we found that the novices improved their scores significantly from the pre- to the post-test (W = 102, p = .017, Cohen's d = 1.14). Furthermore, their improvement on the code writing question was also significant (W = 75, p = .039, Cohen's d = .87). There were no significant improvements between the pre- to post-test scores for the advanced students. We also compared the scores of novices and advanced students. We expected to see significant differences for most of their pre- and post-test scores, due to the disparity between their abilities. The scores for identifying and fixing errors were significantly different only on the pre-test (U = 58.5, p = .023), but not on the post-test. Similarly, the output prediction scores on the pre-test were significantly different (U = 60, p = .029) but there was no difference on the post-test, showing benefits on the learning of novices. Even though novices improved significantly on their code writing scores, there was still a significant difference between their post-test scores when compared with advanced students (U = 59.5, p = .026). We used the Wilcoxon Signed Ranks test to identify whether there were significant improvements for the subgroups. Only novices in the control group

improved significantly between pre-/post-test (W = 15, p = .043, Cohen's d = 1.38). The imbalance of the students between control and experimental group likely affected our results.

Table 2

Some Pre-/Post-test Results for Novices and Advanced Participants

Scores (%)	Novices (15) 10 Exp. and 5 Cont.	Advanced (14) 5 Exp. and 9 Cont.	Mann-Whitney U test
Pre-test	57.64 (12.9)	88.99 (8.5)	U = 0, p = .000
Post-test	74.17 (15.9)	89.88 (8.9)	U = 48, p = .007
Pre-test Dbg, Fix and Dbg –> Fix	60 (28.7)	83.33 (25.3)	U = 58.5, p =.023
Post-test Dbg, Fix and Dbg –> Fix	66.67 (28.2)	85.72 (21.5)	ns
Pre-test Out	60 (20.7)	82.14 (24.9)	U = 60, p= .029
Post-test Out	73.33 (32)	75 (32.5)	ns
Pre-test Code Writing	42.67 (32.6)	91.43 (16.6)	U = 16, p = .000
Post-test Code Writing	70.67 (31.7)	95.71 (6.5)	U = 59.5, p = .026
Normalized Gain	33.66 (46.74)	14.29 (74.3)	ns

We also compared performances of novices and advanced students within PyKinetic. There were no significant differences on the average time, number of attempts per problem and average problem level between novices and advanced students. However, average scores in PyKinetic of advanced students were significantly higher than scores of novices (U = 41, p = .004). Novices in the experimental group received significantly more Level 3 (Dbg) problems than novices in the control group (U = 7.5, p = .028); they were also significantly faster on the initial seven problems (U = 46, p = .008). Furthermore, advanced students from the experimental group received significantly harder problems in the first half of the session than advanced learners from the control group (U = 5, p = .019). Advanced students in the experimental group received significantly fewer Level 1 (Reg_Pars) problems (easiest activity in PyKinetic) than advanced in the control group (U = 40.5, p = .012).

Our results show enough evidence to accept hypothesis H1, that PyKinetic is more beneficial for novices. Despite having only interacted with PyKinetic for an hour, novice students learned significantly from the pre- to post-test overall and on code writing. Furthermore, the Cohen's d effect size on both were notably high, 1.14 for the overall improvement, and 0.87 for code writing. However, the post-test scores of novices for code writing were still significantly lower than scores of advanced students. This is consistent with results from literature, showing that code writing skills require higher order of knowledge than other coding skills. We also found some evidence that the novice learners were learning multiple coding skills. The scores of novices on debugging (identifying errors and code fixing) and output prediction questions on the pre-test were significantly lower than the scores of advanced students. However, their post-test scores for those question types were not significantly different; indicating that the novices have reduced their gap on debugging and code tracing skills.

Hypothesis H2 was that advanced students in the experimental group would benefit more than those in the control group. Results revealed that advanced students in the experimental group received more difficult problems than those in the control group. However, that was not enough evidence to support hypothesis H2. There were only five advanced students in the experimental group but ten in the control group which most possibly affected our results. The main contribution of this paper is that a programming tutor in a smartphone can also be useful even for students with low prior knowledge to learn multiple coding skills by support of various activities. PyKinetic is designed to enhance coding skills in Python, and our findings support this.

The limitations of this study include the small set of participants, unbalanced abilities of participants in both groups, and limited feedback provided by PyKinetic. One direction for future improvement of PyKinetic is to add more hints and provide more constructive feedback. This study has the potential of revealing which activities are most effective for specific coding skills. Our future work includes repeating the study with more participants, and possibly conducting a longer study to investigate the benefits of PyKinetic over a longer period.

References

Ahmadzadeh, M., Elliman, D., & Higgins, C. (2005). An analysis of patterns of debugging among novice computer science students. *ACM SIGCSE bulletin*, 37(3), 84-88. ACM.

Aleven, V., Koedinger, K. R., & Cross, K. (1999) Tutoring answer explanation fosters learning with understanding. In: *Proc. 9th Int. Conf. Artificial Intelligence in Education*, (pp. 199-206).

Atkinson, R. K., Renkl, A., & Merrill, M. M. (2003). Transitioning from studying examples to solving problems: Effects of self-explanation prompts and fading worked-out steps. *Educational Psychology*, 95(4), 774.

Berthold, K., Eysink, T. H., & Renkl, A. (2009). Assisting self-explanation prompts are more effective than open prompts when learning with multiple representations. *Instructional Science*, 37(4), 345-363.

Cheng, N., & B., Harrington. (2017). The Code Mangler: Evaluating Coding Ability Without Writing any Code. In *Proc. ACM SIGCSE Technical Symposium on Computer Science Education*, (123-128). ACM.

Chi, M. T., Bassok, M., Lewis, M. W., Reimann, P., & Glaser, R. (1989). Self-explanations: How students study and use examples in learning to solve problems. *Cognitive science*, *13*(2), 145-182.

Denny, P., Luxton-Reilly, A., & Simon, B. (2008). Evaluating a new exam question: Parsons problems. In *Proc. 4th Int. workshop on computing education research* (pp. 113-124). ACM.

Fabic, G. V. F., Mitrovic A., & Neshatian, K. (2017a) A comparison of different types of learning activities in a mobile Python tutor. *Proc. 25th Int. Conf. Computers in Education*, (pp. 604-613).

Fabic, G. V. F., Mitrovic, A., & Neshatian, K. (2018) Adaptive Problem Selection in a Mobile Python Tutor. *Adjunct Proc. 26th User Modeling, Adaptation and Personalization conference,* (pp. 269-274). ACM.

Fabic, G., Mitrovic, A., & Neshatian, K. (2016) Investigating strategies used by novice and expert users to solve Parson's problem in a mobile Python tutor. *Proc. 9th Workshop on Technology Enhanced Learning by Posing/Solving Problems/Questions*, pp. 434- 444, APSCE.

Fabic, G., Mitrovic, A., Neshatian, K. (2017b) Investigating the Effectiveness of Menu-Based Self-Explanation Prompts in a Mobile Python Tutor. *Proc. 18th Int. Conf. Artificial Intelligence in Education*, (pp. 498-501).

Harrington, B., & Cheng, N. (2018). Tracing vs. Writing Code: Beyond the Learning Hierarchy. In *Proc.49th ACM Technical Symposium on Computer Science Education* (pp. 423-428). ACM.

Ihantola, P., Helminen, J., & Karavirta, V. (2013). How to study programming on mobile touch devices: interactive Python code exercises. In *Proc.13th Koli Calling Int. Conf. Computing Education Research* (pp. 51-58). ACM.

Johnson, C. I., & Mayer, R. E. (2010). Applying the self-explanation principle to multimedia learning in a computer-based game-like environment. *Computers in Human Behavior*, 26(6), 1246-1252.

Kumar, A. N. (2018). Epplets: A Tool for Solving Parsons Puzzles. In *Proc.49th ACM Technical Symposium on Computer Science Education* (pp. 527-532). ACM.

Lister, R., Clear, T., Bouvier, D. J., et al. (2010). Naturally occurring data as research instrument: analyzing examination responses to study the novice programmer. *ACM SIGCSE Bulletin*, *41*(4), 156-173. ACM.

Lister, R., Fidge, C., & Teague, D. (2009). Further evidence of a relationship between explaining, tracing and writing skills in introductory programming. *ACM SIGCSE Bulletin*, *41*(3), 161-165. ACM.

Lopez, M., Whalley, J., Robbins, P., & Lister, R. (2008). Relationships between reading, tracing and writing skills in introductory programming. In *Proc. 4th Int. workshop on computing education research* (pp. 101-112). ACM.

McLaren, B. M., van Gog, T., Ganoe, C., Karabinos, M., & Yaron, D. (2016). The efficiency of worked examples compared to erroneous examples, tutored problem solving, and problem solving in computer-based learning environments. *Computers in Human Behavior*, 55, 87-99.

Mitrovic, A. (2005) Scaffolding Answer Explanation in a Data Normalization Tutor. *Facta Universitatis*, 18(2), 151-163.

Parsons, D., & Haden, P. (2006). Parson's programming puzzles: a fun and effective learning tool for first programming courses. In *Proc. 8th Australasian Conference on Computing Education* (pp. 157-163).

Thompson, E., Luxton-Reilly, A., Whalley, J. L., Hu, M., & Robbins, P. (2008). Bloom's taxonomy for CS assessment. In *Proc. 10th Conf. Australasian computing education* (pp. 155-161).

Venables, A., Tan, G., & Lister, R. (2009). A closer look at tracing, explaining and code writing skills in the novice programmer. In *Proc. 5th Int. workshop on Computing education research* (pp. 117-128). ACM.

Weerasinghe, A., & Mitrovic, A. (2006) Facilitating deep learning through self-explanation in an open-ended domain. *Knowledge-based and Intelligent Engineering Systems*, IOS Press, 10(1), 3-19.

Wylie, R. & Chi, M.T.H. (2014) The Self-Explanation Principle in Multimedia Learning, in Mayer, R.E. (ed.) *The Cambridge Handbook of Multimedia Learning*. Cambridge: Cambridge University Press, pp. 413–432.