

A Comparison of Different Types of Learning Activities in a Mobile Python Tutor

Geela Venise Firmalo FABIC*, Antonija MITROVIC & Kourosh NESHATIAN

Computer Science and Software Engineering, University of Canterbury, New Zealand

*geela.fabic@pg.canterbury.ac.nz

Abstract: Programming (i.e. coding) is becoming one of the skills expected for successful careers in the knowledge economy¹, and is being taught at all levels, including primary and secondary schools. Programming skills are difficult to acquire, as the student needs to learn the specific programming language and many related concepts to write good programs. We present PyKinetic, a mobile tutor for Python that serves as a complement to traditional courses. The overall goal of our project is to design learning activities that maximize learning. In this paper, we present several types of learning activities designed for PyKinetic. The first version of the tutor implemented Parsons problems with incomplete lines, which support code-understanding and code-writing skills. The second version of PyKinetic included various types of activities aimed at code-tracing and code-writing skills. The results of two studies we conducted show that Parsons problems are beneficial for novices, while advanced students benefitted more from learning activities which required them to identify and fix incorrect lines of code.

Keywords: Mobile Python tutor, Parsons problems, self-explanation, code writing, code tracing, code understanding

1. Introduction

Smartphones are mainly used for communication, but are also widely used for leisure and entertainment as they provide ubiquitous access to most services. Smartphones may also prove to be an effective learning platform. In New Zealand, 72% adults have access to or own a smartphone and/or a laptop (Research New Zealand, 2015). A trend also emerged that 59% of people with multiple devices prefer using smartphones. Furthermore, the same study revealed that millennials (aged 18-34) are the highest smartphone users in New Zealand at 91%. Most novice programmers nowadays are millennials who expect fast-paced interactions (Oblinger and Oblinger, 2005). Educators therefore need to investigate alternative avenues of learning that could be more effective and appealing to millennial novice programmers. As trends continue, smartphones may prove to be an effective platform to learn programming, as it provide opportunities to learn while “on the go”.

Python is widely used nowadays as a programming language in introductory programming courses (Guo, 2013). We present PyKinetic (Fabic, Mitrovic and Neshatian, 2016a; 2016b), a mobile Python tutor, developed using Android SDK to teach Python 3.x programming. The tutor is a complement to traditional lecture and lab-based introductory programming courses. Being a mobile tutor, we hope that it would appeal better to a new generation of students, compared to desktop or Web-based educational tools. Traditional code writing exercises may be difficult on a small-screened device such as a smartphone, as the keyboard usually obstructs half of the smartphone screen. For that reason, we have designed learning activities for PyKinetic that require only tap and long-click interactions.

The overall aim of our project is to design activities that will maximize learning (Fabic, Mitrovic and Neshatian, 2017). In this paper, we present a set of learning activities that focus on code-understanding, code-tracing and code-writing skills. The first version of PyKinetic (PyKinetic_IncLOCs) contains only Parsons problems (Parsons and Haden, 2006), which are exercises requiring the student to re-arrange a given set of randomized Lines Of Code (LOCs) to produce the expected outcome. This version of the tutor contains Parsons problems with incomplete LOCs, in which

¹ <https://obamawhitehouse.archives.gov/blog/2016/01/30/computer-science-all>; www.codefuture.org

the student needs to re-arrange the LOCs and complete the missing elements. In this way, PyKinetic_IncLOC supports both code understanding and code writing skills. To further support learner engagement, we have also added self-explanation prompts. Self-explanation (SE) was defined as generating inferences and interpretations from principles not taught explicitly by the material, and has been shown to increase learning (Chi et al., 1989; Wylie and Chi, 2014). We have conducted an experimental study investigating the learning effectiveness of PyKinetic_IncLOCs. Our hypothesis was that PyKinetic_IncLOCs would be successful in supporting learning (S1_H1). We also hypothesize that self-explanation prompts would result in additional learning benefit (S2_H2).

The second version (PyKinetic_DbgOut) contains debugging and output prediction problems, designed to support acquisition of debugging and code tracing skills. We conducted a study using this version, hypothesizing that the combination of activities in this version of the tutor will also be beneficial for learning Python programming (S2_H1).

The goal of this paper is to compare the learning effectiveness of the two versions of the tutor, focusing on types of students who benefitted from those learning activities. The paper is structured as follows. In the following section, we present related work on Parsons problems. Section 3 presents the study conducted using PyKinetic_IncLOCs. Section 4 presents learning activities in PyKinetic_DbgOut, as well as the results of Study 2. Section 5 compares the results from both studies and discusses what kind of students benefitted from the learning activities in those two studies.

2. Parsons Problems

Parsons problems were originally proposed as a fun way for learners of Turbo Pascal to improve their syntactic skills (Parsons and Haden, 2006). These puzzle-like activities are suitable for novices, as they already contain syntactically correct code that only needs to be put in the right order. Variations of Parsons problems include extra LOCs (*distractors*), or incomplete LOCs which require the learner to provide missing elements. The latter is implemented in the version of PyKinetic presented in this paper.

There were other variants of Parsons problems considered (Denny et al., 2008). Two variants did not contain any distractors, and the only difference between the two is that one of them includes scaffolding such as curly braces and indentation (since this was used in the context of Java), while the second variant does not provide any. Two other variants (available with and without scaffolding), provided paired options for each LOC which were given in randomized order, with the paired options given right next to each other. The last variant still contains pairs of options for each LOC. However, every LOC including the paired options, were given in a randomized order. It was not specified whether the last variant was presented with or without scaffolding, but this variant ended up being discarded as it was perceived to be unreasonably difficult. For example, because of the paired options, seven lines of code for the puzzle becomes 14. Having twice the amount of LOCs in a completely randomized order may be overwhelming for students.

More research is encouraged to pinpoint accurately which skills benefit most by Parsons Problems. Some believe Parsons problems to be simpler than code tracing (Lopez et al., 2008) while some find that based on its complexity, Parsons problems lie between code tracing and code writing (Lister et al., 2010). Code writing requires higher order skills, while code tracing falls into lower categories in Bloom's taxonomy (Thompson et al., 2008). Moreover, a weak correlation was found (Denny et al., 2008) between scores on Parsons problems with code tracing questions, and a moderate positive correlation with code writing. Denny et al. (2008) suggested that Parsons problems were similar to code writing. A recent study (Morrison et al., 2016) found that Parsons problems pose a lower cognitive load compared to code writing, which may be due to the correct syntax given in Parsons problems. However, this may not be always true, as Parsons problems may require higher cognitive load depending on the type, complexity, and interface used (on paper or on a device). Moreover, there are opinions that the position of Parsons problems in the hierarchy of programming skills can vary, depending on their type (with or without distractors) and complexity (Ihantola and Karavirta, 2011). More factors that could influence learning are scaffolding and feedback provided.

3. Study 1

3.1 Parsons Problems in PyKinetic_IncLOCs

We have chosen Parsons Problems as the first activity for PyKinetic as they are simple activities for novice programmers, also suitable for a smartphone interface. Parsons Problems in PyKinetic are completed by dragging and dropping single LOCs in the correct order. Karavirta, Helminen, and Ihantola (2012) also perceived Parsons problems to be suitable for mobile devices and have developed MobileParsons for Android and iOS. Solving Parsons problems on a smartphone means learners are not required to use the keyboard to type their answers.

PyKinetic_IncLOCs contains Parsons problems with incomplete LOCs (Figure 1, left). The student fills an incomplete line by tapping to see alternatives and selecting one of them, like the implementation by Ihantola, Helminen, and Karavirta (2013). We believe that solving Parsons problems with incomplete LOCs would enhance learners' code writing skills more than Parsons problems with or without distractors. Furthermore, a recent study (Harms, Chen and Kelleher, 2016) found that Parsons problems with distractors decrease learning efficiency of middle-school children aged 10-15.

We conducted a study with PyKinetic_IncLOCs, using 15 problems covering six Python topics: string manipulation, conditional statements, *while* loops, *for* loops, lists, and tuples. All problems required LOCs to be rearranged to match the given problem description and expected output. The student had to complete the current problem to proceed to the next problem.

The first two problems were used as practice, to familiarize participants with the mode of interaction supported by the system. The remaining 13 problems had between 3 and 16 LOCs, with a maximum of three incomplete LOCs per problem. The problems were given in a fixed order of increasing difficulty, with initial problems having a smaller number of LOCs, and a smaller number of incomplete LOCs. Furthermore, the first half of the problems focused on a single topic each, while the other problems covered at least two topics. The initial seven problems were code snippets, while latter problems were more complex: each consisted of a function with function calls. PyKinetic_IncLOCs recorded information about all actions performed by participants in system logs.

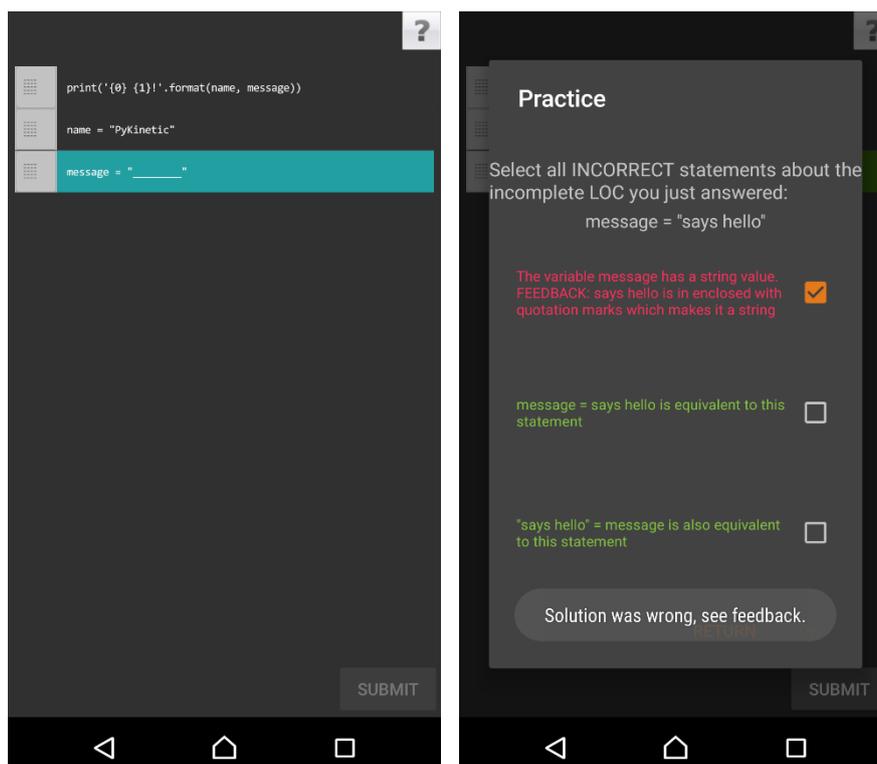


Figure 1. Example of a problem in PyKinetic_IncLOCs (left) and SE prompt (right)

To further improve learning, we introduced menu-based self-explanation (SE) prompts (Wylie and Chi, 2014). Figure 1 (left) provides an example Parsons problem, with one incomplete line (the

highlighted line). The code must be rearranged to match the given expected output, which is *PyKinetic says hello!* After completing that line, the student is given an SE prompt (right screenshot in Figure 1), which can only be attempted once to help prevent trial and error. Each SE prompt starts with the completed line (*message = "says hello"* in Figure 1, right), and asks the student to select correct (or incorrect) statements related to that line. The screenshot in Figure 1 (right) illustrates a situation in which the student selected only the first option, but that selection was wrong. There can be more than one correct option for each SE prompt. The tutor provides feedback for every wrong option (Figure 1, right), and highlights correct options in green and incorrect options in red.

There were 22 SE prompts in total, 14 of which were conceptual questions and 8 were procedural questions. All incomplete LOCs must be completed before the solution to the Parsons problem can be submitted. Learners receive two types of feedback for submitting a Parsons problem, either: "*Correct! Great job!*" for a complete solution (including completed LOCs) or "*Check the order of your solution.*"

3.2 Experimental Design

There were two conditions in the study: the only difference between the versions of *PyKinetic_IncLOCs* presented to the control and experimental group was the additional SE prompts in the experimental condition. Our first hypothesis was that all participants, irrespective of the group, would improve their Python skills by solving Parsons problems (S1_H1). Secondly, SE prompts would help experimental group participants learn more than control group (S1_H2).

We recruited 47 volunteers enrolled in COSC121, an introductory programming course at the University of Canterbury (UC), and 13 volunteers from a local high school (HS). The high school participants were taking a Year 13 course on Digital Technology. We also recruited 23 volunteers enrolled in an introductory computing course at the Ateneo de Manila University (ADMU). There were 83 participants in total, randomly assigned into two groups: experimental group (with SE prompts) and control group (without SE prompts). The study was approved by the school principal, and Human Ethics committees of UC and ADMU.

Each participant participated in a group session that lasted for 1.5-2 hours. There were one up to 13 participants per session. At the start of each session, the participants were introduced to the study and provided informed consent. Afterwards, a 15-minute pre-test was administered on paper, and the participants were instructed on how to download and install the tutor. After working with the tutor, the participants received a 15-minute post-test, also administered on paper. Lastly, instructions were given on uninstalling and deleting the application. Some participants used their own Android smartphones, while we provided phones to other participants.

The UC participants have learned previously all topics covered in *PyKinetic_IncLOCs* in their course. Although the ADMU participants have not learned about tuples, they were advised to attempt all 15 problems (including practice questions) within the time limit of an hour. We have later confirmed that the ADMU participants were not disadvantaged on the pre-test, as there were no statistically significant differences between the results of UC and ADMU participants on the pre-test question about tuples. High school participants were instructed to attempt 13 problems only, because they have not learned about tuples, and we had very limited time constraints with the high school (the sessions were conducted during strict time-scheduled periods of 50 minutes). Due to the same reason, high school participants received pre-tests on a different day of the same week, and the rest of the study was conducted on another day in 50 minutes.

The pre/post-test had eight questions each: six conceptual questions (6 marks) and two procedural questions (2 marks). The conceptual questions were multiple-choice or True/False questions. One procedural question asked the participants to predict the code output, and the other one was a Parsons problem. Questions with multiple correct answers were marked depending on the options selected. Partial marks were given for selecting correct options, and for not selecting wrong options. Partial marks were deducted for selecting wrong options. This was done to avoid discrepancy for participants who seemed to be guessing answers by selecting all options. Parsons problems from the pre/post-test were marked based on the number of LOCs written in the correct order combined with expert knowledge. Parsons problems in the tutor itself were not marked. Only the SE prompts were marked using the same marking scheme used for multiple-choice questions in pre/post-test.

3.3 Findings

There was no significant difference on the pre-test scores between the three populations of students (UC, HS, ADMU), thus showing that they had similar pre-existing knowledge. Table 1 reports the pre/post-test scores of the experimental and control groups on all questions, and on conceptual/procedural questions separately. We used the paired non-parametric Wilcoxon Signed Ranks test to verify hypothesis S1_H1. Both experimental and control groups improved scores from pre- to post-test overall (the *Improvement* row), as well as on conceptual questions (the *Improvement Conceptual* row). For procedural questions, there was no significant improvement. These results show that there is enough evidence to accept our first hypothesis S1_H1, which was that PyKinetic_IncLOCs would be effective in supporting learning.

Table 1. Statistics from Study 1 (at $p < .05$: * denotes significant; ns denotes not significant)

	Experimental (36)	Control (40)	U, p
Time/problem in tutor (min)	4 (1.57)	3.18 (1.13)	U = 502, p = .023*
Pre-test %	64.75 (18.52)	66.01 (12.34)	ns
Post-test %	75.86 (16.15)	70.56 (14.37)	U = 529.5, p = .047*
Improvement	z= -3.315, p = .001	z=-2.45, p= .014	
Cohen's d	d = .64	d = .34	
Normalized Gain %	14.94 (77.79)	4.82 (63.71)	U = 530, p = .048*
Pre-test Conceptual %	62.40 (19.05)	63.41 (14.31)	ns
Post-test Conceptual %	75.71 (16.91)	69.19 (16.71)	U = 550, p = .077
Improvement Conceptual	z=-3.221, p = .001	z= -2.37, p = .018	
Pre-test Procedural %	71.82 (26.98)	74.42 (19.48)	ns
Post-test Procedural %	76.17 (21.42)	74.58 (19.95)	ns

Table 1 also reports the results of the Mann Whitney U test for checking significant differences between the two groups. There was no difference on the pre-test scores, but the experimental group performed significantly better on the post-test (U = 529.5, $p < .05$). There was also a significant difference on the normalized gain (U = 530, $p < .05$). Both groups had a positive Cohen's d effect size, but the effect size was higher for the experimental group (experimental: d = .64; control: d = .34). These results support our second hypothesis S1_H2, which was that participants who self-explained would learn more than the control group. Participants from the experimental group spent significantly more time per problem in comparison to the control group, which was expected, as they needed to answer SE prompts ($p < .05$, U = 502).

Table 2. Effect sizes for novices/advanced students

Group	Ability	Pre-test	Post-test	Cohen's d
Experimental	Novices (20)	51.76 (12.78)	71.34 (17.48)	d = 1.28
	Advanced (16)	80.99 (9.30)	81.51 (12.68)	d = .05
Control	Novices (18)	55.23 (7.99)	65.58 (15.06)	d = .86
	Advanced (22)	74.84 (7.05)	74.64 (12.70)	d = -.02

We performed a post-hoc split of participants based on the median of the pre-test scores (66.47%). Participants who scored less than the median were labelled as novices, while the rest were considered as advanced participants. As presented in Table 4, there was a big difference between the effect sizes for novices and advanced participants in both groups. The effect sizes were very small for advanced participants in each group, while there were substantially higher effect sizes for novice participants. The novices from the experimental group obtained a higher effect size than novices in the control group (Table 2). Moreover, novices in the experimental group had a significantly higher normalized gain than novices in the control group (U=120, $p = .08$).

4. Study 2

4.1 Learning Activities in PyKinetic_DbgOut

The problems in PyKinetic_DbgOut consisted of the problem description, code (containing 0-3 incorrect LOCs), and 1-3 questions. There were five types of questions (Table 3): three types of debugging questions, and two types of output-prediction questions.

Table 3: Five Types of Debugging and Output Prediction Questions in PyKinetic_DbgOut

Type of Question	Additional Information Given	Task
Dbg_Read	Test cases with actual output	Is the code correct? (<i>Yes</i> or <i>No</i>)
Dbg_Ident	Test cases with actual output	Identify n erroneous LOCs (n is given)
Dbg_Fix	Test cases with expected output	Fix erroneous LOCs (by tapping through given choices)
Out_Act	Test cases	Select actual output of the code
Out_Exp	Test cases	Select expected output of the code

In *Dbg_Read* questions, the learner is given some test cases with the actual output; the learner's task is to specify whether the given code is correct or not. The second type of debugging questions (*Dbg_Ident*) provides similar information to the learner, but requires the learner to identify one or more incorrect LOCs. An example is shown in Figure 2 (left screenshot), where the student needs to identify two incorrect lines (the lines the student selected are highlighted in blue). The third type of debugging questions is *Dbg_Fix*, which starts with requiring the student to identify incorrect lines (*Dbg_Ident*), and then to fix them (Figure 2, right). To fix incorrect LOCs, the student needs to select the correct option from given choices. In the screenshot shown in Figure 2 (right), the student has completed the line highlighted in green, and is working on the other line (highlighted in orange).

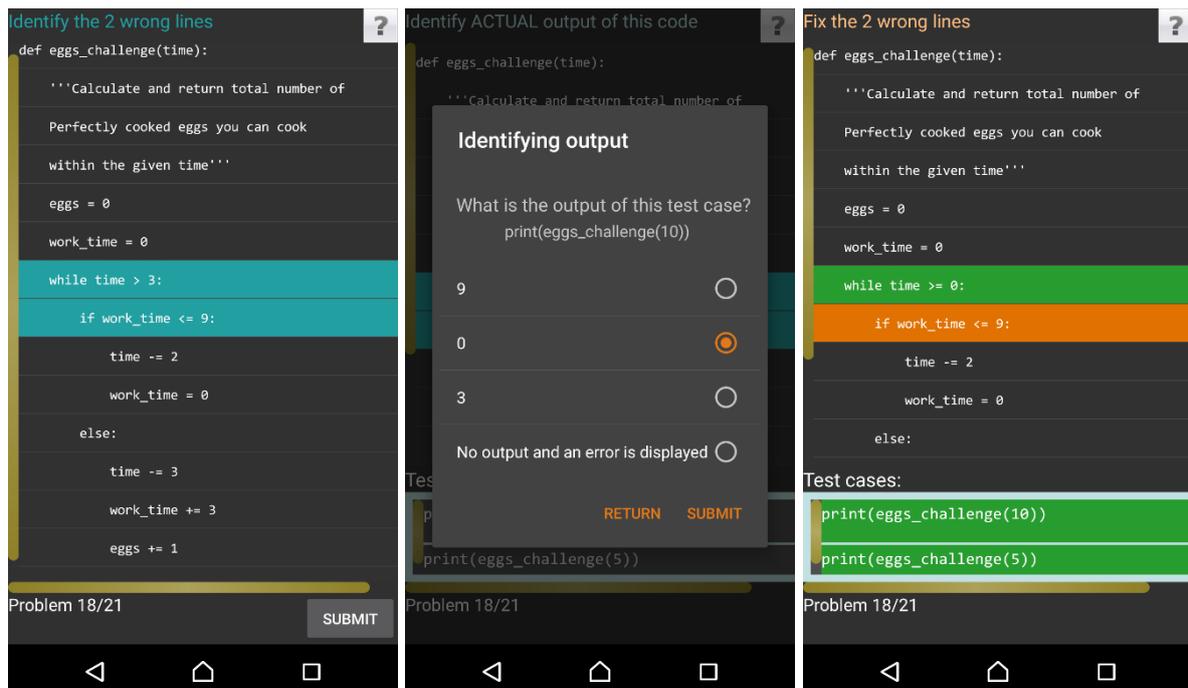


Figure 2. Screenshots from PyKinetic_DbgOut (left *Dbg_Ident*, middle *Out_Act*, right *Dbg_Fix*)

Each output-prediction question contains 1-3 test cases. In the first type (*Out_Act*), the student needs to specify the actual output of the given code for each given test case (Figure 2, middle). For example, if the code is erroneous the actual output may be none with an error displayed (Figure 2,

middle, last option). On the contrary, in *Out_Exp* questions, the student specifies the expected code output matching the problem description.

PyKinetic_DbgOut had 21 problems provided in a fixed order. There were seven levels of complexity, each containing 2-4 problems (Table 4). Problems on levels 1-3 cover conditionals, string formatting, tuples and lists; these problems consist of 4-8 LOCs (excluding function definition, comments, and test cases), and only one question. For example, problem one is a code-reading problem, containing only a *Dbg_Read* question. The complete code, problem description, and test cases with function calls are given; the task is to identify if the code is correct or not.

Every problem in levels 4-7 each contained 2-3 questions, and covered same topics plus *for* loops, *while* loops, and importing a module. Each problem on these levels started by requiring the student to identify incorrect LOCs (*Dbg_Ident*). After that, levels four and five had output prediction questions next: identifying the actual output (*Out_Act*) for level four, and identifying the expected output (*Out_Exp*) for level five. Level six targets code writing skills, by requiring the student to fix erroneous LOCs (*Dbg_Fix*) in the second question. Lastly, level 7 contains three types of questions in each problem: identifying erroneous LOCs (*Dbg_Ident*), identifying actual output (*Out_Act*) and fixing erroneous LOCs (*Dbg_Fix*). The problem illustrated in Figure 2 belongs to level 7. It is important to note that the ordering of the problems is not solely reliant on the number of LOCs and topics involved in the problem. In some cases, the code and/or the problem itself may be more logically complex than others even though it had fewer lines and topics.

Table 4: Combinations of Questions in Levels 1-7

Level	Problems	Additional Information Given	Topics Covered	Number of LOCs
1	<i>Dbg_Read (2 problems)</i>	Test cases with actual output	Conditionals	4-6
2	<i>Dbg_Ident (4 problems)</i>	Test cases with actual output	String Formatting and Conditionals	4-8
3	<i>Out_Act (4 problems)</i>	Test cases	String Formatting, Conditionals, List, Tuples	4-8
4	<i>Dbg_Ident -> Out_Act (2 problems)</i>	Test cases	String formatting, Conditionals, List, Tuples, For loops	10
5	<i>Dbg_Ident -> Out_Exp (2 problems)</i>	Test cases	String formatting, Conditionals, Lists, For loops	8-9
6	<i>Dbg_Ident -> Dbg_Fix (3 problems)</i>	Test cases with expected output	String formatting, Conditionals, Lists, For/While loops, Importing a module	9-11
7	<i>Dbg_Ident -> Out_Act -> Dbg_Fix (4 problems)</i>	Test cases	Nested While loops, Conditionals, Lists, Tuples and String Formatting	11-16

4.2 Experimental Design

We conducted a study with PyKinetic_DbgOut, with recruited 37 participants enrolled in COSC121, which was the same course where we recruited most of our participants for Study 1 (Section 3.2). We have eliminated data about two participants as they have not finished the study. The sessions were two hours long, with 1-9 participants per session. The participants provided informed consent, followed by an 18-minute pre-test, which included questions on demographics and programming background. We then gave brief instructions on using the tutor, and provided Android smartphones with the tutor already installed. Participants interacted with the tutor for roughly an hour. Lastly, participants were given an 18-minute post-test either when time had run out or when they had finished all problems. The post-test included open-ended questions for comments and suggestions about the tutor. The study was approved by the Human Ethics Committee of the University of Canterbury.

The topics covered in the study have previously been covered in COSC121 lectures. The pre- and post-tests had six questions each and were administered on paper. The tests contained same types of questions from Table 3 (worth one mark each), and additionally a code-writing question (worth 5 marks). The participants were not used to doing any programming exercises on paper, because all lab quizzes and assessment in COSC121 are completed using computers. Therefore, code syntax on their pre/post-test were not strictly penalized. There were no multiple-choice questions in the pre/post-tests. The code-writing question provided the problem description, test cases with expected output, function definition statement and the docstring. The code-writing question from both tests had an ideal solution of 5 LOCs (without any comments), which was the reason for a maximum of 5 marks on this question. The participants did not receive scores for the problems completed in the tutor.

4.3 Findings

The results from Study 2 are presented in Table 5. There were no significant differences on any reported measures. The problems the participants were solving in the tutor are of different nature to the problems they were used to in the course, where they were asked to write code. For that reason, we investigated whether there is a difference between the participants based on their code-writing skills. Before Study 2, the participants were assessed in a COSC121 lab test, which consisted of 20 code-writing questions. The median score on the lab test was 79%. We therefore divided the participants post-hoc into two groups based on the lab test median: we refer to the 16 participants who scored less than 79% as novices, and to the 19 participants who scored 79% or higher as the advanced students.

Table 5. Pre/post-test scores (%)

Question	Pre-test%	Post-test%
Total score	68.55 (23.28)	72.88 (24.67)
Dbg_Read	77.14 (42.6)	74.29 (44.34)
Dbg_Ident	88.57 (32.28)	80 (40.58)
Dbg_Fix	57.14 (46.8)	60 (44.64)
Out_Act	93.57 (23.75)	90.71 (26.49)
Out_Exp	89.05 (23.89)	78.1 (30.99)
Code Writing	56 (40.09)	69.14 (36.17)

Table 6 presents the results for novices and advanced students. Advanced students were expected to perform better than novices, as they started with a higher level of existing knowledge. Indeed, that was the case: advanced students outperformed novices by completing more problems ($U=35$, $p < .05$) and by getting higher pre/post-test scores. Although the overall pre-test score was significantly different for the two subgroups of students, this was not the case with the score on the code-writing question alone, where there was no significant difference between novices and advanced students. Furthermore, only advanced students improved their score on the question for fixing erroneous code ($z= -2.51$, $p=.012$) and on the code writing question ($z= -2.07$, $p=.039$); the novices have not improved on those questions. Lastly, it seemed that output prediction questions were unfavorable for the learning of advanced students, as there was a significant difference between novices and advanced on the pre-test but no significance in the post-test.

We investigated further on whether there was a correlation between the average time spent per completed problem and the normalized gains. The normalized gain on all questions for novices was moderately positively correlated to the time spent per problem ($r = 0.52$, $p < .05$), but the correlation was not significant for advanced students. Contrary to that, there was a strong positive correlation between the normalized gain on only code-fixing and code-writing questions, and time spent per problem for advanced students ($r = 0.7$, $p < .001$), and no significant correlation for novices.

5. Discussion and Conclusions

We presented two versions of PyKinetic, and the findings from the two studies. In Study 1, all participants interacted with Parsons problems with incomplete LOCs, but the experimental group participants additionally had to answer SE prompts. The results from Study 1 supported our hypotheses: that PyKinetic_IncLOCs was successful in supporting learning (S1_H1), and that the SE prompts provide additional learning benefits (S2_H2). Both groups improved their scores on conceptual questions from pre- to post-test: a potential explanation for this improvement may be that Parsons problems contribute to the acquisition of conceptual knowledge. Furthermore, SE prompts have been shown in previous studies to contribute to conceptual knowledge (Najar, Mitrovic and McLaren, 2016).

Table 6. Novices vs. Advanced Students (at $p < .05$: * denotes significant; ns denotes not significant)

Measure	Novices (16)	Advanced (19)	U, p
Completed problems	19.63 (1.54)	20.53 (1.17)	U= 35, p= .037*
Time/problem (min)	2.67 (.80)	2.82 (.44)	ns
Pre-test (%)	58.39 (21.09)	77.11 (22)	U= 76, p= .011*
Post-test (%)	57.92 (28.3)	85.48 (10.75)	U= 63.5, p= .003*
Improvement	ns	ns	
Normalized gain	-.03 (.7)	.13 (.74)	ns
Pre-test Dbg_Fix	34.38 (46.44)	76.32 (38.62)	U= 77, p= .012*
Post-test Dbg_Fix	34.38 (42.7)	81.58 (34.2)	U= 62, p= .002*
Improvement Dbg_Fix	ns	z= -2.51, p=.012	
Pre-test Code Writing	45 (37.59)	65.26 (40.74)	ns
Post-test Code Writing	46.25 (41.13)	88.42 (14.25)	U= 76.5, p= .011*
Improvement Code Writing	ns	z= -2.07, p=.039	
Pre-test Output Questions	84.11 (19.62)	97.37 (7.88)	U= 84.5, p=.024*
Post-test Output Questions	84.9 (17.86)	83.99 (21.17)	ns
Improvement	ns	z=-2.29, p=.022	

Furthermore, Parsons problems with incomplete LOCs proved to be effective for novice learners, but showed no effect for advanced learners. This outcome is consistent with Morrison et al. (2016), who found Parsons problems posed a lower cognitive load compared to code writing. Furthermore, based on our own experience in teaching Python, lower performing students usually have difficulties in writing their own code. Parsons problems provide sufficient scaffolding as the complete code is given with correct syntax, which only requires re-arranging. It is possible that advanced learners did not benefit from using PyKinetic_IncLOCs because they already had mental models of solutions. Hence, it is probably easier for advanced learners to write their own code based on their mental model. We have observed this with experts, when investigating strategies used in solving Parsons problems with distractors (Fabic, Mitrovic and Neshatian, 2016b).

Our hypothesis (S2_H1) for the study conducted with PyKinetic_DbgOut was that a combination of debugging and output prediction problems would also be effective for learning. However, our results revealed no significant improvement between pre- and post-test for all participants. This outcome might be due to the small number of participants in the study. Delving deeper, we found that, contrary to Study 1, PyKinetic_DbgOut proved to be more beneficial to advanced students, and showed no effect for novices. Code-writing and code-fixing exercises revealed to be more suitable for advanced students. Furthermore, a strong positive correlation was found for advanced students between the normalized gain on only debugging and code-writing questions and time spent per problem, but no significant correlation for novices. The correlations suggest that advanced students improve their code-writing skills more as they spend longer time on each problem, but minimal effect for novices. This was possibly because novices require more support.

The findings from the two studies would enable us to develop and adaptive version of PyKinetic. We plan to add a student model, which would be initialized based on the student's result on the pre-test. The student model would be updated with every activity the student performed, and will enable the tutor to select learning activities for the student tailored per his/her student model.

Acknowledgements

We thank Dr. Ma. Mercedes Rodrigo and her team from Ateneo de Manila University for collaborating with us on Study 1, and Mr. Patrick Baker and Middleton Grange School for allowing us to conduct our study with their students.

References

- 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., Mitrovic, A., & Neshatian, K. (2016a). Towards a Mobile Python Tutor: Understanding Differences in Strategies Used by Novices and Experts. *Proc. 13th Int. Conf. Intelligent Tutoring Systems*, (vol. 9684, pp. 447-448). Springer.
- Fabic, G., Mitrovic, A., & Neshatian, K. (2016b) 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. V. F., Mitrovic, A., & Neshatian, K. (2017). Learning with Engaging Activities via a Mobile Python Tutor. In *Proc. International Conference on Artificial Intelligence in Education* (pp. 613-616). Springer, Cham.
- Guo, P. J. (2013). Online Python tutor: embeddable web-based program visualization for CS education. In *Proc. 44th ACM technical symposium on Computer science education* (pp. 579-584). ACM.
- Harms, K. J., Chen, J., & Kelleher, C. (2016). Distractors in Parsons Problems Decrease Learning Efficiency for Young Novice Programmers. In *Proc. ACM Conference on International Computing Education Research* (pp. 241-250). 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.
- Ihantola, P., & Karavirta, V. (2011). Two-dimensional parson's puzzles: The concept, tools, and first observations. *Journal of Information Technology Education*, 10.
- Karavirta, V., Helminen, J., & Ihantola, P. (2012). A mobile learning application for parsons problems with automatic feedback. In *Proc. 12th Koli Calling Int. Conf. Computing Education Research* (pp. 11-18). 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.
- 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.
- Morrison, B. B., Margulieux, L. E., Ericson, B., & Guzdial, M. (2016). Subgoals Help Students Solve Parsons Problems. In *Proc. 47th ACM Technical Symposium on Computing Science Education* (pp. 42-47). ACM.
- Najar, A. S., Mitrovic, A. & McLaren, B. M. (2016). Learning with Intelligent Tutors and Worked Examples: Selecting learning activities adaptively leads to better learning outcomes than a fixed curriculum. *User Modeling and User-Adapted Interaction*, 26, 459-491.
- Oblinger, D., & Oblinger, J. L. (2005). 2 Is It Age or IT: First Steps Toward Understanding the Net Generation. In Oblinger, D., Oblinger, J. L., Lippincott, J. K. (Eds.), *Educating the next generation*. Boulder, Colorado: EDUCAUSE., pp. 12-31.
- 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-Volume 52* (pp. 157-163). Australian Computer Society.
- Research New Zealand (2015). A Report on a Survey of New Zealanders' Use of Smartphones and other Mobile Communication Devices.
<http://www.researchnz.com/pdf/Special%20Reports/Research%20New%20Zealand%20Special%20Report%20-%20Use%20of%20Smartphones.pdf>
- 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-Volume 78* (pp. 155-161). Australian Computer Society.
- 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.