UNIVERSITY OF CANTERBURY

Designing Pedagogically Effective Activities for Learning Programming in a Mobile Tutor

---

A thesis submitted in partial fulfilment of the requirements for the Degree of
Doctor of Philosophy in Computer Science

by Geela Venise Firmalo Fabic
Married name: Geela Venise Fabic Chee

supervised by

Professor Antonija Mitrovic

Dr. Kourosh Neshatian

examined by

Professor Peter Brusilovsky

University of Pittsburgh

Professor Anthony Robins
University of Otago

2019

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

My life has gone through unexpected twists and turns while doing my PhD research. Therefore, I would like to take this opportunity to express my heartfelt gratitude to my supervisors, colleagues, friends, and family.

First of all, I would like to thank my supervisor Prof. Tanja Mitrovic for her invaluable input and support. I am very privileged to have worked with her. I am grateful for her guidance throughout my research, her patience, and understanding especially through difficult seasons of my life. I would also like to thank my associate supervisor Dr. Kourosh Neshatian for his knowledge in the Python programming aspect of the project. I am very blessed and indebted to have them both as my supervisors, I would not have chosen anyone else.

I would also like to thank my colleagues: Jay Holland, Dr. Enos Chen, Dr. Moffat Matthews, Ted Ahmadi, and Ashish Sharma for their feedback and support. Thank you for the thought-provoking discussions and for helping me test versions of PyKinetic. For Dr. Enos Chen and Ted Ahmadi, thank you for helping me conduct my evaluation studies by distributing smartphones and paper forms to participants. Thank you to Dr. Richard Lobb for helping me in setting up the online pre-/post-tests for my third evaluation study. Thank you also to Prof. Tim Bell for his suggestions on the computer science education components of my research. Thank you to my other fellow PhD students who have inspired me, and spurred me onwards especially Dr. Caitlin Duncan, Dr. Mengmeng Ge, Dr. Tieta Putri, Dr. Josh Mculloch. I would also like to thank other staff in the CSSE department for the support and friendship especially Dr. Walter Guttman, Yalini Sundralingam, Paul McKneown, and Liam Laing. Thank you to some of the staff (Python tutors) who participated in my pilot study.

Thank you to the students who participated in my evaluation studies from University of Canterbury, Ateneo de Manila University, and Middleton Grange High School. Thank you to Dr. Ma. Mercedes Rodrigo and her team from Ateneo de Manila for her collaboration on my first evaluation study. Thank you to Mr. Patrick Baker of Middleton Grange School for allowing me to conduct evaluations with his students in my first and third evaluation study.

To Dr. Calvin Fung Chye Lim, my late fiancé who tragically passed away amid my PhD study, thank you. I thank him for his love, care, and encouragement as a partner, best friend, and a fellow PhD student. I also thank his family.

I would also like to thank my parents Joy and Vinsant, and my siblings Gayle and Jovin. Thank you for always being there for me and jovially encouraging me forward. Thank you to my friends especially Angeli, Stephen, Likha, Paul, Tanya, and Hannah who cared for me and provided a listening ear when I needed.

Last but certainly not the least, I sincerely express my love and gratitude for my husband, Doctor Brian Nee Hou Chee. Thank you for taking a leap of faith with me, to embark on this journey despite me being in the most stressful part of my PhD. Thank you for your deep-rooted love and care. Thank you for appreciating my work despite our career differences. Thank you for your patience and understanding, and for encouraging me in a way that only you can. Thank you for making me the happiest I have ever been. I love you so much.

Most importantly, I would like to thank God for everything. All glory and honour to Him alone.

ABSTRACT

Smartphones are engaging and powerful devices which provide opportunities for ubiquitous learning. My PhD research project focuses on developing learning activities for programming in a smartphone application. I present PyKinetic–a mobile tutor for learning Python programming targeted at introductory programming students. The project is interdisciplinary as it uses foundational principles from several disciplines: computer science education, educational psychology, human computer interaction, and software engineering. This work is in the field of computer science education, as I developed and designed activities and strategies for learning programming in a mobile tutor. I achieved this by applying some theories from educational psychology and human computer interaction, and finally, I built PyKinetic from null and developed it with the guidance of basic software engineering principles.

This research is aimed at teaching Python programming because it is one of the most popular languages used to teach introductory programming. There are several educational systems that are successful in teaching programming. However, my research aims to bridge the gap by exploring a rather uncharted territory in teaching programming – specifically by using smartphone devices. The aim of my project is to design effective learning activities in a mobile tutor to enhance specific coding skills (code debugging and code writing). Another goal is to find direct associations between my designed activities to selectively target certain programming skills. Towards building an effective mobile tutor, another goal is to develop a novel framework for pedagogically effective programming activities, and to design activities and strategies based on that framework. The main contribution of my research includes authoring the content for PyKinetic, designing, and implementing a mobile tutor effective for enhancing Python programming skills.

The results of my studies revealed that introductory programming students enjoy learning programming via smartphones. More importantly, majority of my participants improved their coding skills when using PyKinetic even when they only used it for a short session. I also discovered that certain activities are more beneficial for students with lower prior knowledge, while other activities are more suitable for those with higher prior knowledge. Furthermore, I found benefits for combining programming tasks of different nature, and that it gives learners the opportunity to learn multiple coding skills within one learning encounter. I also found evidence of the hierarchy of coding skills in learning programming, which was astoundingly diverse when students with different knowledge levels were compared. Lastly, I found evidence that a mobile programming tutor is more effective with adaptive problem selection.

UNIVERSITY OF CANTERBURY
Te Whare Wānanga o Waitaha
CHRISTCHURCH NEW ZEALAND

Deputy Vice-Chancellor's Office
Postgraduate Office

Co-Authorship Form

This form is to accompany the submission of any thesis that contains research reported in co-authored work that has been published, accepted for publication, or submitted for publication. A copy of this form should be included for each co-authored work that is included in the thesis. Completed forms should be included at the front (after the thesis abstract) of each copy of the thesis submitted for examination and library deposit.

Please indicate the chapter/section/pages of this thesis that are extracted from co-authored work and provide details of the publication or submission from the extract comes:
Chapter 5-7 and Publications in Appendices

Please detail the nature and extent (%) of contribution by the candidate:
80%

Certification by Co-authors:
If there is more than one co-author then a single co-author can sign on behalf of all
The undersigned certifys that:

- The above statement correctly reflects the nature and extent of the PhD candidate's contribution to this co-authored work
- In cases where the candidate was the lead author of the co-authored work he or she wrote the text

Name: *Tanja Mitrovic* Date: *12/6/2019*

GLOSSARY

Terminology	Description
LOC	Line of Code
UI	User interface
Parsons Problems	Exercises composed of a set of randomised lines of code which the student needs to rearrange in the correct order by dragging and dropping to form a correct snippet of code.
Regular Parsons problems	My variant of Parsons problems where lines of code are rearranged in the same area, instead of dragging and dropping LOCs from a problem area to a solution area. All LOCs in this variant consists of a single LOC per puzzle block with correct indentation.
Parsons problems with distractors	Parsons problems with extra LOCs not needed in the correct solution.
Distractors	Extra LOCs not needed in a Parsons problem
Parsons problems with incomplete LOCs	Parsons problems with one or more of its LOCs with missing elements.
Incomplete LOC	A LOC with one or more missing elements identified by a blank space.
COSC121	Introduction to Computer Programming, an introductory course to Python programming in the University of Canterbury.
IDE	Integrated Development Environment
Android SDK	Android Software Development Kit
ADT	Android Development Tool, a plugin which allows integration of Android SDK with Eclipse IDE.
MVC	Model-View-Controller a design pattern used in software development.

XML	Extensible Markup Language
Android application screen or application screen	A page or a window within an Android application which is often presented as full-screen but can be embedded between other pages. This is technically called an "activity" in Android terminology, but it is denoted as Android application screen or application screen in this thesis to not be confused with a pedagogical "activity".
ER diagram	Entity Relationship diagram
Edit text element	An input text element in an Android application where it is usually used for users to enter alphanumeric characters.
SE	Self-explanation prompts designed to promote deeper learning
LP	Low prior knowledge students
HP	High prior knowledge students

1 INTRODUCTION

1.1 Motivation

Learning involves the acquisition of new or reinforcing of existing knowledge or skills. To learn successfully, an individual must engage with the learning activity. More specifically, learning programming is challenging: the student is required to learn the syntax and semantics of the programming language and understand its purpose in context, perform problem solving tasks, logical thinking, as well as develop design skills and strategies (Linn and Dalbey, 1985). Aside from learning programming concepts, the student must also understand concepts behind programming; such as how code structures are compiled and translated behind the scenes, and how programs are executed. Novice programmers find it rather difficult to grasp the concepts behind certain characteristics of programming, which might lower their motivation to learn more and to practice. It has been claimed that it takes about 10 years of experience for a novice programmer to become an expert (Winslow, 1996)

Python is widely taught in introductory programming courses (Guo, 2013). It is also the number one programming language in 2018 based on IEEE Spectrum Ranking (Cass and Parthasaradhi, 2018). Mobile handheld devices (specifically smartphones) are engaging devices with distinctive features which are increasingly being utilised for learning. Pea and Maldonado (2006) summarized the unique attributes of mobile devices for learning into seven features: size and portability, small screen size, computing power and modular platform, communication ability, wide range of applications, synchronization and back-up abilities, and stylus-driven interface. These attributes are largely still relevant at this day and age, but nowadays most handheld devices, like smartphones, provide touchscreen surfaces where users can interact with directly using their bare hands without using a stylus or a thumb-pad keyboard. Smartphones are used to access course material, listen to podcasts, watch instructional videos, and communicate with peers (Dukic et al. 2015). Smartphones are also being used in classrooms, to increase interaction between the teacher and students or between students (Au et al. 2015; Anshari et al. 2017).

Smartphones have some weaknesses compared to a personal computer such as screen space. However, the focus of this research is not to probe into the effectiveness of a tutor being in a mobile device compared to other mediums. Rather, the emphasis is on investigating and developing pedagogically beneficial activities in a mobile device while effectively using the features of smartphones. I envisage that this work is useful for students, researchers and educators especially if the popularity of mobile devices and mobile applications continues to increase. As the popularity of Python and smartphones are increasing, I aim to provide opportunities for learners to continue enhancing their Python programming skills even when

they are away from personal computers. Being a mobile tutor, I hope that it would appeal to the new generation of students; allowing them to continue learning outside of a classroom setting.

Activities must be designed carefully to be pedagogically effective for a mobile tutor, while also being aware of its strengths and limitations. An example of a programming activity suitable for a mobile tutor is called a Parsons problem (Parsons and Haden, 2006). A Parsons problem consists of a set of Lines of Code (LOCs) presented in a random order, which the student needs to reorder, based on a given description and expected output. I have developed my own variations of Parsons problems in this research project. The aim of this project is to develop activities that are effective for learning programming in a mobile tutor. My work aims to bridge the gap in programming education by investigating the effectiveness of a mobile tutor for learning programming. Moreover, I investigate how I can design activities that directly target specific coding skills.

1.2 Scope of the Project and Research Questions

I present PyKinetic (Fabic, Mitrovic, and Neshatian, 2016a; 2017b), a tutoring system whose name was coined from a combination of two words: Python (programming) and kinetic. I chose the name PyKinetic as it is a tutor containing short and quick activities for learning Python programming. PyKinetic is a mobile Python tutor, developed using Android SDK to teach Python 3.x programming. PyKinetic is not meant to be a stand-alone learning resource; instead, the tutor is a complement to traditional lecture and lab-based introductory programming courses. My approach is to implement PyKinetic with a component-skills perspective (McArthur et al., 1988) as I recognise that a set of interrelated skills are necessary for learning programming. Similar with McArthur et al., I also consider the sequencing of my activities as a pedagogical strategy in PyKinetic.

This research project is interdisciplinary, but its core is in the field of computer science education. I developed activities for learning programming in a mobile tutor. Building the activities was achieved by applying foundational principles from a variety of disciplines: computer science education, educational psychology, human computer interaction, and software engineering. PyKinetic is developed by applying some theories from educational psychology and human computer interaction, and finally, I built the system – PyKinetic from null and developed it with the guidance of basic software engineering principles. I propose that this research would not be as successful as it is if I did not seek guidance from the other disciplines mentioned above. The focus of this research is to investigate the effectiveness of the activities as a teaching medium, and its pedagogical implications for novice programming learners. My intention was to conduct the evaluations as close as possible to reality, to get a full grasp of the implications of using smartphones to learn programming. Therefore, I have developed several stand-alone versions of PyKinetic for all my

evaluation studies, where participants used smartphone devices. I propose that I would not have attained a full comprehension of my evaluations if I used other alternatives such as using a smartphone emulator, or intangible mock-ups. Moreover, it is apparent that PyKinetic should be deployed for further use. However, deploying an autonomous version of PyKinetic is also outside the boundaries of my research.

The tutor is aimed at students enrolled in a first-year introductory programming course in the University of Canterbury learning Python programming. I recognise that from my focus group, there are students who have low prior knowledge which I refer to as *LP*; while those who have higher prior knowledge, I refer to as *HP*. The evaluations in this research were aimed at the activities and the tutor as an entity. The Python topics covered were: *String Manipulation*, *Conditional Statements*, *Lists*, *For Loops*, *While Loops*, *Dictionaries*, *Tuples*, and *Data Types*. More specifically, my research focuses on two fundamental programming skills: *code debugging* and *code writing*.

The following are the research questions addressed in this research project:

(R1) How does one create a framework for designing effective learning activities in a smartphone programming tutor?

(R2) What problem-solving strategies can be observed when solving Parsons problems?

(R3) Which programming skills can be enhanced by a mobile tutor?

(R4) What combination of activities is effective for learning Python programming in a mobile tutor?

(R5) What pedagogical strategies are effective for learning in PyKinetic?

(R6) Which of the activities I developed are effective for improving *code debugging* and *code writing* skills?

(R7) What is the learning effectiveness of students with lower prior knowledge (LP) compared to those with higher prior knowledge (HP)?

(R8) What is the relationship between coding skills (code debugging, code tracing, and code writing) when solving problems in a mobile tutor for LP compared to HP students?

I address (R1) by conforming with the literature presented in Chapter 2. I present the fundamental guidelines and framework I developed for PyKinetic in Chapter 3 and evaluate them in my studies in Chapters 4-7. (R2) is addressed in my pilot study in Chapter 4. (R3-R7) are addressed in my evaluation studies as I started to develop the first version of PyKinetic and continued to improve it based on my findings. (R8) is addressed in my second evaluation study in Chapter 6. Finally, Chapter 8 summarises my research questions.

1.3 Contributions

This research project aimed to develop pedagogically effective activities for learning Python programming in a mobile tutor. As part of this research, one of the main contributions is a set of guidelines for developing learning activities for programming in smartphones. Another main contribution is implementing a mobile tutor application. Moreover, I developed activities using the guidelines I designed which target code debugging and code writing skills. The following activities were developed: variants of Parsons problems, debugging activities, and output prediction activities. I also implemented my own variant of Parsons problems with incomplete LOCs where 1) blocks of code are received and solved in the same area; 2) indentations are provided as scaffolding; 3) all blocks of code contain single LOCs; 4) a menu-based self-explanation (SE) prompt (designed to facilitate deeper learning) is provided after solving an incomplete LOC. Each component of my Parsons problems variant is not unique, but the variant is unique as a unit. Furthermore, I am the first one to combine Parsons problems with SE prompts to the best of our knowledge. Also, I am contributing to SE literature by combining it with an exercise such as Parsons problems. Most SE prompts are combined with worked examples or problem-solving. But I combined it with Parsons problems which are more cognitively demanding than worked examples but less challenging than problem-solving. Evaluating SE prompts and combining it with exercises that are in the middle of the spectrum in comparison to worked examples and problem-solving is one of my contributions; evaluating SE on a mobile tutor also has not been done before to the best of our knowledge. I also investigated whether the activities I developed can specifically target specific coding skills. I found that certain activities are more effective for learners with lower prior knowledge, while others were more suitable for those with higher prior knowledge.

1.4 Thesis Structure

The first two chapters of the thesis (Chapters 1-2) envelop the underpinning knowledge of my research. Then, I present Chapter 3 which contains my design framework and early prototypes. The following four chapters (Chapters 4-7) cover the series of evaluation studies that I conducted. Finally, the last chapter (Chapter 8) binds my entire research project.

Chapter 1 aims to deliver my motivations, research goals, and contributions. Next, Chapter 2 comprises of my literature review which served as my theoretical basis for my research. My literature review covers several areas: programming, debugging, theories on multimedia design, self-explanation, mobile learning, Parsons problems, and educational systems for Python. I also present related work on educational systems for programming in Chapter 2. Furthermore, Chapter 3 presents the set of guidelines

that I have designed and used for all activities across all versions of PyKinetic. Chapter 3 also exhibits early prototypes of my learning activities.

After laying the groundwork for my research in Chapters 1-2 and presenting my design frameworks in Chapter 3, I introduce my evaluation studies in Chapters 4-7. Chapter 4 presents my pilot study with PyKinetic_Pilot. The activities evaluated in this study were Parsons problems with and without distractors (extra LOCs). Then, Chapter 5 covers my first evaluation study with Parsons problems with incomplete LOCs where there were two versions: one with supplementary menu-based self-explanation prompts (PyKinetic_IncLOCs_SE) and the other with no self-explanation (PyKinetic_IncLOCs). Chapter 6 presents other programming activities that I designed to target several coding skills (PyKinetic_DbgOut). My second evaluation study with PyKinetic_DbgOut (Chapter 6) investigated a combination of debugging and output prediction activities. My third evaluation study is covered in Chapter 7 where I developed PyKinetic_Fixed and PyKinetic_Adaptive. The latest versions of PyKinetic contained a combination of programming activities that were used in my previous studies: Parsons problems, debugging activities, and output prediction activities. Moreover, I developed and evaluated a version with adaptive problem selection (PyKinetic_Adaptive) which is also discussed in Chapter 7. Finally, the thesis ends with Chapter 8 where I present the summary of my research, contributions, limitations, and future directions.

2 RELATED WORK

This chapter presents literature review which motivated and supported various aspects of the research project and guided me to address all my research questions (Chapter 1.2). As my PhD project is interdisciplinary, I present literature from various areas such as computer science education, educational psychology, and mobile learning. I also present examples of educational systems for Python and present background on Parsons problems which were used in versions of PyKinetic covered in Chapters (4-5, 7).

2.1 Programming

Programming is writing and inputting a sequence of code instructions where it is in a suitable format that can be fed and processed by a computer (Blackwell, 2002). However, nowadays programming is more than just coding instructions as it comprises of using problem-solving skills and applying computer science concepts (Lye and Koh, 2014). Programming is similar to building various components together, and the best way to learn programming is by doing it (Areias and Mendes, 2007). Rather than a body of knowledge, programming is more like a skill, and is learned by doing tasks (Shaffer, 2005). Most programming students agree that learning by doing is most effective for learning programming (Tan, Ting, and Ling, 2009; Piteira and Costa, 2013). Most students prefer learning programming in hands-on sessions and labs, while attending lectures is ranked the least based on the perceived effectiveness for their learning (Tan, Ting, and Ling, 2009). Nowadays, most novice programmers expect fast-paced interactions (Oblinger and Oblinger, 2005). Moreover, the new generation of programming students prefer other learning avenues, like watching YouTube videos or on the web rather than by traditional methods (Piteira and Costa, 2013). Klopfer et al. (2009) recommend that educators should endeavour to leverage the advantages of emerging technologies for instructional gain towards a more seamless way of teaching that connects with the manner of students' interactions with the outside world. However, Bennett, Maton, and Kervin (2008) debate that there is no sufficient evidence that there is a widespread dissatisfaction for new generation of learners and neither there is profound proof that these learners are exhibiting a different learning style. Instead, the evidence reveals that technology plays a role in the methods in which young people use it in school and at home. Authors suggest including the opinions of new programmers and their teachers to grasp the situation better before promoting that a widespread shift is obligatory. However, as technologies are rapidly changing, I suggest that it is worth investigating whether there are alternative avenues that can be utilised nowadays for teaching programming.

Programming is difficult to learn (Jenkins, 2002; McCracken et al., 2001; Robins, Rountree, and Rountree, 2003), but widely sought out by students, as the demands of programmers continue to increase (Robins, Rountree, and Rountree, 2003). Novices are slow in solving problems due to the lack of declarative and/or procedural knowledge (Anderson, 1982). In programming, declarative knowledge includes the syntax of the programming language and familiarity with code constructs. According to (Winslow, 1996), it takes ten years for a learner to become an expert programmer. Learners often lack mental models and are unable to translate a problem into manageable tasks (Winslow, 1996). Novice programmers may know the syntax and semantics of the code, but they are unable to apply their knowledge and design their own program (Lahtinen, Ala-Mutka, and Järvinen, 2005). The difficulty in structuring code might be evidence of a deficiency of procedural knowledge. Some students perceive code as series of instructions that are anticipated to happen when the code is executed (Ahmadzadeh, Elliman, and Higgins, 2005). Also, students often do not realise that a statement is executed based on the previous instructions (Lahtinen, Ala-Mutka, and Järvinen, 2005). Students have difficulties in comprehending the execution order, predicting the output, debugging and code writing (Pea, 1986). A study was conducted with 182 first year introductory programming students to investigate the perception and difficulties of students in learning programming (Tan, Ting, and Ling, 2009). Participants were asked to rank responses using a 5-point Likert scale, from 1 (Strongly Disagree) and 5 (Strongly Agree). Results revealed that the top difficulties were: designing a program to solve certain task (avg = 3.52, sd = 0.99), dividing functionality into procedures (avg = 3.51, sd = 0.88), learning the programming language syntax (avg = 3.37, sd = 0.96), and finding bugs in one's own program (avg = 3.34, sd = 0.92). Interestingly, the perceived top difficulties by introductory programming students are consistent with literature, even though students are known to frequently overestimate their comprehension (Lahtinen, Ala-Mutka, and Järvinen, 2005).

A variety of skills necessary for programming has been discussed in the literature. Code tracing is to simulate how a program is executed by "running a mental model of a program with some input, while also running a separate mental model of a notional machine for which the program itself serves as input." (Sorva, 2013). "Code explaining" (as referred to in this thesis) is not a skill for programming, rather it is a task which requires code tracing and code reading skills. The task of code explaining is when one is presented with a code snippet and asked what it does "in plain English" (Lister et al., 2006). Researchers found that the skill of code tracing should ideally be learned before code writing (Lopez et al., 2008; Thompson et al., 2008; Harrington and Cheng, 2018). Further evidence proves relationships between doing tasks of code tracing, code writing, and code explaining (Lister et al., 2009; Venables, Tan, and Lister, 2009). A strong positive correlation was found between code tracing and code writing (Lister et al., 2010). Harrington and Cheng (2018) conducted a study with 384 students, to investigate whether a gap exists between the ability of students to trace and write code. The study was conducted in an examination setting, where students

were given two questions. The participants were randomly assigned to perform code tracing on one question and code writing on other. The results show that 56% of the students had almost no gap between their code tracing and code writing skills. For the remaining students who had of at least two out of eight marks, a strong negative correlation was found between the learners' performance in the course and the size of the gap. Regardless of whether the student was better in either code tracing or code writing, authors suggest that a large gap was more likely due to the student struggling in the course. The authors found that underachieving students were struggling with understanding the core programming concepts. The conclusion of the authors is also supported by literature, as mentioned earlier, learners need both declarative knowledge and procedural knowledge (Anderson, 1982).

2.2 Debugging

One of the most vital skills in implementing a program is debugging. Debugging is the process of finding errors in programs (Gould and Drongowski, 1974) and eradicating faults that do not comply to the specification of the program (Chmiel and Loui, 2004). To elaborate further, it involves identifying the source of faults in a program and fixing it (Yoon and Garcia, 1998). Debugging is different to testing, as it requires to explicitly find the exact location of the errors (Myers, 1978). The process of debugging is like a doctor's diagnosis. Often times a program demonstrates certain "symptoms" and the programmer needs to remedy them by finding the "disease" which causes them, and "treating" the disease by fixing the faults in the code (Yoon and Garcia, 1998).

Debugging is an integral part of implementation because a program must perform within its specifications and be reliable upon deployment and beyond. In the industry, the cost of delivering a dependable system by debugging, testing, and verification is around 50 to 75 percent of the total cost of development. Therefore, it is important to investigate the reasons on why it is expensive (Hailpern and Santhanam, 2002). Perhaps one of the reasons which attributes to the high cost of debugging in the industry is the lack of formal education on debugging provided to programming students in universities. Chmiel and Loui (2004) conducted a study to investigate whether formal training for debugging is beneficial for programming students. They found that students who completed formal debugging training were significantly faster in debugging their assignments compared to those who did not undergo the training, based on development logs and their reported time spent on debugging. The authors also verified that the significant result was not due to the two groups having a difference in abilities.

Debugging is cognitively demanding and requires both theoretical knowledge and problem-solving skills (Yoon and Garcia, 1998). Debugging is difficult because of three main reasons: a programmer normally has limited skills to simultaneously maintain several aspects of a program. Debugging also

requires a high degree of accuracy to accommodate and deliver a reliable computer system (Gould, 1975). Vessey (1985) presented four strategic goals for debugging: identify the problem, determine the purpose and architecture of the program, explore the execution of the program, and to fix the errors.

For novices, debugging is often challenging as it requires them to demonstrate new skills simultaneously (Fitzgerald et al., 2008). Novices normally have limited knowledge and only some of it could be relevant for a certain debugging task. Even expert programmers still have challenges with mastering the process of debugging (Ducasse and Emde, 1988). However, there are differences in the strategies used by novices and experts in debugging. Experts have a system view of the program and are skilful in compartmentalising the programs. Moreover, experts adapt using breadth-first strategies. On the contrary, some novices use depth-first techniques. However, some novices use breadth-first techniques but due to their lack of skills, they are unable to comprehend the overall system. Novices also exhibit sub-optimal strategies and irregular behaviour when debugging as they are also less able in breaking a program apart (Vessey, 1985). Novices display approaches to solve errors by hacking around it without fixing the root cause (Murphy et al., 2008).

There are several approaches to debugging. One debugging strategy called isolation strategy is when tests are executed to pinpoint the location of the error, then symptoms that come up are investigated and analysed, until the precise location of the error is found (Yoon and Garcia, 1998). Another similar technique for debugging is cyclic where the programmer runs the code sequentially until an error comes up. If an error was found, the programmer would add some breakpoints to trace the code to determine the cause/s of the error and execute the program again (LeBlanc and Mellor-Crummey, 1986). Other actions that could be employed to succeed in debugging include rereading the program specification, tracing the execution of the code, understanding the code, using debugging tools and resources, pattern matching and consideration of other external factors (Murphy et al., 2008).

A study focusing on the debugging patterns of novices (Ahmadzadeh, Elliman, and Higgins, 2005) found that most learners skilled at debugging were also competent programmers (66%). Furthermore, it was also revealed that only 39% of advanced programmers had good debugging skills. The participants were tested on their debugging skills by working on erroneous programs. A potential explanation for these findings is that understanding someone else's code requires a higher order of skill. Fitzgerald et al., (2008) confirmed results of (Ahmadzadeh, Elliman, and Higgins, 2005) and found that good programmers are not necessarily proficient debuggers, but programmers who are good in debugging are also competent in code writing. The ability to read and comprehend code, which was not written by the learner, will expand their abilities as a programmer. These findings provide evidence that debugging someone else's program requires a higher order of skill than code writing. I have implemented debugging and output prediction activities in the versions of PyKinetic covered in Chapters (6-7).

2.3 Cognitive Load Theory

To successfully implement pedagogically effective tasks for programming, it is essential to understand how humans learn, their capabilities and limitations. The Cognitive Load (CL) theory (Sweller, Van Merrienboer, and Pass, 1998; Sweller, Ayres, and Kalyuga, 2011) stipulates that the knowledge of human cognitive architecture is vital for instructional design. The CL theory is based on the human cognitive architecture. Humans have two memory spaces: *working memory* and *long-term memory*. Long-term memory is a large storage for information containing chunks of knowledge used not only for learning but in other tasks such as doing daily activities. The theory proposes that when attempting to perform a task, humans retrieve relevant information from their long-term memory to use it temporarily in their working memory. The working memory has limited capacity, which is why for example, telephone numbers usually only consists of seven digits. Learning is considered as successful when information temporarily stored in the working memory is transferred and stored in the long-term memory. Knowledge elements which are not transferred to the long-term memory will be forgotten.

From the standpoint of an instructional designer, having a clear understanding of working memory and long-term memory is important. The limitations of human memory set priorities on the design, particularly avoiding adding too many features that may result in unnecessary cognitive load to the learners. The difficulty of a learning task is related to the knowledge elements within the task and the amount of knowledge processing the learner is required to do. A task is typically composed of many elements. The elements involved in a particular task may be perceived differently by different individuals. This is because a group of elements may be classified as a schema on the long-term memory of some individuals. A *schema* is used by humans to combine multiple elements of information into a single chunk of knowledge (Chi, Glaser, and Rees, 1981).

The CL theory poses three types of load: *intrinsic cognitive load, germane cognitive load*, and *extraneous cognitive load* which contributes to total cognitive load. The inherent features of the learning content contribute to *intrinsic cognitive load*. On the other hand, learning processes contribute to *germane cognitive load*. Lastly, *extraneous cognitive load* is due the manner that the material is presented. The CL theory also describes how certain activities can be easy for some learners, but difficult for others. One of the reasons for this is that if an individual has experience with activity **X** several times, it is processed by retrieving, constructing and automating similar *schemas* from the long-term memory which contributes to *germane cognitive load*. The determinant for the element interactivity will be the interaction between the long-term memory and activity **X**. Retrieving and processing a schema from the long-term memory is easier than constructing a new schema by analysing the required interaction between the elements. This may happen when an individual may not have encountered activity **X** and will have to resort in either recalling

similar activities or taking each element of activity **X** and process interactivity between elements. A large quantity of interacting elements would therefore result in a heavier working memory load, thus giving the individual a perception that the activity is difficult. The interaction of the elements can be either essential to learning and contribute to *intrinsic cognitive load* or be unnecessary within that context and add to *extraneous cognitive load*. In the instructional designer's perspective, it is important to be aware of both to maximise the effectiveness of the learning material and most especially avoid adding too many components that may end up being detrimental to learning. Sometimes reducing extraneous cognitive load may not be enough or may not help at all to avoid overloading the working memory. In this case, it may be necessary to simplify the task or reduce its elements towards reducing intrinsic cognitive load. For example, instructional material may contain too many game elements or decorative features that may overload the extraneous cognitive load of learners which will leave no space for processing intrinsic and germane cognitive load. In this example, the learners will end up being distracted from more important aspects such as focusing on achieving the learning objectives.

Based on the CL theory, the difficulty in learning new concepts is not due to the absence of learning strategies, but rather it is the complexity that comes with learning the new material. This also relates back to the idea of schemas being stored in long-term memory available for retrieval and processing when similar tasks are encountered. Therefore, the difficulty involved in learning new material will also depend on the number of schemas similar to the new material obtainable from the long-term memory. Whenever individuals encounter situations familiar to them, the theory suggests that general cognitive strategies are automatically used. However, when individuals encounter unfamiliar situations the same strategies are not used. This is the reason why the theory focuses on the complexities involved within specific domains and not on learning and developing general cognitive strategies.

2.4 Multimedia Learning Theory

An instructional design is bounded within the cognitive thoughts of the designer. Therefore, it is important to maximise one's knowledge of both the cognitive load theory and multimedia learning theory for designing effective multimedia learning material. One of the guidelines of the multimedia learning theory (Mayer, 2009) is that the focus of the multimedia must be on the needs of the learners rather than its technological platform: a *learner-centred* approach rather than *technology-centred* approach. This may already be obvious to designers, but it is still important to be reminded that the main emphasis of the design should be the learning outcome and other needs of the learners. It is easy to be distracted from this and to dwell too deep on other factors such as fancy features that a certain piece of technology offers that may seem essential to the design; but in reality, only appears to be deceivingly important. In relation to this,

multimedia instructional material must also be designed and built upon the concept of human cognition and be consistent on the way the human mind works. It may be helpful to include some elements that may increase other aspects that may eventually be beneficial to learning. Examples of these are: increasing motivation, self-efficacy and attentiveness, or promoting behavioural and emotional influences aimed at increasing learning. However, evidence of high behavioural activity for instance does not guarantee that a learner is cognitively active. The theory suggests that it is possible for an individual to be cognitively active even when he/she seems to be behaviourally inactive. One of the plausible reasons why multimedia learning theory discourages including extra elements to the instructional material is that adding extra elements may contribute to extraneous cognitive load. Even though the additional elements are aimed to influence behaviour and emotions in learning, influencing such aspects does not always guarantee to maximise learning benefits. Moreover, the working memory is not limitless. Therefore, an individual may be able to process several elements at a time, but his concentration is not guaranteed to be focused on the important learning aspects. One can offer a learner a wide range of learning material but ultimately, the responsibility for learning resides within the learner himself.

There are several principles mentioned in Multimedia Learning (Mayer, 2009): *multimedia principle*, *spatial contiguity principle*, *temporal contiguity principle*, *coherence principle*, *modality principle*, *redundancy principle* and *individual differences principle*. The *multimedia principle* suggests that verbal and pictorial messages are better for learning instead of only verbal messages. The *spatial contiguity principle* suggests that space on an instructional design interface is scarce, therefore space must be maximised effectively. Having said that, the manner of presenting the messages is also important. Based on the *temporal contiguity principle,* learning is more effective when messages are presented simultaneously rather than successively. In relation to the spatial contiguity principle, the *coherence principle* suggests that extraneous elements distract students from the learning objectives. Cluttering the design with unnecessary elements that are merely ornamental can overflow the working memory load and can lead to negative effects on learning. Additional design elements may also be considered as distractions which not only clutter the design space but may also cause interference to the learner's thinking that may lead to a poor learning outcome. The learner may not perceive the extra design elements to be distracting or disadvantageous for his learning outcome. However, from the educator's perspective of the student's learning episode, this may be disadvantageous especially if this completely takes the learner off the track with the material's learning objectives.

Another principle to be taken into consideration when designing instructional material is the *modality principle.* Based on *modality principle,* learning is more effective when text is spoken verbally rather than printed to be read. In addition, based on the *redundancy principle,* learning is more effective when a combination of animation and narration is presented instead of animation, narration and printed text.

Different individuals have different ways of learning effectively. The *individual differences principle* suggests that low-knowledge learners need the instructional material to have more guidance and scaffolding, whereas high-knowledge learners do not require as much. The same principle also suggests that high-spatial learners need less effort to form mental models compared to low-spatial learners. Therefore, the differences between individuals must also be considered when designing instructional material.

Effective learning does not only involve gaining new knowledge but also reinforcing existing knowledge stored in the long-term memory that will eventually be used in the future. If only the working memory of an individual changed, and nothing was transferred to long-term memory, the new knowledge will eventually be forgotten, and it implies that nothing was learned. This relates to two of the techniques used in evaluating multimedia learning material which is to test for *retention* and *transfer* of knowledge.

2.5 Self-Explanation

The main goal of this research is to deliver effective learning activities for programming. One of the ways I have achieved this is by introducing self-explanation prompts (Chapters 5,7). Self-explanation (SE) is a learning activity in which the learner is explaining the learning material (e.g. a worked example or instructional text) to oneself, by making inferences from existing knowledge (Chi et al. 1989). SE results in deep learning, as it allows the learner to integrate new with existing knowledge, identify and eliminate misconceptions, and reflect on their knowledge (Chi 2000). SE has been shown to improve problem-solving skills when learning from worked examples (Chi et al. 1989), as well as to learn declarative knowledge from text (Chi et al., 1994). Research also shows that learning effectiveness is increased when students explain their own solutions when solving problems in intelligent tutoring systems (Eleven, Koedinger, and Cross 1999; Rau, Aleven, and Rummel, 2009; Rau, Aleven, and Nikol, 2015; Conati and VanLehn, 2000; McLaren et al. 2016; Mitrovic, 2005; Weerasinghe and Mitrovic 2006a; Najar, Mitrovic, and McLaren 2016). Furthermore, Najar, Mitrovic, and McLaren (2016) acquired evidence that self-explaining enhances conceptual knowledge.

Empirical studies show that brilliant students produce a lot of high-quality self-explanations, while novices do not generate enough self-explanations (Chi et al. 1989; Someren, Barnard, and Sandberg 1994; Ainsworth and Loizou 2003). Although many students do not spontaneously self-explain, most will do so when prompted (Chi 2000), and can learn to do it effectively (Bielaczyc, Pirolli, and Brown 1992). Additionally, asking students deep questions to encourage high quality self-explanations is one of seven recommended instructional strategies in the 2007 practice guide of the Institute for Educational Science (Pashler et al., 2007). Over the years, SE prompts have been proven useful in various domains:

- electrical circuits (Johnson and Mayer, 2010)
- probability calculations (Atkinson, Renkl, and Merrill 2003; Berthold, Eysink, and Renkl 2009)
- geometry (Aleven, Koedinger, and Cross 1999)
- algebra problem solving (Nathan, Mertz, and Ryan, 1994)
- solving simple mathematical equations (Rittle-Johnson, 2006; Matthews and Rittle-Johnson, 2009)
- fractions (Rau, Aleven, and Rummel, 2009; Rau, Aleven and Nikol, 2015)
- physics (Conati and VanLehn, 2000)
- chemistry (McLaren et al. 2016)
- biology (Butcher, 2006; Ainsworth and Loizou, 2003; Chi et al., 1994)
- medicine (Chamberland et al., 2011)
- SQL programming (Najar, Mitrovic, and McLaren 2016)
- database modelling (Weerasinghe and Mitrovic 2006a; Weerasinghe and Mitrovic 2006b; Weerasinghe, Mitrovic, and Martin 2007)
- data normalization (Mitrovic, 2005)

Self-explanation is often combined with less demanding tasks like worked examples (Najar, Mitrovic, and McLaren 2016; Atkinson, Renkl, and Merrill 2003; Berthold, Eysink, and Renkl 2009; Conati and VanLehn, 2000) or more challenging tasks like problem-solving (Nathan, Mertz, and Ryan, 1994; Rau, Aleven, and Rummel, 2009; Chamberland et al., 2011; Weerasinghe and Mitrovic 2006a; Weerasinghe and Mitrovic 2006b; Weerasinghe, Mitrovic, and Martin 2007; Mitrovic, 2005; Aleven, Koedinger, and Cross 1999). Chi et al. (1994) did something similar by introducing SE with problem solving and reading a passage. Other work involved more elaborate worked examples with enhancements on self-explanation. Berthold, Eysink, and Renkl (2009) presented worked examples with multi-representational solutions and combined them with SE prompts. Conati and VanLehn (2000) also combined worked examples with visual representations in a tutoring system for Physics. Their worked examples comprised with a problem description and visuals representing the situation in the problem (i.e. a drawing of a helicopter), and a diagram portraying the elements of the problem description within the x and y-axis. Conati and VanLehn also provided incremental aid for self-explanation by offering various levels of scaffolding. Atkinson, Renkl, and Merrill (2003) combined SE prompts with worked examples and evaluated them with fading of the prompts aimed to encourage learners to identify the principle used in each step of the worked example. Butcher (2006) and Ainsworth and Loizou (2003) used SE when learning by reading and learning with diagrams and text. On the other hand, there are few implementations where SE is combined with an exercise

that is neither challenging or far less involved like studying worked examples. For example, Johnson and Mayer (2010) combined SE with multiple choice questions in a game. Most importantly, Rittle-Johnson (2006) and Matthews and Rittle-Johnson (2009) offered a more targeted approach and offered SE prompts after filling in a blank line in mathematical equations for fifth grade children.

SE prompts were first introduced as open-ended questions which encourage learners to think without any set limitations. Wylie and Chi (2014) discuss the range of SE prompts that have emerged, arranged by increasing amount of support provided: open-ended, focused, scaffolded, resource-based and menu-based prompts. Focused SE prompts are variations of open-ended prompts, which provide more explicit instructions (i.e. compare and contrast). Scaffolded SE prompts provide explanations with missing keywords to be filled in by the learner. Resource-based SE prompts offer a resource library (such as a glossary) which students can refer to. Lastly, menu-based SE prompts are similar to resource-based self-explanation prompts, but selections are provided from a menu instead of a resource library. A study by Aleven et al. (2004) compared open-ended SE with resource-based SE, where students selected reasons for their actions from a glossary. The results revealed no significant differences with overall learning gains between open-ended group and resource-based group, showing that open-ended SE is not always the best. However, authors suggest that the benefits of open-ended SE possibly did not manifest since it took more time to self-explain with their own words than selecting an explanation.

Other studies also compared different forms of self-explanation in different domains and contexts, such as open-ended vs. scaffolded in worked examples (Berthold, Eysink, and Renkl 2009), focused vs. menu-based in problem solving (Kwon, Kumalasari, and Howland 2011), and open-ended vs. menu-based in a game-like environment (Johnson and Mayer 2010). Among these studies, I am most interested in the work by Johnson and Mayer, because of the puzzle game-like nature of PyKinetic. The instructional domain was electrical circuits. Johnson and Mayer first conducted an experiment comparing transfer tests scores of participants who used menu-based SE prompts vs. participants without any SE prompts. The results revealed that the SE group significantly outperformed the group without self-explanation. The second experiment compared transfer test performances of the menu-based SE group and no self-explanation group from the first experiment, with a new group using open-ended SE prompts. Results showed that learners who used open-ended SE had similar transfer test results to learners who did not do any self-explanation. Therefore, the authors found that menu-based SE were more effective than open-ended SE prompts. The authors explained the results by reflecting on the limited cognitive capacity of learners based on the Cognitive Load Theory (Sweller, Van Merrienboer, and Pass, 1998; Sweller, 2011). Within a game-like environment, menu-based SE prompts may have resulted in effective learning due to minimising extraneous cognitive load while fostering germane and intrinsic load. Johnson and Mayer (2010) concluded that menu-based SE prompts may be more suitable in complex environments compared to open-based prompts.

Lee, Ko, and Kwan (2013) implemented open-based SE prompts in a game. Lee et al. (2013) conducted a study with a programming game Gidget which uses a Python-like programming language. The study was conducted with non-programmers, where half of the participants were assigned to a control condition who played the game without in-game assessment, and the other half (experimental condition) had the identical system but with in-game assessment. The experimental group performed in-game assessments with open-based SE. Their findings reveal that the experimental group significantly increased their speed and were more engaged as they have voluntarily completed more levels compared to those in the control group. However, since learners in the control group did not have any in-game assessments, it is difficult to investigate the sole effectiveness of open-based prompts in their system. The authors suggest that the SE prompts allowed learners to reflect on their solutions. Therefore, perhaps using open-ended SE prompts in a game environment is still useful, but not as effective as menu-based SE prompts. Furthermore, menu-based SE prompts were found more effective than open-ended prompts in several other studies (Gadgil, Nokes-Malach, and Chi, 2012; van der Meij and de Jong, 2011).

Wylie and Chi (2014) advise that regardless of the form, all types of SE prompts may lead to deeper learning. Further insights into the effectiveness of SE prompts can be obtained within the ICAP (Interactive, Constructive, Active and Passive) framework, which focuses on learners' engagement (Chi and Wylie 2014). According to the ICAP framework, learning increases as engagement increases. When referring to the ICAP framework, self-explanation is constructive in nature. However, as mentioned previously from work by Aleven et al. (2004), open-ended SE prompts, which may be considered more constructive than other types, were not found to be more effective than menu-based SE prompts. Therefore, various aspects need to be considered when choosing the type of SE prompts to use. For example, an important consideration in designing SE prompts, regardless of their form, is providing appropriate feedback. Learners, especially novices, have misconceptions, and require guidance via feedback. If feedback is not properly given, learners continue to operate using their false understanding which could be detrimental to their learning. However, an intricate balance needs to be achieved to help the learners just enough to facilitate in enhancing their skills rather than just spoon-feeding information.

2.6 Mobile Learning

The focus of this research is to design a smartphone tutor for Python programming. Mobile learning does not only involve learning via a mobile device such as a smartphone or a tablet. Rather, mobile learning is defined as learning that transpires in an undetermined setting or learning through mobile technologies (O'Malley et al., 2005). Park (2011) devised a pedagogical framework for mobile learning activities and classified them into four types: 1) high transactional distance and socialized (HS); 2) high transactional

distance and individualized (HI); 3) low transactional distance and socialized (LS) and 4) low transactional distance and individualized (LI).

The first type (HS) is when the pedagogical strategies are mostly driven by mobile learning applications and when students are heavily participating with peers by communicating to learn together. The second type (HI) is similar to HS; the learning is directed by the mobile application, but students in this type learn individually instead of collaborating with peers. The third type (LS) is where a mobile application provides a lenient structure for learning, but students work together with their peers and teachers. Lastly, LI also provides a lenient structure like LS but students in this type learn independently. A future goal for PyKinetic is aimed at type LI: mobile learning activities where students can independently learn Python anytime and anywhere. However, the versions of PyKinetic presented in this thesis can be classified as HI, as it poses a higher transactional distance (i.e. low level of learner autonomy), with a fixed order of learning activities for evaluation purposes.

Apart from one-on-one teaching like in PyKinetic, mobile learning systems are also being utilised to support diverse learning situations (Oyelere et al., 2018). Mobile applications are implemented to support an assortment of pedagogical strategies such as the following:

- inquiry learning (Shih, Chuang, and Hwang, 2010; Jones, Scanlon, and Clough, 2013; Nouri et al., 2014; Sun and Looi, 2017)
- flipped classrooms (Wang, 2016; Grandl et al., 2018)
- game-based learning (Klopfer et al., 2012; Perry and Klopfer, 2014; Su and Cheng, 2015; Vinay, Vaseekharan, and Mohamedally, 2013; Browne and Anand, 2013)
- cooperative learning (Roschelle et al., 2010)
- collaborative learning (Wong, Looi, and Boticki, 2017)
- competition-based learning (Hwang and Chang, 2016)
- blended classroom learning (Wang et al., 2009)
- exploratory learning (Liu et al., 2012)
- context-aware learning (Sun, Chang, and Chen, 2015)

There are also mobile applications developed as a learning management system (LMS) (Wen and Zhang, 2015; Oyelere et al., 2018), and as a massive open online course (MOOC) (Grandl et al., 2018). Moreover, the number of applications that support learning is growing rapidly in various instructional domains:

- medicine (Vinay and Vishal, 2013; Gavali et al., 2017)
- science (Liu et al., 2012; Klopfer et al., 2012; Jones, Scanlon and Clough, 2013; Nouri et al., 2014; Perry and Klopfer, 2014; Su and Cheng 2015; Sun and Looi, 2017)

- social science (Shih, Chuang, and Hwang, 2010; Hwang and Chang 2016)
- language learning (Wang et al., 2009; Kim and Kwon, 2012; Nakaya and Murota, 2013; Sun, Chang, and Chen, 2015; Wang, 2016; Wong, Looi, and Boticki, 2017)
- mathematics (Roschelle et al., 2010; Wen and Zhang, 2015)
- computing education (Hürst, Lauer, and Nold, 2007; Karavirta, Helminen, and Ihantola, 2012; Boticki et al., 2013; Vinay, Vaseekharan and Mohamedally, 2013; Wen and Zhang, 2015; Mbogo, Blake, and Suleman, 2016; Grandl et al., 2018; Oyelere et al., 2018).

Furthermore, mobile learning is proved to be effective for learning and motivating learners from across diverse age groups:

- children and pre-teens aged between 7-14 (Roschelle et al., 2010; Shih, Chuang and Hwang, 2010; Liu et al., 2012; Klopfer et al., 2012; Vinay, Vaseekharan and Mohamedally, 2013; Nouri et al., 2014; Su and Cheng 2015; Hwang and Chang, 2016; Wong, Looi, and Boticki, 2017; Sun and Looi, 2017; Grandl et al., 2018)
- teenage students ages 14-16 (Perry and Klopfer, 2014; Wang, 2016)
- undergraduate and graduate students aged 16-35 (Hürst, Lauer, and Nold, 2007; Boticki et al., 2013; Browne and Anand, 2013; Nakaya and Murota, 2013; Mbogo, Blake, and Suleman, 2016; Sun, Chang, and Chen, 2015; Oyelere et al., 2018)
- university students and teachers (Wen and Zhang, 2015)
- working professionals (Wang et al., 2009)

There are many mobile applications that support various aspects of computing education such as the following: supporting entire courses through an LMS (Oyelere et al., 2018; Wen and Zhang, 2015), learning algorithm executions through visualisations (Boticki et al. 2013; Hürst, Lauer, and Nold, 2007), control-flow learning (Karavirta, Helminen, and Ihantola, 2012; Vinay, Vaseekharan & Mohamedally, 2013; Fabic, Mitrovic, and Neshatian, 2017c; Grandl et al., 2018) and code writing (Mbogo, Blake, and Suleman, 2016). Oyelere et al. (2018) implemented MobileEdu, a LMS for third year students in a systems analysis and design course. The goal of MobileEdu was to alleviate poor engagement of students and to provide support for lecturers in managing the students. Oyelere and colleagues conducted a study where they compared students' learning with traditional lectures (control) to students learning almost entirely through MobileEdu (experimental). MobileEdu had communication features which allowed students in the experimental group to interact with their instructor despite not having face-to-face lectures. Results revealed that students who learned via MobileEdu learned significantly more than the control group. They also conducted a survey which revealed that the experimental group had significantly improved perception towards the course

compared to the control group. Wen and Zhang (2015) also presented a LMS referred to as Micro-Lecture Mobile Learning System (MMLS). The system contained 12 courses. The results showed that participants who used the MMLS in Data Mining, Digital Signal Processing, and MATLAB courses showed significantly higher final exam marks than previous course intakes before introducing the MMLS.

Boticki and colleagues (2013) also conducted a semester-long study. The authors introduced SortKo, an Android application for teaching sorting algorithms with the help of visualisations. The authors found that SortKo learners learned 30% more than those who did not use SortKo, and their survey results verified that it helped motivate the students. Hürst, Lauer, and Nold (2007) also introduced animations to help with understanding executions of algorithms. However, the study by Hürst and colleagues was more concerned on HCI implications. They compared differences in learning between devices (laptop vs. iPod with video capability) and modalities (audio vs. no audio).

There are also several implementations in control flow learning with different approaches. For example, Vinay and colleagues (2013) and Grandl et al. (2018) both used gamification approaches. However, Grandl and colleagues had a flipped classroom design, where learners create various games via code like Scratch. Moreover, other implementations used Parsons problems (Parsons and Haden, 2006) where syntactically correct code is given but needs to be reordered in the right order to match the given expected output (Karavirta, Helminen, and Ihantola, 2012; Fabic, Mitrovic, and Neshatian, 2017c). The study by Karavirta et al. (2012) was focused on supporting automated feedback for Parsons problems within their system MobileParsons.

Lastly, Mbogo, Blake & Suleman, (2016) conducted a 2-hour long qualitative study with a mobile learning system, which provided code writing exercises in Java by designing static scaffolding. The learners were able to collapse parts of the code and toggle between collapsed and full versions of their programs to support Java code writing on a smartphone. Participants remarked that the scaffolding for code segmentations made the application easier to use.

There are very few mobile learning tutors in the literature specifically for enhancing programming skills. My work is similar to work by Karavirta et al. (2012) and Mbogo et al. (2016), with the notion of providing programming exercises to support learning. However, instead of only providing one type of activity as in those two projects, I provide various learning activities to target several coding skills. One of the pedagogical strategies I applied in PyKinetic is an implementation with a component-skills perspective (McArthur et al., 1988). I aim to fill the gap and provide an opportunity for learners to continue practicing their programming skills even when not in a lecture setting. Furthermore, I aim to provide learners an avenue to enhance several coding skills while they are away from a personal computer by supplying a combination of various coding activities.

2.7 Parsons Problems

Parsons problems or Parsons puzzles were originally proposed as a fun way for introductory learners of Turbo Pascal to improve their problem-solving skills. Parsons problems are aimed at students learning programming for the first time (Parsons and Haden 2006). These problems are suitable for novices as they contain correct syntactic constructs and impose low cognitive load (Morrison et al. 2016). Furthermore, learners often lack mental models and have difficulty in translating a problem into manageable tasks (Winslow, 1996). Parsons problems provide scaffolding helpful for novices, unlike in traditional code-writing exercises, where a student is expected to construct code where the only scaffolding is the problem description. Moreover, one of the common logical programming bugs classified by Pea (1986) is parallelism, where an error is caused because a learner thinks that multiple LOCs are executed at the same time. Thus, the learner is completely unaware of the significance of the sequence of the code. Therefore, Parsons problems may be good practice for novices to grasp the skill of understanding the sequential nature of a program.

Parsons problems usually contain a problem area and a solution area. The problem area contains blocks of code, where some blocks may contain more than one line of code. Parsons problems implemented on personal computers and mobile devices are solved by dragging blocks of code from the problem area onto the solution area, where these two areas are usually beside each other (Parsons and Haden 2006; Ericson, Margulieux, and Rick 2017; Ihantola and Karavirta 2011; Karavirta, Helminen, and Ihantola 2012; Ihantola, Helminen, and Karavirta 2013; Harms, Chen, and Kelleher 2016; Kumar, 2018; Hosseini, 2018). However, most often Parsons problems are implemented for personal computers, where screen space is not an issue (Parsons and Haden 2006; Garner, 2007; Ihantola and Karavirta 2011; Harms, Chen, and Kelleher 2016; Ericson, Margulieux and Rick 2017; Kumar, 2018; Hosseini, 2018; Bari et al., 2019). Although smartphones have limited screen estate, Parsons problems are compatible with a smartphone interface because the drag and drop motion used in solving these problems is commonly used when interacting with smartphones. Moreover, Parsons problems can be implemented in smartphones without having to sacrifice too much screen space as they do not require typing. Requiring the student to write code on a smartphone might not be ideal, as half of the screen will be obstructed by the keyboard when typing. Karavirta et al. (2012) also perceived Parsons problems to be suitable for mobile device interface and have developed MobileParsons for Android and iOS mobile devices. Despite restrictions on screen size, MobileParsons implemented Parsons problems on mobile devices with two work spaces; dragging and dropping blocks of code side by side when on landscape mode, and bottom to top when on portrait mode (Karavirta, Helminen and Ihantola 2012. MobileParsons was further developed to allow limited editing of lines (Ihantola, Helminen and Karavirta 2013).

There are different variants to Parsons problems. Denny et al. (2008) considered five initial variants of Parsons. The first two variants contain the exact lines of code that can be used to form a correct code structure. The first variant 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. The next two variants contained *distractors* which are extra incorrect LOCs. These variants were available with and without scaffolding. Both variants are composed of paired options for each line of code given in a randomised order, but the paired options are clearly placed right next to each other. These variants were available with and without scaffolding. The last variant contains pairs of options for each line of code, but instead of being placed next to each other, these pairs are provided in a random 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, 7 lines of code for the puzzle becomes 14 and having these 14 lines of code in a completely randomised order may be viewed by students to be overwhelming to attempt. Instead of having the same number of distractors and correct LOCs in a problem, other researchers have evaluated Parsons problems usually with only a few distractors (Parsons and Haden 2006; Garner, 2007; Denny et al., 2008; Harms, Chen, and Kelleher 2016; Ericson, Margulieux and Rick 2017; Ihantola and Karavirta 2011; Karavirta, Helminen and Ihantola 2012; Kumar, 2018). Moreover, Ihantola and Karavirta (2011) introduced two-dimensional Parsons problems which is a variation which requires the learner to also specify the correct indentations of the code to indicate the start and end of code structures in Python, aside from rearranging blocks of code in the correct order. Another variation of Parsons problems contains incomplete LOCs which require the learner to provide missing elements (Ihantola, Helminen, and Karavirta 2013; Fabic, Mitrovic, and Neshatian, 2017c).

There is no widespread agreement on how Parsons problems compare to other types of exercises typically used in introductory programming courses, such as code reading, tracing, writing and explaining. However, there have been adequate evidence that Parsons problems is pedagogically similar to code writing (Denny et al., 2008; Ericson, Margulieux, and Rick, 2017; Ericson, Foley, and Rick, 2018). Denny et al. (2008) explored the potential of Parsons problems as exam questions and found a weak correlation between scores on Parsons problems with code tracing questions, and a moderate positive correlation with code writing. Therefore, Denny and colleagues suggested that Parsons problems were similar to code writing. Some researchers find the complexity of Parsons problems to be in between code tracing and code writing (Lister et al. 2010), while others believe Parsons problems to be easier than code tracing (Lopez et al. 2008).

A recent study was conducted using Parsons problems as an assessment activity, as they pose a lower cognitive load compared to traditional code writing (Morrison et al. 2016). The authors propose that this is because learners are only required to focus on sequencing the code, since the correct syntactic constructs are provided. However, this may not be always true, as Parsons problems may require higher cognitive load

depending on the complexity, type, and interface used (on paper or on a device). Moreover, others also perceive that the position of Parsons problems in the hierarchy of programming skills depend on complexity and type of Parsons problems (Ihantola and Karavirta 2011). More factors that could be considered are scaffolding and feedback provided. Recent work by Ericson et al. (2017) compared learning gains between three groups of students solving the same set of Python problems, but with each group using a different method of problem solving. One group was solving two-dimensional Parsons problems with distractors, another was tasked to fix the broken code, and the third group wrote the same code without scaffolding. Two-dimensional Parsons problems required learners to specify the correct indentations for each LOC in addition to re-ordering the LOCs (Ihantola and Karavirta 2011). All three groups showed evidence of learning from pre- to post-test on fixing and writing code, which is consistent with the positive correlation between Parsons problems and code writing found by Denny et al. (2008). Furthermore, Ericson and colleagues (2017) found neither a significant difference between the three groups on their learning performance, nor a significant difference on retention of knowledge when evaluated a week after the study. A notable finding was that the group who solved two-dimensional Parsons problems with distractors were significantly faster than the other groups who solved the same set of problems but used a different method of problem solving. Therefore, the authors found the Parsons problems group to be most efficient as they gained the same knowledge significantly faster than the other two groups.

A more recent study (Ericson, Foley, and Rick, 2018) compared the effectiveness of learning with adaptive and non-adaptive two-dimensional Parsons problems with distractors in comparison to learning by code writing exercises. The study was conducted with introductory Python programming undergraduate students. Participants were randomly assigned to four groups which had pairs of worked examples and an additional exercise. The additional exercises for the groups were: non-adaptive two-dimensional Parsons problems with distractors, adaptive two-dimensional Parsons problems with distractors, code writing exercises with the same content as the Parsons problems, and a control group who solved adaptive two-dimensional Parsons problems with distractors unrelated to the pre-test questions. The adaptive Parsons problems provided intra-problem and inter-problem adaptation. When a learner is having a difficulty in solving the problems, intra-problem adaptation provided access to a “Help Me” button. Pressing the button allowed learners to the following: elimination of distractors, demonstration of correct indentations, or combining blocks. Furthermore, based on the performance of the learner on the previous problem he/she solved, inter-problem adaptation altered the difficulty of the next problem given to the learner. Results reveal that students who practiced with Parsons problems (adaptive and non-adaptive) solved the same problems significantly faster than those who solved code writing exercises. Moreover, students from the three groups who solved problems on task significantly improved their composite scores from pre- to post-test. But, when the three groups were compared, there was no significant difference between their learning

gains. Therefore, the authors confirmed that solving non-adaptive and adaptive two-dimensional Parsons problems with distractors is more efficient than practicing via code writing exercises. Results also revealed that adaptive Parsons group proved to be most effective compared to the other groups revealing a medium Cohen's d effect size, while there was only a small effect size for the non-adaptive Parsons group.

There are limited results in literature about problem-solving strategies used by novices and experts for Parsons problems. Ihantola and Karavirta (2011) report on a small study involving four experts. Senior teachers and teaching assistants participated in the study. The study was conducted in JSParsons, a Web environment for Parsons problems built using the js-parsons library. The study presented ten Parsons problems with distractors, which required indentations to be specified. For each problem, the name of the algorithm was provided (such as insertion sort). The participants were asked to solve ten Parsons problems on simple tasks and algorithms on tree traversal and sorting. The study followed the think-aloud protocol partially, requiring the participants to verbalise their thoughts while solving the puzzles but the authors decided not to do either audio or video recording to keep the discussion open. Suggestions were asked after each session which took 20-30 minutes per participant. The results were interesting in that they have found that even the experts followed a different strategy in solving Parsons problems. This strategy which seems like a "divide and conquer" strategy rather than linearly going through each line of code, is similar to what I observed with participants in my pilot study described in Chapter 4.9. The strategy was to start with the function signature and find the loop statements and the conditional statements. Other findings in their study also revealed similar observations on my pilot study, which is that participants rarely requested feedback and only did so when they felt that their solution was ready to be submitted. The authors also faced similar issues found in my pilot study where there could be several solutions to a problem, but some solutions could be functionally correct but still logically or stylistically incorrect. The authors propose that several solutions can be accepted, but the authors of the instructional material must be alerted whether alternative solutions should be accepted.

Another study conducted with novices (Helminen et al., 2012) found that students followed a top-to-bottom strategy for simple Parsons problems. In two problems, the first step in 98.5-99.3% of the solutions was to position the function signature. A recent paper (Kumar, 2019) investigated strategies used by novices in Epplets (Kumar, 2018), a tool for solving Parsons problems in Java, C#, and C++. The novices were introductory programming students, who used Epplets for five semesters in Java or C++. Kumar (2019), specifically focused on the strategies in doing if-else statements. Kumar identified problem-solving strategies by formulating these three steps: retrieve the student's solution sequence, utilise Backus-Naur Form (BNF) grammar to provide a representation of ideal strategies, and thirdly implement a "best-match parser" to measure the solution of the student against the BNF grammar. Results reveal that C++ students had a significantly higher "best-match" score than Java students, implying that C++ students employed

better strategies than Java students. C++ students used strategies that are more similar to optimal problem-solving strategies as defined in the BNF grammar compared to Java students.

I investigated problem-solving strategies used by novices compared to experts when solving Parsons problems with and without distractors in my pilot study (Chapter 4). I also implemented another variant of Parsons problems (with incomplete LOCs) and evaluated its effectiveness with menu-based SE prompts (Chapters 5, 7).

2.8 Educational Systems for Python

Most educational systems for learning introductory Python programming are implemented on personal computers. Some systems contain practice exercises and a lesson plan (Barria-Pineda et al., 2017; Lokar and Pretnar, 2015; Quinson and Oster, 2015). One system, (Zingaro et al., 2013) Python Classroom Response System (PCRS) can be used by teachers to facilitate multiple choice and code writing questions. The system is built on Online Python Tutor (Guo, 2013) which is a web-based visualisation tool also for Python programming. Since the PCRS is built on the Online Python Tutor, students are allowed to step through the code and provided with a visualisation of the memory model.

In Google Play for Android as of 22 May 2019, the keyword "Python" resulted in 250 applications. The results were a mixture of applications: two applications about pythons (snake), Python programming, and some could not be verified as they were not in English. For the applications about Python programming the results were repositories for Python documentation, Python interpreters, and IDEs for Python. Others were designed for education, but mostly offered learning by examples, through a combination of text descriptions, code, images and videos. A few applications had interactive exercises for Python programming by offering multiple choice quizzes. An example is Quiz&Learn Python (Karavirta, 2019), which is a game to test and improve knowledge on Python 2.x programming available on Android and iOS devices. The aim of the game is to answer 20 multi-choice questions and to answer them correctly within one minute for each question as fast as possible to gain more points. There are four help options that can be used only once each for every game: remove two incorrect answers, skip a question, debug the code and stop the timer. Remove incorrect answers removes two incorrect choices out of the given choices. Choosing to skip a question, skips the current question to move on to the next without answering it. Choosing the help option to debug the code gives the users an access to a debugger which shows a line by line visualisation of the execution of the given code snippet on the question. The last option: stop the timer, gives the user an unlimited time to answer a question. The game ends when the user answers a question incorrectly, has ran out of time, or has successfully answered all 20 questions. The application is developed using the following technologies: Apache Cordova, Zepto.js, Topcoat, SASS, Node.js, PostgreSQL

(Karavirta, 2019). Based on observations while using the application, it seems that it is presented more as a game rather than a tutor with game elements. It seems to be more focused on gaming features rather than providing pedagogical aspects to support learning.

From the search results on Google Play, nine applications provided a wider array of interactive exercises. Six of these applications offered partially written code writing exercises where the learner is required to fill in the missing elements to complete the code (SoloLearn, 2019; DataCamp, 2019; Enki.com, 2019; Py, 2019; Mimohello GmbH, 2019; Learn Programming, Programming App, Coding App Education, 2019). Three applications (ApkZube, 2019; Coding and Programming App, 2019; Learn Programming, Programming App, Coding App Education, 2019) were coupled with a Python compiler to verify the code of the students for code writing. However, it was not clear whether learners were prompted to answer code writing questions, or the compilers were just used so that the learners can inspect and experiment with the programs in the applications. Two applications (SoloLearn, 2019; Py, 2019) provided regular Parsons problems where single LOCs are required to be rearranged in the correct order match the problem description and expected output with the indentations provided as scaffolding. One application (Py, 2019) also provided output prediction exercises. On the other hand, Mimo: Learn to Code by Mimohello GmbH (2019) provided exercises with erroneous code requiring learners to identify the error/s. Codemurai-Learn Programming by Zenva Pty Ltd (2019), offered an interesting exercise where learners are required to drag and drop elements of a single LOC in the correct order. This appears to be similar to Parsons problems except it delivers a more targeted approach because instead of rearranging single LOCs or blocks of code to form a program, elements of a single LOC are reordered to form one line of code. They gave an example for Javascript where learners are instructed to declare a variable isZombie with a value true: and the blocks given are `true`, `var`, `;`, `isZombie`, `=`.

Some of the applications are also available for other programming languages. DataCamp (2019) also supports R and SQL. Py (2019) is also available for Swift, Javascript, HTML, and CSS. The application by Enki.com (2019) is also available to learn other programming languages such as SQL, JavaScript, Blockchain, CSS, HTML and Java. Zenva Pty Ltd (2019), is also available for learning other languages and HTML, CSS, Javascript, TypeScript, Java, Swift and C#. Mimohello GmbH (2019) is also implemented for learning HTML, CSS, Javascript, Java, Swift, C++, SQL and PHP. Lastly, Programming Hub: Learn to Code by Coding and Programming App (2019) is available for learning a wide array of programming languages such as Java, C, C++, HTML, Javascript, R, CSS, VB.net, C#, Linux Shell Scripting, Swift, SQL, Jquery, and Assembly 8086.

Two applications (SoloLearn, 2019; Enki.com, 2019) provided additional features to allow collaboration where learners could interact with each other through discussions. All nine applications allow learners to keep track of their progress, but the level of detail provided differs. Five of these applications

were also designed with game elements where points can be earned to level up and earn badges (SoloLearn, 2019; DataCamp, 2019; Mimohello GmbH, 2019; Py, 2019; Zenva Pty Ltd, 2019). Learn Python by SoloLearn (2019) combined gamification and collaboration by also allowing students to compete with each other on the leader board and can be awarded with a certificate of completion. Programming Hero: Coding App Just Got Fun (beta) by Learn Programming, Programming App, Coding App Education (2019) is unique as the application is designed with a story line. For example, one of the first lessons covered print statements to display output. The application prompts learners to print a planet. When learners are successful in writing the code, the application displays a visual of a planet. In reality, Python would only display a string "planet" rather than a visual representation of a planet. However, this is probably for a younger audience to motivate them in learning programming.

There are other educational environments for Parsons problems. *js-parsons* (Ihantola and Karavirta, 2011) is a JavaScript library intended for developing Parsons problems. The library supports "two dimensional" Parsons problems which allows students to drag lines of code (LOCs) from a set on the left-hand side of the screen and drop it on the solution space on the right-hand side of the screen. The second-dimension feature is that indentations are supported, and students can change the indentations of the LOCs in their solutions. The library is language independent, therefore it can be used to develop Parsons problems for "any" programming language. Since *js-parsons* is a JavaScript library, it can be used to develop Parsons problems on webpages designed for personal computers or mobile webpages for tablets and smartphone devices. *js-parsons* provides additional support for developing Parsons problems in Python. *js-parsons* is also used in Runestone Interactive (2019) which offers interactive Python books to teach Python and other Computer Science concepts, which contains various coding exercises including Parsons problems.

MobileParsons (Karavirta, Helminen, and Ihantola, 2012) which was briefly mentioned in the previous section is developed using *js-parsons* library. The application is implemented using web technologies such as specifically JavaScript libraries: jQuery and jQueryUI. Automatic feedback is provided for solutions which was implemented based on longest common subsequences (LCS). The LCS is searched within the student solution and compared to the model solution for feedback. Feedback abuse was prevented by deactivating the feedback button if the frequency goes above a certain threshold. MobileParsons was not specified as an ITS by the authors. It seemed like MobileParsons was only implemented as a mobile version of the existing desktop version utilising the *js-parsons* library. An extension of MobileParsons was made in 2013 (Ihantola, Helminen, and Karavirta, 2013). The extension introduced a new type of exercise to the tutor which included a Parsons problem variant that had LOCs with missing keywords that the learners can toggle through by tapping. The authors also presented design guidelines based on their own research and experience on presenting Parsons problems on a mobile platform effectively. The guidelines were useful for the variants of Parsons problems I developed in my research.

2.9 Conclusions

It is generally agreed upon by researchers that programming is a difficult skill to learn. Furthermore, debugging, which is an essential part of implementing programs, requires higher level of skill than programming (Ahmadzadeh, Elliman, and Higgins, 2005; Fitzgerald et al., 2008). Normally, debugging is not formally taught in programming courses. But as found by Chmiel and Loui (2004), students benefit from formal education on debugging. But are our traditional methods of teaching enough for the new generation of students? Klopfer et al. (2009) suggest that educators should make use of the new technologies to better support the new generation of programming students. Although it was debated 11 years ago (Bennett, Maton, Kervin, 2008) that there is no profound evidence for a widespread shift in teaching programming, it was found that new programming learners prefer other learning methods i.e. by watching YouTube videos (Piteira and Costa, 2013). (Bennett, Maton, Kervin, 2008) suggest involving the opinions of new students and teachers.

It is widely accepted by researchers and even students that programming is learned by doing (Shaffer, 2005; Tan, Ting, and Ling, 2009; Piteira and Costa, 2013). Furthermore, there are a variety of skills needed for programming such as code tracing and code writing skills. I endeavour to develop PyKinetic to support a variety of skills by designing multiple tasks of different nature and utilising them to produce an effective learning experience. I also combine one of the programming exercises in PyKinetic with SE to facilitate deeper learning. SE is proven effective when combined with various domains and is normally combined with either a problem-solving task or a worked example. I combined SE with Parsons problems, a task that is cognitively in between problem-solving and worked examples similar to work by (Rittle-Johnson, 2006; Matthews and Rittle-Johnson, 2009).

To design effective learning activities and ultimately to develop an effective tutor, I presented literature on CLT and MLT to get a better understanding on how humans learn. Humans have limited cognitive capacity and based on the CL theory (Sweller, Van Merrienboer, and Pass, 1998; Sweller, Ayres, and Kalyuga, 2011), learning is challenging due to the complexity of learning new material because a learner is unfamiliar with the individual elements involved and how they interact with each other. To design an effective tutor, it is vital to leave enough room for learners to process intrinsic cognitive load and germane cognitive load, which are both essential for learning. Giving too much context and displaying unnecessary visuals may add to extraneous cognitive load and overload the working memory of learners. Furthermore, I also took into consideration the Multimedia Learning principles (Mayer, 2009) when developing PyKinetic. Even though the MLT theory is devised for personal computers, the principles are still applicable to designing on a smaller device such as a smartphone.

Mobile learning is proven effective in many domains including computing education (Hürst, Lauer,

and Nold, 2007; Karavirta, Helminen, and Ihantola, 2012; Boticki et al., 2013; Vinay, Vaseekharan and Mohamedally, 2013; Wen and Zhang, 2015; Mbogo, Blake, and Suleman, 2016; Grandl et al., 2018; Oyelere et al., 2018). A search on Google Play for Android applications revealed some commercial applications that support learning computer science education including supporting coding skills through interactive tasks (SoloLearn, 2019; DataCamp, 2019; Enki.com, 2019; Py, 2019; Mimohello GmbH, 2019; Learn Programming, Programming App, Coding App Education, 2019; ApkZube, 2019; Coding and Programming App, 2019; Zenva Pty Ltd, 2019). However, there are not many evaluations covered in the literature focusing on enhancing programming skills through control-flow learning and code writing using a mobile device (Karavirta, Helminen, and Ihantola, 2012; Vinay, Vaseekharan and Mohamedally, 2013; Fabic, Mitrovic, and Neshatian, 2017c; Grandl et al., 2018; Mbogo, Blake, and Suleman, 2016). Therefore, there are opportunities to contribute in the knowledge of designing a mobile tutor focused on enhancing coding skills.

In the next chapter, I present my own frameworks that I have devised based on the literature I presented in this chapter. I present key principles that I have followed for effectively executing various aspects of the research project. Moreover, I present frameworks specifically for designing the activities, authoring the content of the tutor, and the evaluation material used for the studies. I also present early prototypes for activities in PyKinetic.

Parsons problems are designed for novices and therefore known to be low cognitive load exercises (Parsons and Haden 2006; Morrison et al. 2016). Furthermore, there is sufficient evidence that they are similar to traditional code writing exercises (Denny et al., 2008; Ericson, Margulieux, and Rick, 2017; Ericson, Foley, and Rick, 2018). Also, due to its drag and drop controls, they are suitable for the touch screen interfaces of smartphones (Karavirta et al., 2012). In Chapter 4, I present my pilot study using Parsons problems with and without distractors. I also investigated the strategies used by novices and experts in this study. In Chapter 5, I evaluated another version of PyKinetic offering a different variant of Parsons problems containing incomplete LOCs. In Chapter 6, I evaluated a version of PyKinetic with output prediction and debugging exercises. Lastly, in Chapter 7 I present my last evaluation study with two versions of PyKinetic where both versions offer regular Parsons problems, Parsons problems with incomplete LOCs and menu-based SE prompts, output prediction, and debugging exercises. One version presented in Chapter 7 is enhanced with adaptive problem selection where a different type of problem is given to the participant based on his/her performance on the previous task.

3 PLANNING, DESIGN, AND PAPER PROTOTYPES

This chapter presents the initial planning and design phase which occurred before implementation started. The set of guidelines that I designed for my PhD research is also covered in this chapter. (R1) (How does one create a framework for designing effective learning activities in a smartphone programming tutor?) is addressed in this chapter by proposing a set of frameworks for designing pedagogical activities for learning programming in a mobile tutor (Section 3.1). The guidelines were designed from own ideas, experience in developing mobile applications and teaching Python programming, and suggestions from my supervisors. Furthermore, the guidelines were guided by literature presented in Chapter 2. I followed the guidelines presented in this chapter in all versions of PyKinetic and investigated the effectiveness of the guidelines in my evaluation studies.

The approach taken in developing the resarch was more of an agile (Appendix P.1) approach rather than a traditional waterfall model (Appendix P.2). Each evaluation study was treated like a sprint used in an agile approach. Overall goals and research questions were first established at the start of the research. However, in between evaluations more research questions were explored following the agile approach. The planning and design were an ongoing process, adapting from results of my evaluations and literature at the time. The following section covers the set of guidelines I designed and used for all aspects of the research project. After that, I present my prototype activities, followed by my plans for implementation and evaluations. Lastly, I present a technical overview of PyKinetic.

3.1 Guidelines

In this section I present the frameworks that I followed throughout the research project. I start by presenting the five important factors that I adhered to in developing various aspects of the research. Next, I present my user interface design guidelines that I followed for designing activities. Then in Section 3.1.3, I discuss my guidelines in authoring the content of the activities. All activities implemented in PyKinetic were carefully designed adhering to all guidelines discussed in this section. I did not follow the guidelines independent from each other, rather my programming activities were developed while adhering to all guidelines in a cohesive manner. My intention was to follow all guidelines in coordination towards my goal to develop effective learning activities. Finally, I present my guidelines when authoring evaluation material in Section 3.1.4.

3.1.1 Important Factors

The planning and design of the tutor were defined by my overall resesarch goals, research questions, and influenced by literature. Several activities were designed to support multiple coding skills, and at the same time remain suitable for a smartphone interface. Most of the activities were designed before any evaluations were conducted. However, since an agile process was followed, some activities were designed in later stages of the resesarch. Designs which emerged from the results of evaluations are described in the corresponding chapter of each evaluation study. In designing activities at any stage of the research, there were **five factors** that were taken into careful consideration to ensure that the activities adhere to the overall goals of my PhD resesarch:

(i) **Coding skills -** ensuring that the activities collectively cover multiple coding skills such as code writing, code tracing, code understanding, debugging.

(ii) **Content -** guaranteeing that content provided is relevant to the Introduction to Computer Programming (COSC121) material. COSC121 is a first-year course in University of Canterbury. The curriculum in COSC121 was used as my basis for the content in PyKinetic as most of my participants were enrolled in the course.

(iii) **Complexity -** providing complexity appropriate for the target audience (introductory programming students)

(iv) **Feedback -** providing enough scaffolding and feedback, without overdoing the attention on feedback, since feedback is not the focus of my resesarch. By "enough" meaning feedback must be carefully administered while considering the intended level of difficulty of the activity. For example, too much feedback may harm the projected complexity for the activity; similarly, not enough feedback may cause an activity to be more difficult than planned.

(v) **User-interface design -** the user-interface is not the primary focus of this research. However, this is an important aspect in designing activities because unnecessary complexity in the user-interface should be avoided to maximise the learning benefits.

3.1.2 User Interface Design Guidelines

The focus of this resesarch is not on UI design. However, providing a good user interface is crucial to maximise the learning experience. My main goal in designing the interface is to focus on simplicity and functionality, not ornamentation. To ensure that activities and the tutor as an entity are designed based on the goals and research questions of the resesarch project, the following design guidelines were established before development:

(i) Provide a light-weight and seamless interface

The interface should provide a smooth user-experience where students can feel that they have one combined learning experience. Although the activities are of different nature, learners should not be burdened in coping with distinctive designs for each activity. The design layout should be similar across activities and design elements and controls should be consistent. Learners should only be obligated to shoulder a light burden in familiarising with the interface, to focus on enhancing their programming skills rather than spending time on learning how to interact with the interface. Moreover, background processes must be concealed, such as minimising loading time and displaying a loading screen if necessary.

(ii) Consistent use and placement of design elements

Buttons and labels should be used in a coherent manner and positioned in the same areas in all activities. Moreover, feedback should be shown in similar regions of the interface across various activities. The focus in designing the elements of the interface is simplicity and functionality, not beautification. Therefore, there is no need to create design elements from null unless proved necessary; default elements and widgets from the Android library must be utilised as much as possible.

(iii) Consistent colour scheme and smart visual cues

Usage of visual cues and colour should be consistent and comprehensible. For example, the colour red usually has a negative connotation; so, it might be confusing to utilise the colour red to indicate a positive result. It may not always be possible to display all visual cues and feedback on the interface due to space limitations. Therefore, careful consideration should be applied when feedback can only be shown for a short amount of time.

(iv) Clever use of available space

The application interface is currently designed specifically for Android smart phone devices; therefore, the space should be utilised well while avoiding the layout to be too cluttered. Careful consideration should be applied to ensure that important parts of the learning activities are not obstructed. The application can still be functionally used on Android tablets, but the

overall design, proportions and sizes of icons and text may not look aesthetically pleasing compared to when using it on a smart phone device.

3.1.3 Guidelines in Authoring Content for the Activities

The focus of this resesarch is not purely on composing teaching material for programming. However, carefully authoring the content for PyKinetic is vital for the effectiveness of PyKinetic to support learning. For example, a problem may contain a syntactically and semantically correct code, but students may struggle to answer them not because of their lack of skills but due to the problem being too abstract (Parsons, Wood, and Haden, 2015). Therefore, to ensure a high standard of content in the tutor, the following guidelines were established for authoring the content in PyKinetic:

(i) Difficulty

The problem difficulty must be appropriate to enrich the abilities of introductory programming students. The content should pose the right amount of complexity for a stimulating challenge that would engage the students but would not overwhelm them. However, not all the content should be challenging. There should be a sufficient range of easy to difficult material to best cater for the spectrum of abilities of the target audience. A problem is generally considered more difficult if it requires the learner to perform more than one activity. For example, finding erroneous LOCs in the code and predicting output. Moreover, the problem difficulty is measured by a combination of the following factors: number of LOCs, number of topics covered, and levels of indentation.

(ii) Feedback

The wording of the feedback must be taken into careful considernuation. If applicable, the jargon used in the other parts of the task (i.e. in the code and problem description) must be similar to terminology used in the feedback to avoid confusion.

(iii) Learning objectives

The material must be intricately designed to target certain learning objectives and/or address common misconceptions in Python programming, instead of just creating random tasks.

(iv) Ordering

When applicable to an experimental design, the complexity of the material should be gradually increasing; while also taking into consideration the user interface design guidelines in Section 3.1.2. For instance, an activity x with a more complex design but covers an easy content is more likely to pose as more difficult than activity y with a simpler design but is comprised with a more challenging material.

(v) Presentation of text

Words must be carefully chosen based on how the information will be presented. For example, some content such as feedback are only shown for a limited time i.e. five seconds, whereas others are a permanent fixture.

(vi) Characteristics of smartphones

Since PyKinetic is a smartphone tutor, there is limited screen estate. Therefore, it must be ensured that the content still poses the right amount of difficulty while still providing concise and clear information that is compatible with a smartphone tutor.

(vii) Technological limitations

The content must be malleable enough to be adjusted for unexpected technical difficulties, or unforeseen time constraints in implementation. Some content may be too complex to use in an activity. For example, when creating Parsons problems, the correct position of some lines of code (LOCs) is dependent on another LOC. Like when a variable must be initialised before being used by another statement. Therefore, the solutions in the database should also store those dependencies. It may take too much development time to also store dependencies. So, the content should be adaptable enough for alterations.

3.1.4 Guidelines for Authoring Evaluation Materials

(i) **For Consent forms and Information sheets –** adhere to the human ethics committee standards and structure of these documents. Also, too much text must be avoided as most likely participants would not read it all.

(ii) **For Instructional material for PyKinetic –** provide appropriate number of screenshots for the participants and make use of and other visual cues like colours, arrows and labels. However, graphics should only be used for the purposes of providing clarity; additional visuals for aesthetics should not be added. In addition, putting too much unnecessary information must be avoided; the goal is to provide clear and concise instructions.

(iii) **For Pre- and post-tests –** Both tests must have comparable difficulty and contain similar learning material. The tests should also be of comparable difficulty and comprise with similar activities that are offered in PyKinetic for that particular evaluation study.

3.2 Prototype Activities

There were several activities considered in the early phases of the resesarch. However, not all activities were implemented in the tutor. The activities considered in the planning stage included “tap and trace”, variations of Parsons problems, identifying erroneous LOCs, fixing erroneous LOCs, and output prediction.

3.2.1 Tap and Trace

The “tap and trace” activity is designed to target code tracing skills. The tutor presents the learner with a code snippet and its expected output. The task is to tap on each of the LOC based on the order of its execution from the start to the end. The idea is to provide learners with continuous feedback while tapping on the lines, such as temporarily highlighting the line when tapped while also showing the order in which it was chosen. However, even if the tutor provided feedback which is triggered with each tap on a line, the learners are still expected to remember the sequence in which they have tapped the lines. Perhaps it is better for the order of the lines chosen to be permanently visible to the learners after tapping i.e. by displaying the order in each line. For example, a LOC with the start of a for loop i.e. `for element in list:` will have the numbers 3, 6, 9, 12 shown; because this for loop line was the 3rd, 6th, 9th, and 12th line that the learner has tapped. Another option may be to combine this with a replay feature where learners can recall the order in which they have tapped the lines by visually seeing them highlighted one at a time. All the ideas mentioned for scaffolding may help, but the task may still be overwhelming for learners, especially when solving problems with a high number of LOCs and therefore long sequence of execution. Some programs may not have many lines, but some lines are executed more than once (i.e. loops). Therefore, the “tap and trace” activity was not implemented since this exercise may result in cognitive overload especially when working with exercises on loops.

3.2.2 Parsons Problems with Distractors

Parsons Problems with distractors is a variant of Parsons problems containing extra LOCs that the learner needs to remove, since distractors are not needed in the correct solution. So, apart from rearranging the LOCs, the learner is required to eliminate distractors. Parsons problems with distractors are aimed to enhance code reading and code writing skills. The task starts by the tutor showing the expected output of the code that must be produced by reordering the LOCs (Figure 3.1, left). The randomised LOCs with the distractors are shown in Figure 3.1 (middle). Learners most likely need to refer to the problem statement while on task, so the problem details need to be easily accessible from the screen containing the Parsons problem. This is not shown in the screenshots provided, but the problem statement will be designed to be shown as a silhouette dialog box on top of the Parsons problem that can be easily toggled open and closed

on a tap of an icon/button instead of being in a separate screen. A dialog box for the problem statement may be better than being displayed on a separate screen for learners to preserve their thought process and their mental model/s for the problem. When the student is ready, the solution can be evaluated at any time by pressing on the SUBMIT button at the bottom-right hand corner.

When a user removes a LOC, it is added to the "Trash". The LOCs removed can be accessed by tapping on the trash bin icon, which displays a dialog box containing the trash contents (Figure 3.1, right). The user can select LOCs in the trash to be re-appended to the puzzle. By default, lines that are retrieved from "Trash" will be appended at the start of the puzzle, so that the student can see the retrieved lines easily. Adding retrieved LOCs at the start of the problem may be better than appending at the end since this will avoid users having to scroll down just to see retrieved LOCs. The described variant of Parsons problems with distractors was evaluated in my pilot study (Chapter 4). I have evaluated Parsons problems without distractors (regular Parsons) in my pilot study and third evaluation study (Chapters 4, 7).

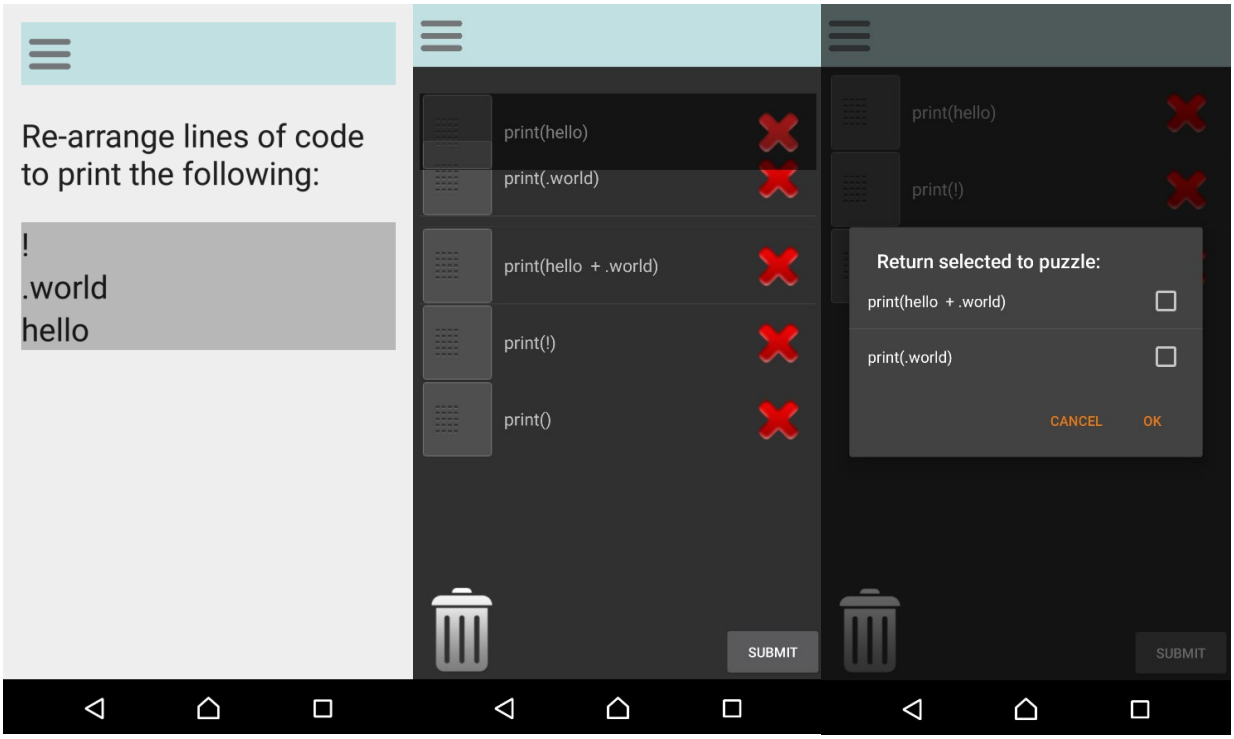


Figure 3.1 (left) problem statement, (middle) Parsons problem, (right) Trash contents

3.2.3 Parsons Problems with Distractors and Incomplete LOCs

Another variant of Parsons problem includes an additional level of complexity: incomplete LOCs. This type of learning activity is also aimed to enhance code reading and code writing skills. The initial screen presented to the learners is similar to the initial state of Parsons problems with distractors (Figure 3.2, left). The manner of deleting and retrieving LOCs is the same as in Parsons problems with distractors. The problem screen (Figure 3.2, middle) is shown once the learner clicks on the "OK" button on the problem statement dialog box. Students may view the problem details at any time by clicking on the "?" icon on the top right-hand corner. Solutions are evaluated similarly to Parsons problems with distractors, by pressing on the SUMBIT button at the bottom-right hand corner.

The additional task is presented in Figure 3.2 (right), where the learner must tap on the incomplete LOC to be presented with another dialog box with choices to complete the LOC. The idea for including LOCs with missing keywords was inspired from the extension used in MobileParsons (Karavirta et al., 2012). The student is asked to complete the missing elements by choosing from a predefined set of choices. Such tasks are not simple, since there may be problems containing four types of LOCs: non-distractor LOCs with and without missing keywords, as well as distractors with and without missing keywords. It may be counter-intuitive to design problems with distractors containing missing keywords, but it is possible. The puzzles also need to be carefully designed such that there are no two LOCs that can be identical unless this behaviour is intended. For example, a LOC **x** missing some keywords: `if my_num_____:` can end up being identical to another LOC on the same puzzle **y**: `if my_num > 5:`. Identical LOCs can be avoided by choosing the choices carefully for filling in the missing keywords. If problems are evaluated for correctness by checking the learner's solution with the ideal solution, problems need to be carefully designed so that no distractors can be used to substitute correct LOCs. Therefore, the solution must also be checked against specified constraints and rules. However, a better way of evaluating the solutions is possibly to implement a small Python engine running to check solutions for correctness.

The combination of Parsons problems with distractors and incomplete LOCs in the same problem need to be carefully designed to provide the right amount of complexity. The combination was not implemented, since it may be overwhelming for students. However, I have implemented a variant of Parsons problems with distractors, and separately another variant of Parsons problems with incomplete LOCs. I have evaluated Parsons problems with incomplete LOCs in my first evaluation study in Chapter 5, and in my third evaluation study in Chapter 7.

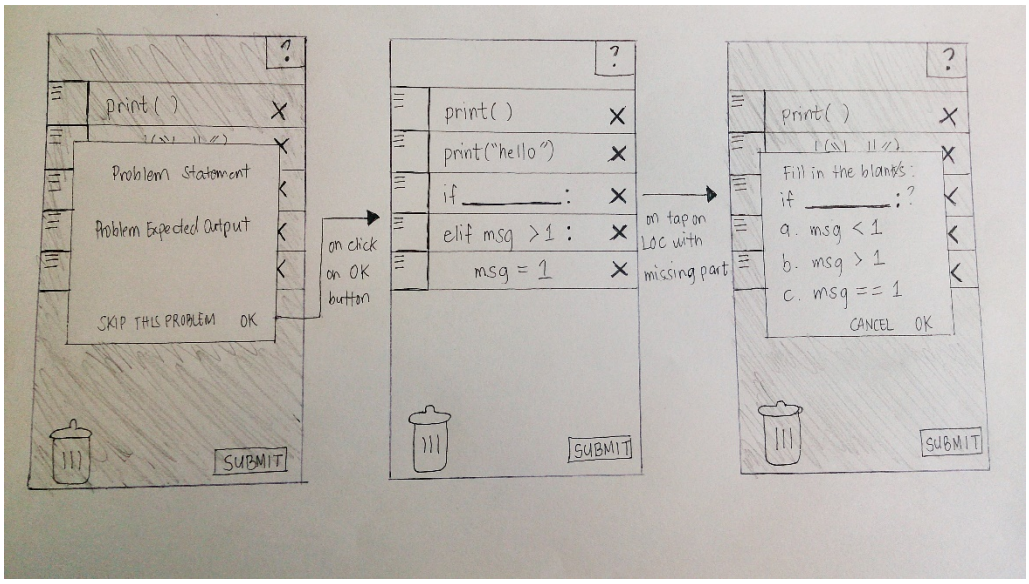


Figure 3.2 Initial design screen flow for Parsons problems with distractors and LOCs with missing keywords

3.2.4 Identifying Erroneous LOCs within a Code Snippet

I also designed learning activities which enhance debugging skills. The tutor presents learners with an expected output and an erroneous code snippet. The task is to identify the erroneous LOCs within the provided code. The task of identifying errors in code aims to enhance debugging skills. The notion of identifying errors in code is similar to identifying distractors in Parsons problems. The main difference from Parsons problems with distractors is that this task does not require ordering of the LOCs, but only identification of erroneous LOCs. Like the other activities, the tutor will firstly present the problem details which includes the expected output in a dialog box (Figure 3.3). The problem details can be viewed again at any time by clicking on the "?" icon on the top right-hand corner. The screen containing the erroneous code will be revealed once the learner clicks on the "OK" button on the problem statement dialog box. Figure 3.3 shows that the student has identified erroneous LOCs (highlighted in yellow). The solution is evaluated once the student clicks on the "SUBMIT" button on the bottom-right hand corner. I have evaluated the activity of identifying erroneous LOCs in my second and third evaluation studies (Chapters 6-7).

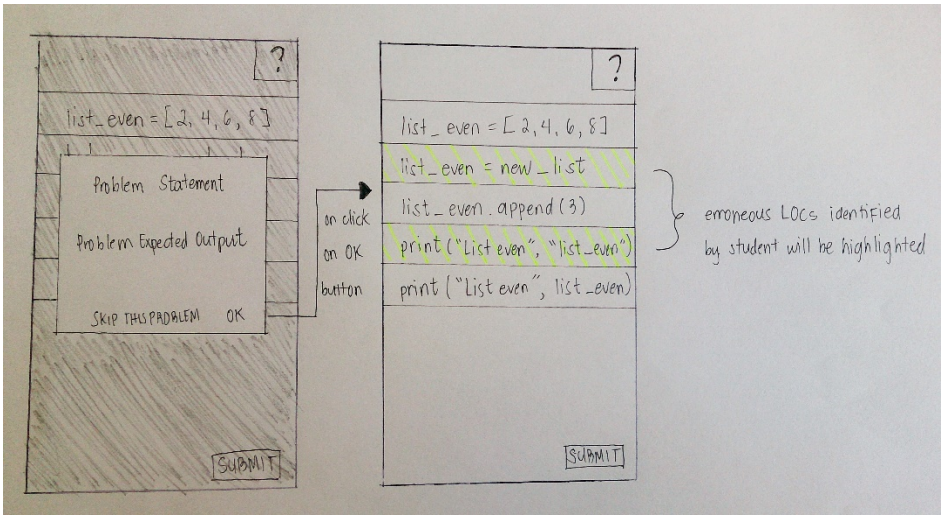


Figure 3.3 Initial design screen flow for task identifying erroneous LOCs

3.2.5 Identifying Types of Errors of Erroneous LOCs

Identifying erroneous LOCs (described in Section 3.2.4 and illustrated in Figure 3.3), can be extended further into another task to further support debugging skills. The task is to identify the type of error the LOC has, i.e. syntactic or logical/semantic error. Identifying the type of error is illustrated in Figure 3.4, where learners start by first identifying the erroneous LOCs (highlighted in green). Once the erroneous LOCs are successfully identified, the learner taps on each identified LOC to be presented with a dialog box to identify the corresponding type of error. The type of error must be identified for all the erroneous LOCs from the previous exercise. The task of identifying the error type in LOCs was not implemented. However, it might be a good activity to enhance future versions of the tutor.

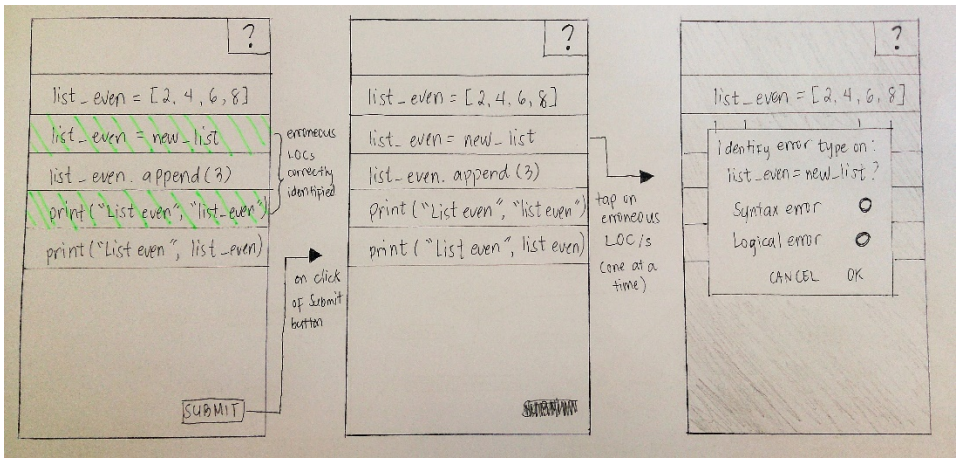


Figure 3.4 Initial design screen flow for task identifying types of error (syntax or logical error)

3.2.6 Identifying Type of Error Upon Code Execution

Another variant of identifying code errors was designed where learners need to predict the type of error they will get upon execution of the code (compile time error, run-time error, or logical error) instead of identifying the type of error that a LOC has (Figure 3.5). The task of identifying the type of error upon execution can be implemented independently or in conjunction with another task i.e. firstly ask learners to identify the erroneous LOCs in the code. This type of activity was not implemented because of the types of errors in Figure 3.5. More specifically, compiling errors are not checked by default in Python. Python is a dynamic language where compiling errors are not checked by default; unlike in static languages like Java/C/C++. It is better for learners to identify the type of run-time error a LOC produces i.e. SyntaxError, IndentationError. A possible direction for future work will be to implement this task in the tutor to further support debugging skills.

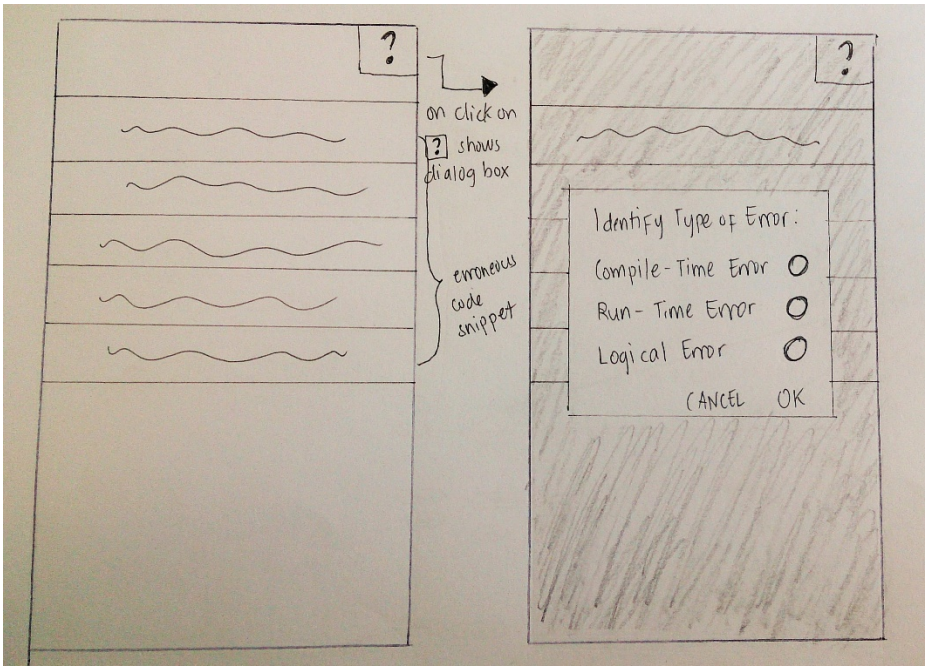


Figure 3.5 Initial design for task identifying type of error on output

3.2.7 Fixing Erroneous LOCs

Another activity was designed to extend the task of identifying erroneous LOCs in the code to support debugging and code writing skills. It may not be ideal to ask learners to simultaneously identify and fix erroneous LOCs as doing them concurrently may cause learners to be overwhelmed. Therefore, learners are asked to do the tasks sequentially instead of simultaneously. Firstly, the learner is asked to identify erroneous LOCs in the code. After the learner successfully identifies LOCs, he/she is then asked to fix the erroneous LOCs that he/she have just identified. To do that, the learner selects from a set of choices to replace the incorrect LOCs similar to controls used when solving incomplete LOCs in a Parsons problem. The initial design for this task is illustrated in Figure 3.6. In selecting the correct LOCs to replace the erroneous ones, it is possibly better to highlight erroneous LOCs, and then the student can either tap or swipe on the erroneous LOCs to find and select the correct LOC/s for each respective LOC. I have evaluated the task of fixing LOCs in my second and third evaluation studies (Chapters 6-7).

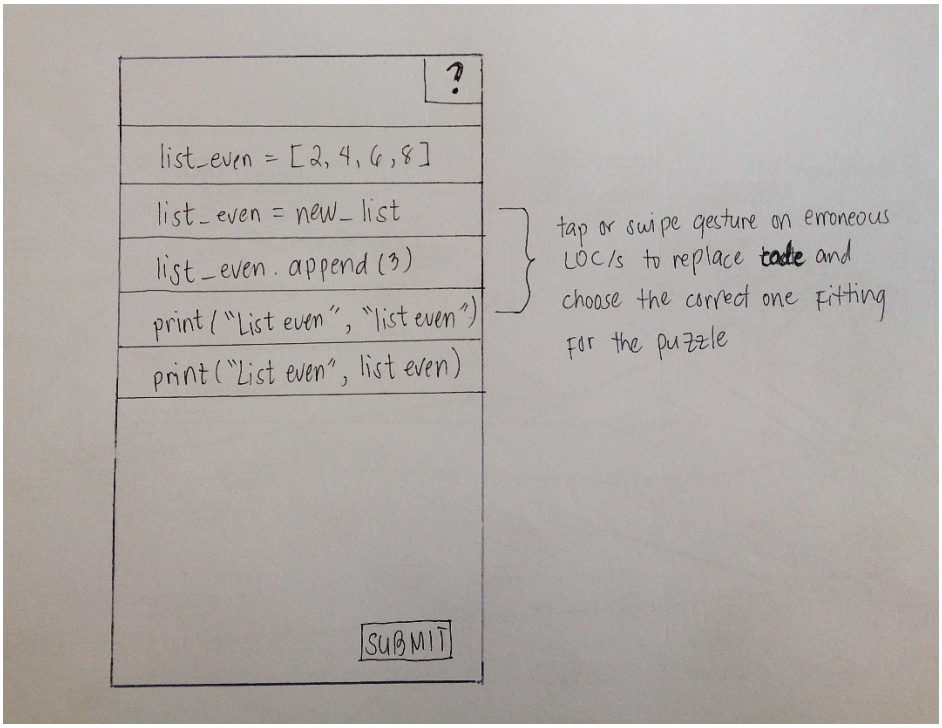


Figure 3.6 Initial design for task of fixing erroneous LOCs

3.2.8 Identifying Expected Output Given a Correct Code Snippet

Another activity was designed where the student is asked to predict the expected output of a code snippet. The output prediction task is focused on supporting code reading and code tracing skills. The tutor starts by displaying the problem details in a dialog box. The learner is also presented with the code snippet that can be viewed upon closing the dialog containing the problem details. The learner selects the expected output when the code is executed from a multiple-choice question (Figure 3.7). I have evaluated output prediction tasks given correct code in my second and third evaluation studies (Chapters 6, 7).

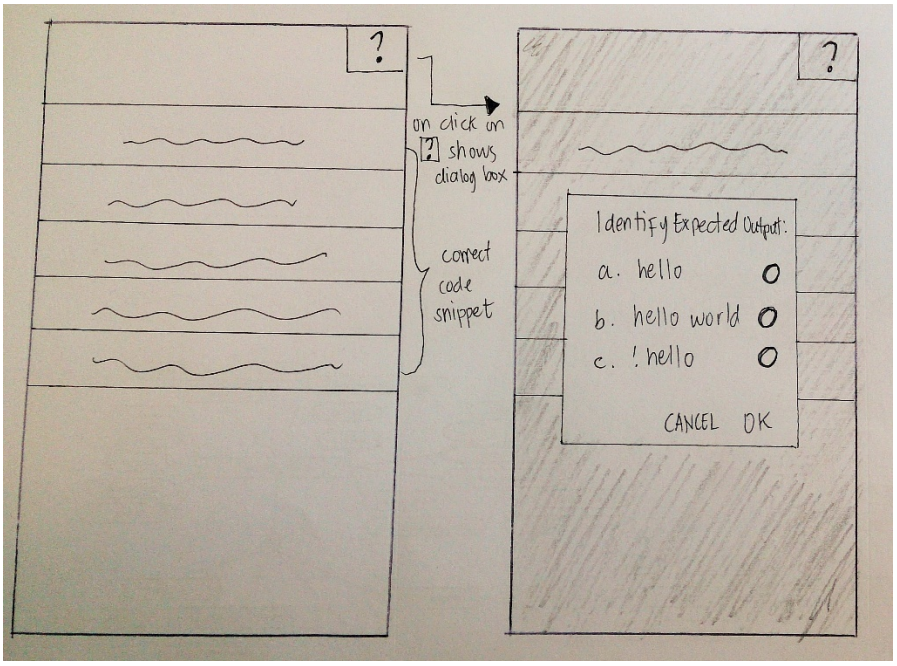


Figure 3.7 Initial design for task identifying expected outputs given a correct code snippet

3.2.9 Identifying Actual Output Given an Erroneous Code Snippet

A variant of output prediction was designed specifically for erroneous code. The task of output prediction is similar to the one described in Section 3.2.8. However, the requirement in this task is to predict the actual output, and the given code contains errors. The design for the task is like illustrated in Figure 3.7, but the dialog box will have a different label asking to identify the actual output. Since the code contains errors, it may be possible that the output does not display anything but errors. Therefore, one of the provided choices specifies that no output is displayed. I have evaluated output prediction tasks when given erroneous code in my second and third evaluation studies (Chapters 6, 7).

3.3 Plans for Implementation and Evaluations

The goal for developing the tutor was to ensure a smooth learning experience and avoid data loss during evaluations. Deployment of the tutor as a stable and stand-alone product was outside of the scope of this research project. In the preliminary stages of development, the focus was to implement the basic features of the tutor to be evaluated in a pilot study. At the same time, the goal was to establish a robust yet flexible foundation for later evaluations. In this section, I describe initial plans for development and conducting evaluations.

The first set of activities implemented was my variant of Parsons problems with and without distractors. The authoring of the problems was focused on the target audience. More specifically, the problems should be at the appropriate level for my participants. Problems should be carefully designed to have the right amount of difficulty and complexity to motivate the learners enough to use the tutor. In authoring the Python problems, it must also be kept in mind to focus on the main research questions of this resesarch project which includes enhancing debugging and code writing skills.

The features of the preliminary version of the application included a working user interface for the entire tutor, consisting a working drag and drop interface for the Parsons problems. The aim was to develop a set of Parsons problems with and without distractors on various Python programming topics. The initial topics were selected based on informal interviews conducted with COSC121 tutors. I expected that the list of topics included in the tutor to be altered depending on results from my evaluation studies.

The tutor was evaluated at various stages of development. The pilot study was exploratory and the one of the main goals was to gather feedback about the usability of the tutor. Therefore, in the first version of the tutor, participants were allowed to select problems freely. Further evaluations were made to address the goals and my research questions. The tutor was mostly evaluated by volunteers recruited from COSC121 (Introduction to Computer Programming) in University of Canterbury. COSC121 students were appropriate participants for my evaluation studies since COSC121 is an introductory programming course. Moreover, COSC121 students are learning programming with Python which is the same programming language taught by PyKinetic. The tutor was evaluated during a week wherein COSC121 students were less busy. The course is taught twice a year, therefore giving more flexibility for conducting evaluation studies. The design of evaluation studies is an important part of this resesarch. The research questions were addressed in the evaluation studies, either together or individually. A pilot study and three full evaluation studies were conducted on PyKinetic (Chapters 4-7).

I gathered data from my evaluations through manual observations and logs recorded by PyKinetic. These logs are text files (.txt) that will contain information about the problems and individual timestamps for each action made by the learner. I analysed the logs by writing Python programs to parse through them

efficiently and output them into .csv files. Then, I will use SPSS Statistics software to run my statistical analyses.

3.3.1 Terminology used for Evaluations

The following are some terminology used in the rest of the thesis, specifically for Chapters 4-7 which are covering the evaluation studies:

- An *activity* is an exercise based on a description, a code snippet, and possibly additional information.
- A *problem* is a task which may contain one or more coding activities, where activities are presented one at a time.
- A *test case* is executable code which may contain more than one line of code; consists of parameters and calls to one or more functions.
- *Actual output* is the true displayed result of a program when executed, regardless of whether the code contains any errors.
- *Expected output* is the anticipated displayed result when a program which contains errors is "fixed" based on the problem requirements.

3.4 Technical Overview

This section provides an overview of the initial technical aspects of PyKinetic. Technical modifications made for an evaluation study are described in each respective chapter for the study. PyKinetic is an application built for the Android operating system. The main programming language used in building the tutor is Java. XML was also used for the layout and other Android resources. PyKinetic was built using a working environment which involved Android SDK provided by Google, Eclipse IDE for development coupled with the Android Development Tool (ADT) plugin which allowed integration of Android SDK with Eclipse. However, the support for ADT plugin has ceased in 2016. Since Google ceased supporting the ADT plugin for Eclipse, I have since then migrated PyKinetic to Android Studio – Android's official IDE.

The oldest Android version compatible with the first version of PyKinetic was Android version 3.0: Honeycomb, supported by Android Software Development Kit (SDK) version 11. The target version of Android devices was Android version 4.1.2 Jelly Bean, supported by Android SDK version 16. The minimum SDK indicates the minimum version of Android that the tutor supports. However, the target version does not imply that later versions of Android are not compatible with the tutor. The target version only sets the default theme for the application. The minimum and target versions were set based on most

known commonly used Android versions at the time and are subject to change in later evaluation studies.

The initial system architecture only contains a client with a database layer (SQLite). A server is implemented in later versions of the application for gathering data in evaluation studies. The SQLite database stores the problems and their solutions. A library called Android SQLiteAssetHelper (Github, 2019b) was used to support retrieving data from SQLite and help manage the creation and version management of the database.

The interface of Parsons problems requires drag and drop capabilities. Learners should be able to drag and drop LOCs in their respective positions. For Parsons problems with distractors, learners should be able to remove LOCs from the problem but also be able to retrieve LOCs which were removed by mistake. There was no library found for implementing Parsons problems in the Android platform. However, a library called DragSortListView (Github, 2019a) was found helpful for the drag and drop interface required in Parsons problems. The library was easy to work with and allowed removal and insertion of elements in a list which are necessary for Parsons problems with distractors.

3.4.1 Application Architecture Overview

For establishing a good foundation in the development of PyKinetic, an architectural design pattern was applied: The Model-View-Controller (MVC) pattern. The model part represents the objects in the tutor i.e. problem, LOC. The view contains the user interface with a visual representation of the data in the model. The controller keeps the view and model separate by controlling the data flow into the model and updates the view as necessary. In PyKinetic, the model contains objects such as problems and LOCs. The view contains the problem interface where learners perform interactions. Learner interactions are responded to by the controller updating the model and view. For example, when a learner submits a solution by tapping on a submit button, an event is triggered. The controller updates the problem, LOCs and feedback in the model as needed and presents feedback to the user which is visually displayed in the view. The implementation of a design pattern promotes a neat architectural groundwork in supporting ease of implementation of additional requirements. Furthermore, a clear separation of the processes in updating the view and controlling data flow is necessary for optimizing the performance of the application. For example, sending data from the view to an outside source (i.e. server) while trying to update the data shown in the view may cause visual delays to the user interface if both tasks are done simultaneously. Establishing a MVC pattern separates the components, which allows asynchronous processing of background tasks to minimise the loading time on the user interface.

All versions of PyKinetic were written in Java and XML code. The Java code contains most of the contents (model, view, and controller) of the application. The XML files are divided into the following classifications: manifest, layout, and values. There is only one manifest file in the tutor, but there are

multiple files for the layout and values. The manifest is a file with a technical overview of the Android application which contains the following information: version code, minimum and target SDK versions (to specify compatibility to Android devices), application permissions, and a list of the application screens. The application permissions are provided to request access to user data and device system features. Requested permissions were minimised to only those genuinely needed by the application; because these permissions are reflected to the user before installing the application. Permissions must be minimised since unnecessary permissions might mislead users about the true intentions of the application. Therefore, only essential permissions should be included to avoid discouraging users from installing the application. Examples of permissions include accessing the device camera, accessing network changes, and writing data to the device. In all versions of PyKinetic, all permissions were requested before installation. However, note that install-time requests are only applicable to devices running Android 5.1.1 or with target SDK version 22 or lower.

The XML files for layout provides a skeleton structure for the views of the application screens. The contents of the view are mostly retrieved from the SQLite database. Contents are retrieved, and views are populated using Java code. Lastly, the values XML files are classified into the following: colour, dimensions, strings, and styles. Only colour and strings files are used in the versions of PyKinetic because the dimensions and styles files are usually used for aesthetics (not the focus of this resesarch). The colour file contains the list of colours used in the application. The strings file contains static texts used throughout the application to label various elements such as buttons, text boxes, and alert messages.

4 PROTOTYPE AND PILOT STUDY

This chapter presents my first prototype of PyKinetic which contained a set of Parsons problems with and without distractors. My pilot study is also covered in this chapter. The main research question addressed in this chapter is (R2): What problem-solving strategies can be observed when solving Parsons problems? Moreover, even though my pilot study was exploratory in nature, findings from the pilot study was used as a stepping stone towards addressing my other research questions (R3-R7).

The chapter starts by introducing my first prototype. Next, I discuss improvements made to the prototype to develop PyKinetic_Pilot, evaluated in my pilot study. After that, technical details specific to the PyKinetic_Pilot are exhibited. Then, the rest of the details about my pilot study are covered: research goals, experimental design, data collection, general findings, responses to the questionnaire, strategies used by participants as observed when solving Parsons problems with and without distractors, challenges and limitations of the study, and finally discussion and conclusions.

4.1 First Prototype

A set of Parsons problems were designed at various levels of complexity, covering a variety of programming concepts. The level of difficulty of Parsons problems are appropriate for my target learners who are introductory programming students. Furthermore, the drag and drop nature in solving Parsons problems are appropriate for a mobile device (Chapter 2.7). Parsons problems can be designed to specifically address common misconceptions about Python by providing *distractors* which are extra lines of code that are not needed in the correct solution. Both Parsons problems with and without distractors require learners to rearrange LOCs by drag and drop motions. However, when solving Parsons problems with distractors, learners have an added task of eliminating distractors. There is evidence that solving Parsons problems is positively correlated with code writing (Denny et al., 2008). Parsons problems may also enhance code understanding skills based on the nature of the task. On the other hand, Parsons problems with distractors possibly also enhance code writing and code understanding skills; as well as debugging skills due to the additional task of identifying unnecessary LOCs.

There are eight Python topics included in the prototype: *String Manipulation*, *Conditional Statements*, *Lists*, *For Loops*, *While Loops*, *Dictionaries*, *Tuples* and *Data Types*. The tutor contains 53 Parsons problems (40 with distractors, and 13 without distractors). Each topic contains 3-8 Parsons problems with distractors and 1-3 without distractors. The problems are designed with varying complexity. Each problem is assigned a value based on its complexity ranging from 1 (the easiest and/or less complex)

to 9 (the most difficult and/or most complex within the context of the topic/category it belongs to). The complexity of a problem is relative to the other problems in the prototype and was determined based on my experience in teaching Python programming. Criteria for identifying the complexity of a problem include depth of nesting of the code constructs, number of LOCs in the problem etc.

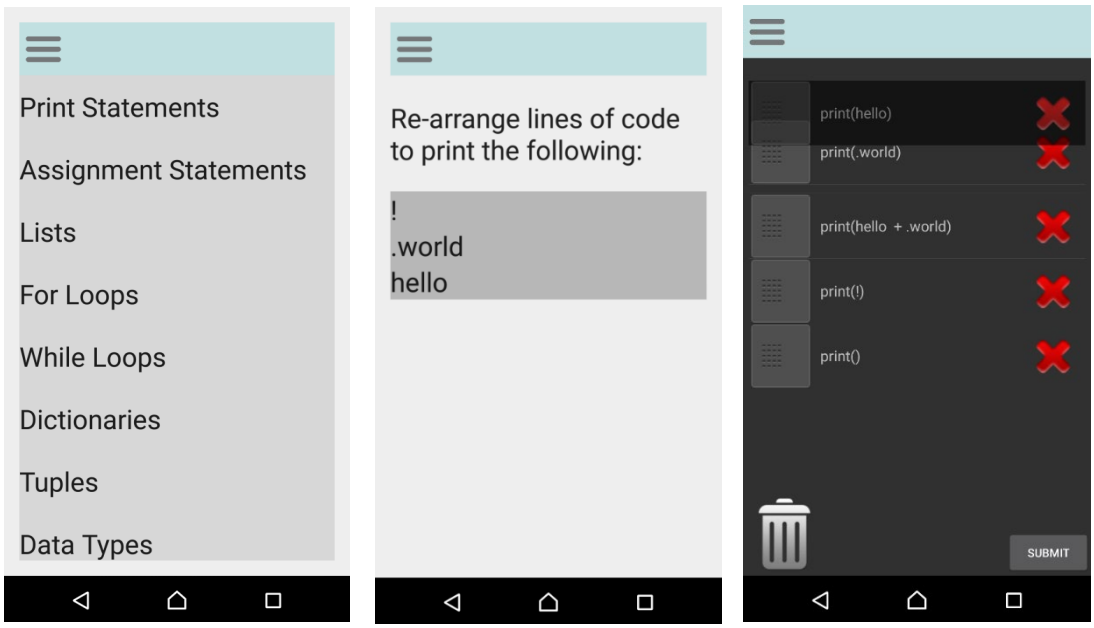


Figure 4.1 Screenshots (from left to right): selection of topic screen, problem details screen, and Parsons problem screen

The initial prototype contained three application screens: topic selection screen, problem description screen, and Parsons problem screen (Figure 4.1). The *topic selection screen* allows the learner to choose a topic he/she would like to work on. When a topic is selected, a random problem is given which covers the selected topic. The *problem description screen* contains the problem instructions, and the expected output of the Parsons problem. Finally, the *Parsons problem screen* serves as the problem-solving space. The problem area and solution area for my variant of Parsons problems are combined as one unlike other implementations which had these two areas as separate (Parsons and Haden 2006; Ericson, Margulieux and Rick 2017; Ihantola and Karavirta 2011; Karavirta, Helminen and Ihantola 2012; Ihantola, Helminen and Karavirta 2013; Harms, Chen, and Kelleher 2016; Kumar, 2018; Hosseini, 2018). Therefore, the initial state of the screen shows all the LOCs and learners rearrange them within the same area. The LOCs are presented in a new randomised order every time a problem is attempted. The student rearranges LOCs by holding on the drag icons on the left. The first two LOCs are being rearranged by the learner in Figure 4.1 (right). Distractors are eliminated by tapping on the red icon for each LOC.

4.2 PyKinetic_Pilot

The prototype was further developed for my pilot study evaluation (PyKinetic_Pilot). Instead of covering all eight topics, there were only seven topics (Dictionaries was removed) to adhere to the material covered in COSC121 (Introduction to Computer Programming) when the pilot study occurred. Furthermore, fewer problems were selected for the study: 21 problems in total, where each topic contained three problems (two with distractors and one without). The number of problems were reduced because of the session length.

The screens in the prototype are shown in Figure 4.2: start screen, topic selection screen, and Parsons problems screen. A start screen was implemented so that learners can explicitly indicate when they are ready to start use the tutor. For the problem details, instead of a separate screen like in Figure 4.1 (middle), this is presented in a dialog box (Figure 4.2, right).

The aesthetics and functionality of the user-interface (UI) were improved. The start screen is showing a temporary image as a placeholder for the logo of the tutor. When the user clicks on the Start button, he/she is presented with the second screen, which shows an expandable list of the seven topics. Unlike the implementation described in the first prototype (Section 4.1), the interface allows learners to select a specific problem within a topic (see Figure 4.1: left and Figure 4.2: middle for comparison). Clicking on a topic toggles between expanding and collapsing the list of Parsons problems for that topic. In addition, topics include the option of selecting a random problem; where the tutor provides a randomly selected problem from the chosen topic. The problems within each topic are displayed in an increasing order of complexity. The third screen is the main screen of the application for solving Parsons problems. This screen initially shows a dialog box which contains the problem details and the expected output for the Parsons problem. Instead of showing the problem details in a separate screen as described in Section 4.1, the UI was improved to achieve a better learning experience. The problem details are shown in a dialog box to follow my design guidelines in Chapter 3 and adhere to the multimedia learning principles (Chapter 2.4). The problem details are displayed in a dialog box overlain on the Parsons problem screen (Figure 4.2, right) so that learners can focus on one screen (instead of previous design Figure 4.1: middle and right). This design decision simplifies the interface and helps reduce *extraneous cognitive load* (Sweller, Ayres, and Kalyuga, 2011). Details about the problem is highly accessible, it can be viewed multiple times by tapping on the "?" icon on the top-right hand corner. Problems can be skipped by tapping on the "SKIP THIS PROBLEM" button on the dialog box.

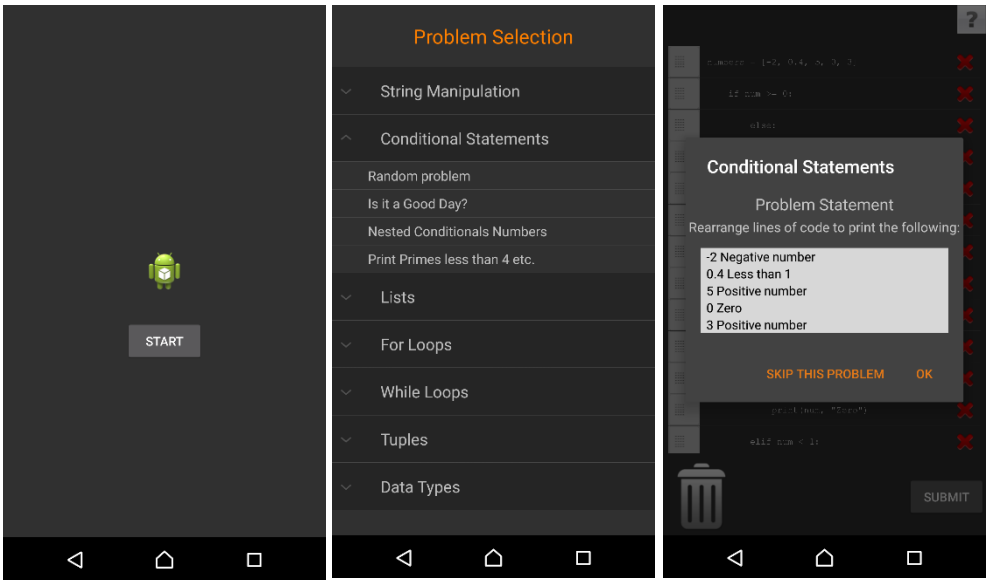


Figure 4.2 Screenshots of improved interface (from left to right): start screen, topic and problem selection screen, and Parsons problem screen (silhouette beneath problem statement dialog box)

The controls in the Parsons problem screen were kept the same. LOCs are moved by long pressing on the drag handle icons on the left of each LOC while moving in an upward or downward motion to drop LOCs to a desired position. For Parsons problems with distractors, distractors are removed by tapping on the 'X' icon on the right-hand side of each LOC. Parsons problems with no distractors do not display any 'X' icons as a visual cue to the learners. Learners can retrieve eliminated LOCs by tapping on the trash icon and selecting the LOCs desired to be retrieved back to the problem. The list of deleted LOCs is also displayed in a dialog box (Figure 4.3).

When solving Parsons problems, feedback is only given upon clicking on the Submit button. Since this was my first study, I did not have any information on the nature of feedback that are appropriate for novice programmers when learning solving Parsons problems with and without distractors. Therefore, only simple feedback was provided for both Parsons problems with and without distractors:

1. "Great Job! You are correct!"
2. "You have distractor/s in your solution."
3. "You are missing some LOC/s in your solution."
4. "Check the order of your solution."

The first feedback is given when the user submits a correct solution. The second feedback is given when there is at least one distractor left in the student's solution. The third feedback (which can be given together with the second feedback) is given when there is at least one missing LOC. This can happen when the student removes a LOC required for the correct solution. The fourth feedback will only be given if the

solution contains the correct set of LOCs; since it might be confusing to give feedback about the order of the LOCs when there are missing LOCs, or the solution contains distractors. The fourth feedback is the only incorrect feedback that learners receive when they are attempting a problem without distractors.

There were aesthetic improvements in the Parsons problem screen (Figure 4.3). The sizes of the icons i.e. drag holder and remove icon were decreased to accommodate longer lines of code. The font face was also changed from the default font used in Android to Courier New; which was selected since this is commonly used in IDEs like Geany. The menu bar (sandwich icon on top-left hand side) as seen in initial designs in (Figure 4.1) was removed as it was deemed unnecessary since the tutor only contains three screens. The menu bar will be added in the future for ease of navigation if more screens are implemented.

Figure 4.3 Improved Parsons problem interface design

4.3 *Architecture and Development*

The general architecture of the application is described in Chapter 3.4.1. Specific characteristics of the version of PyKinetic used in the pilot study are described here. The statistics of the code not including the libraries used are reported in Table 4.1.

Table 4.1 Code Statistics

	Java	**XML**
Classes/Files	20	10
Total LOCs without blank lines	2152	264
Source code lines	1815	264
Comment lines	337	0

With regards to the manifest XML file in PyKinetic_Pilot, there were only two Android permissions in this version of the tutor: reading and writing data in the device. These permissions were needed to record the logs in the device used in the pilot study; where the logs contained user actions.

4.3.1 *Storage*

The problems and all its details were stored within the SQLite database in the application (Figure 4.4). Python topics offered by the tutor were also stored in the database. However, feedback was not stored in the database. Since the tutor only gave simple feedback, these were stored in the strings XML file together with the other texts used in the application. For implementing problem specific feedback or complex feedback in the future, these should be stored in the database.

Figure 4.4 System/Application Architecture Diagram

The ER diagram of the database used in the pilot study is shown in Figure 4.5 (without the attributes). Categories (Python topics) can be created without containing any problems. A problem must belong to only one category and contains 3-16 LOCs. The LOCs are included in one or more problems where the problems

do not necessarily have to be related. For example, the LOC `else:` is used in many problems which are not similar. This particular "else" clause (with indentations) was used in four problems: three from the conditionals category and one from the while loops category. Problems may also contain an expected output which consists of at least one line. There may be problems that do not have an expected output, so the database is structured to allow flexibility. Each line in the output may be used in multiple problems. For instance, output lines displaying "1" may also be contained in the expected output of another problem.

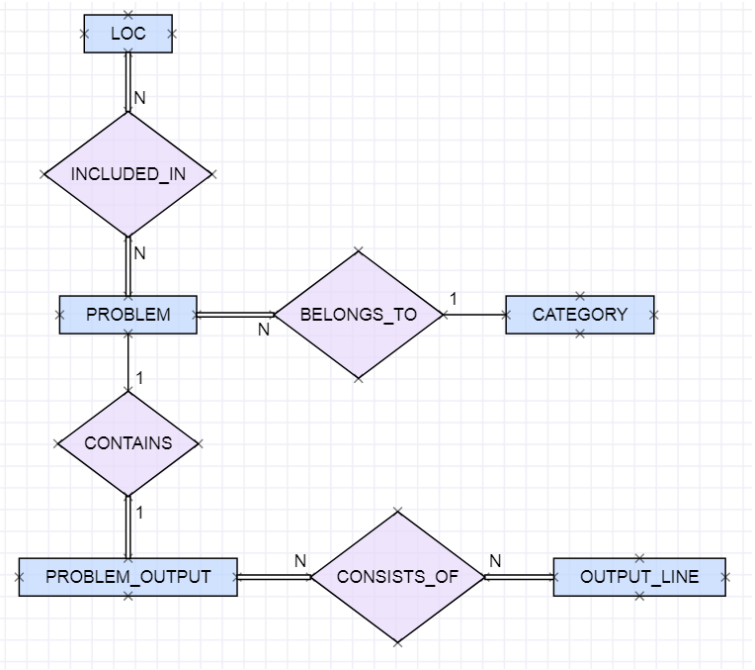


Figure 4.5 ER diagram of the database in PyKinetic_Pilot without the attributes

Solutions for each LOC were stored in the database as separate integers instead of a list of integers. Each integer corresponds to a correct position of the LOC in the problem. For example, for problems with LOCs that are initialising variables, these variables are normally declared at the start of the program. Sometimes, the order in which the initial variables are declared does not affect the program. In the problem shown in Figure 4.6 (top), the goal is to reorder the LOCs to display the message "*Hello I am a sentence here*." after the distractors were removed there were only 4 LOCs (Figure 4.6 bottom). In this example, LOCs in positions 1-3 can be repositioned in any manner, as long as the LOC in position four remains in its place. Therefore, position 1, 2, and 3 are all stored as correct solutions for the first three LOCs in Figure 4.6 (bottom).

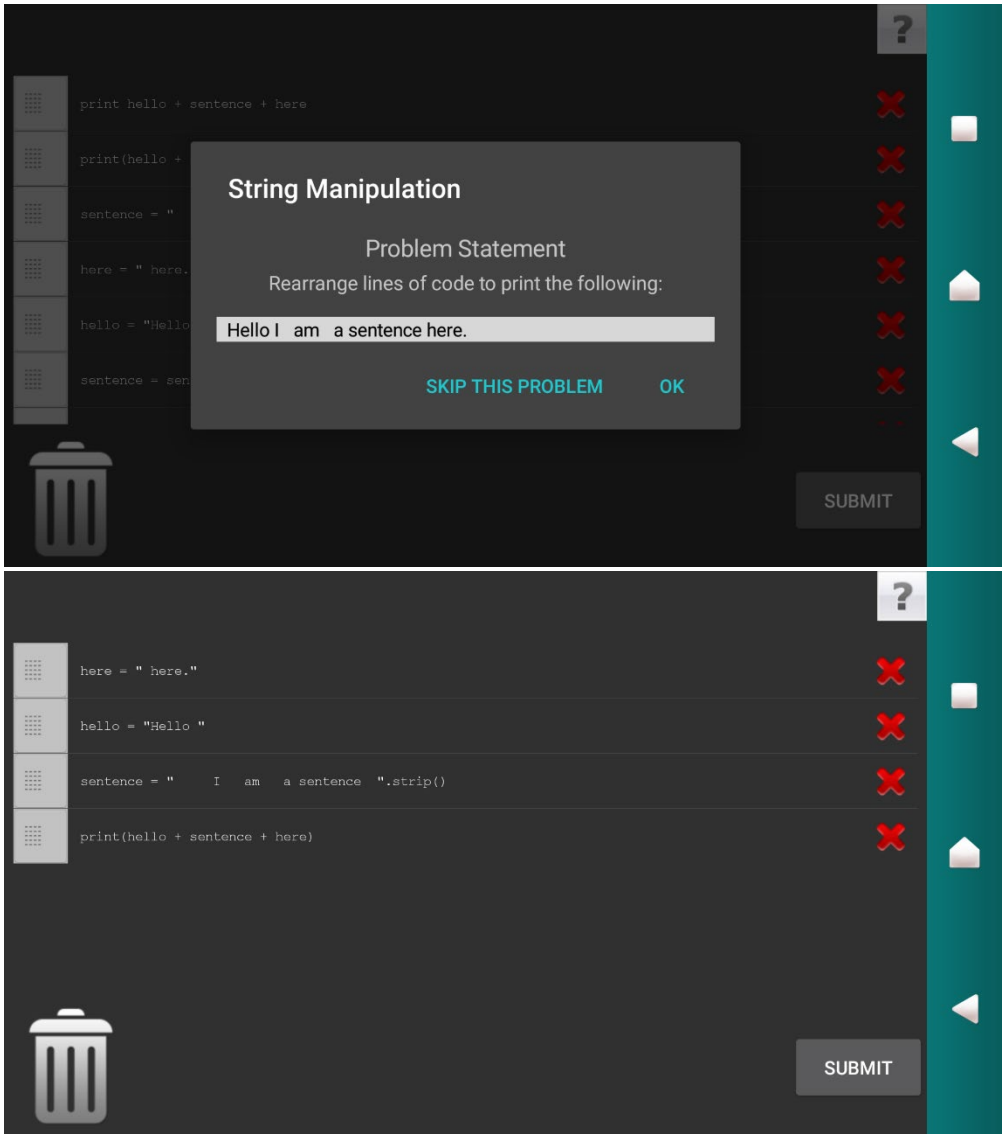


Figure 4.6 Example of a problem with LOCs that can be correctly positioned in multiple places

There were no Parsons problems with LOCs where their order in the problem are dependent on the positions of other LOCs. An example of this type of problem is shown in Figure 4.7, where the LOCs needs to be reordered to display an output which says, “my number is 12”. In this example `num2 = num + 2` must be placed anywhere after `num = 10;` therefore, the order presented in Figure 4.7 is incorrect. The solutions stored in the database do not support problems with LOCs like in Figure 4.7; so, problems were carefully designed to avoid issues with evaluating student solutions.

```
1  num2 = num + 2
2  num = 10
3  text = "my number is"
4  print("{} {}".format(text, num2))
```

Figure 4.7 Example of a hypothetical Parsons problem where the correct order of LOCs depends on the position of other LOCs

4.3.2 Navigational Elements

The list of navigational elements used in the prototype is reported in Table 4.2. All elements require single tap actions, apart from the drag icon beside each LOCs which requires drag and drop actions.

Table 4.2 List of buttons, icons, and navigational elements

Screen/Dialog	Element	Outcome
Start	Start button	Navigates to the topic selection screen
	Device's back button	"Closes" the app (not completely, it will still run in the background)
Topic selection screen	Python topic	Toggles between displaying the list of problems contained in the topic
	Random problem	Navigates to a random problem in a selected (toggled) topic
	Problem title	Navigates to this specific problem
	Device's back button	Navigates back to the Start screen
Parsons problem screen	"?" button	Displays the problem details dialog box
	Submit button	Submits a solution and triggers feedback
	Drag icon beside a LOC	Moves selected LOC to location of drop action
	"X" icon beside a LOC	Eliminates LOC and adds it to trash
	Trash icon	Displays its contents (list of deleted LOCs)
	Device's back button	Navigates back to the topic selection screen
Problem details dialog box	Ok button	Closes the dialog box
	Skip this problem button	Navigates to the next problem in the topic (in order of difficulty)
Trash (deleted LOCs) dialog box	Deleted LOC check box	Toggles between selecting or deselecting a LOC
	Ok button	Appends selected LOCs (if any) to the problem and closes the dialog box
	Cancel button	Cancels selection (if any) and closes the dialog box
Feedback dialog box	Ok button	Closes the dialog box

4.4 Research Goals and Hypothesis

As an initial step towards a mobile tutor for Python, I performed a pilot study with the prototype described in Sections 4.2-4.3. The evaluation is an exploratory study and because this is my first study there were no hypotheses defined. Rather, the study had two research goals:

(Pilot_R1) to evaluate the UI of PyKinetic_Pilot, most importantly to investigate whether the orientation of the interface (portrait or landscape) when solving Parsons problems is important for learning;

(Pilot_R2) to identify problem-solving strategies used by students and tutors when working on Parsons problems with and without distractors.

4.5 Experimental Design

The pilot study was completed from 14 September to 17 September 2015 with eight participants (4 male, 4 female) recruited from COSC121. I conducted sessions with five COSC121 tutors later on and asked them to attempt the problems that the majority of students attempted, in order to compare the problem-solving strategies used. The experiment design was similar to the setup by (Ihantola and Karavirta, 2011). In their study, they have observed strategies used by experts in solving Parsons problems. Another study was also conducted a study with novices to identify other strategies (Helminen et al., 2012). The version of PyKinetic used in the study consisted of 21 problems covering seven Python topics (Section 4.2). The problems used in the study had 3 - 16 LOCs, with a maximum of five distractors. Four problems were forced to the landscape mode (Figure 4.8) since they contained long LOCs, while the rest were in the portrait mode. During the evaluation, the course had already covered a set of topics which were included in PyKinetic_Pilot. The pilot study was approved by the Human Ethics Committee of University of Canterbury (Appendix F).

Figure 4.8 Example of a problem in landscape mode (contains 14 LOCs and no distractors)

Each participant had an individual session for one hour. The participants were presented with the information sheet for the study and the consent form (Appendix C, Pilot Study; Appendix E, Pilot Study). I used the think-aloud protocol (Ericsson and Simon, 1993), asking the participants to verbalize their thoughts while interacting with the tutor. I recorded the screen of the device, as well as verbal comments of participants. The screen was recorded using a screen recording application. At the end of each session, participants filled a questionnaire (Appendix B, Pilot Study), the first part of which included questions about their general background, experience with Python programming and with using educational systems and mobile devices. The questionnaire also included questions about the Python tutor, with questions answered with a Likert scale, and provided a space for comments and suggestions for the next version of the tutor.

The goal of the pilot study conducted with the students was to evaluate the usability of the system. Therefore, participants were allowed to ask Python related questions while solving problems in PyKinetic_Pilot. There were no pre-/post-tests, since the pilot study was not intended to evaluate learning benefits from using PyKinetic_Pilot. Therefore, the problems had a range of difficulty to ensure that any participant, despite of their abilities, will be comfortable with using the tutor. However, the overall complexity of the problems was suitable for introductory programming students. Participants were free to choose the topics and problems to attempt in the tutor and the order in which this was done; but students were asked to try to attempt at least one problem from each topic. Dictionaries were intentionally omitted from the pilot, since the participants have not covered the topic at the time the pilot study was conducted.

4.6 Data Collection

I eliminated data about problems which the participants only viewed but performed no actions on, and data about two problems that were found to be buggy. A problem is considered as attempted if the participant made at least one action on it, either by dragging and/or deleting LOCs, viewing the trash, or submitting the solution. A move is defined as the moving of LOCs when attempting a problem. In the way the Parsons problems were setup in the tutor, each move could affect the positions of other LOCs. The analysis was made simpler by counting a move as one regardless of the difference between the starting and ending positions. However, a move is considered as an abandoned move if the LOC was dropped on the same position where it was dragged from. A submission is when a participant submits his/her solution.

Aside from the filled questionnaires, data was collected in three other ways: logs written by PyKinetic_Pilot, audio/video recording of the mobile device screen, and rudimentary note-taking of general observations that I made. There were two types of logs recorded: one for the topic/problem selection screen, and one for the Parsons problems screen. The log for topic/problem selection recorded the following: timestamp on start and end of selection, student actions and timestamp for expanding and collapsing topics, and total time taken for topic and problem selection. The log for Parsons problems screen recorded the following:

- Timestamp on start and end of selection
- On click on Submit- Recorded timestamp, trash contents, student solution and feedback/s given
- On trigger of student actions such as viewing the problem statement again, opening trash, deleting a LOC- Recorded timestamp, trash contents, student solution and label for the following student actions
- Problem details and LOCs in the order in which they were presented initially to the student
- Total time taken for the attempt
- Feedback that would have been given to the last solution of the student (useful in cases where the student did some actions but gave up and did not submit their solution)

Evaluation logs were recorded directly on the smartphone device. The content of the evaluation logs is gathered while the user is using the tutor (in run-time). A log file stored in the device is triggered every time a user clicks on the SUBMIT button and when a user navigates away from an application screen. For every log file, a blank text file is created to avoid duplication of data.

4.7 General Findings

When asked how much experience the participants had with Python, using the Likert scale from 1 (*Not so experienced*) to 7 (*Highly experienced*), the mean reply of student participants was 2.12 (sd = 1.25), with only one student judging his/her experience as 5, and the rest as either 1 or 2. The mean on the same question for tutors was 5.4 (sd = 1.34), ranging from 4 to 7.

I used the Mann Whitney U test to analyse similarities and differences between the two groups, with the significance level of 0.05. Aside from analysing overall results, I also categorised problems by topic (for the seven Python topics used in the study), as well as by the number of LOCs and distractors, and calculated the same measures.

For each participant, I calculated the number of abandoned problems (AP) and completed problems (CP), as well as the following averages per attempted problem: the number of submissions (S), moves (M) and abandoned moves (AM), time taken (TT in minutes), problem complexity (PC), the number of LOCs, distractors (D), and the number of times problem statement was viewed (TV). Table 4.3 presents the averages for the two groups (standard deviations are given in parentheses). The tutors have not abandoned any problems, while two students abandoned two problems each, and two other students abandoned a single problem each. The tutors solved more problems in fewer submissions/moves and in a shorter time, which was expected. The only significant difference on the distributions of the two groups was found for the number of submissions per problem (U = 0, p = .002).

Table 4.3 Overall results

Measure	**Students**	**Tutors**	**Measure**	**Students**	**Tutors**
AP	.75 (.89)	0 (0)	TT	4.02 (2.6)	2.82 (1.1)
CP	10 (3)	12.8 (3.7)	PC	3.79 (.78)	4.36 (.55)
S	**2.72 (1.63)**	**1.29 (.08)**	D	1.66 (.67)	2.03 (.34)
M	13.65 (11.51)	7.3 (1.45)	LOC	8.75 (1.52)	8.87 (1.3)
AM	.62 (1.02)	.26 (.17)	TV	3.1 (.85)	3.7 (.8)

Table 4.4 Results by problem topic (* denotes significance at the .05 level)

Problem Topic	Measure	Students (8)	Tutors (5)	U, p
Conditionals	Time	6.1 (2.96)	4.26 (.71)	U = 6, p = .045*
Lists	Submissions	3.6 (2.98)	1.33 (.33)	U = 7, p = .065
While Loops	LOCs	5.25(2.74)	10.3 (1.56)	U = 38, p = .006*
While Loops	Distractors	.75 (.8)	2.7 (.67)	U = 38, p = .006*
While Loops	Difficulty	1.87 (1.25)	4.6 (.89)	U = 38, p = .006*
Data Types	Difficulty	3.46 (2)	5.53 (.96)	U = 35, p = .03*

I compared performances of students and tutors on each Python topic used in the study (Table 4.4). The tutors were faster in solving problems of most topics (Figure 4.9), apart from *While* loops, but the only significant difference in time was for *Conditionals*. The reason why the tutors needed more time for the *While* loop problems was that they attempted more complex problems of this category (in terms of the problem difficulty, and the numbers of distractors and LOCs – all three differences are significant). The tutors also attempted more complex problems on *Data types*. For problems in *Lists*, tutors needed fewer submissions to complete problems, but this was only marginally significant. The highest number of errors for both groups was for the problems on Lists. One source of confusion was related to indexing lists (e.g. `my_list[:2:-1]`). All students and two tutors commented that they were used to using only one colon when indexing a list. This was probably one of the reasons for the marginal difference between average submissions for this category.

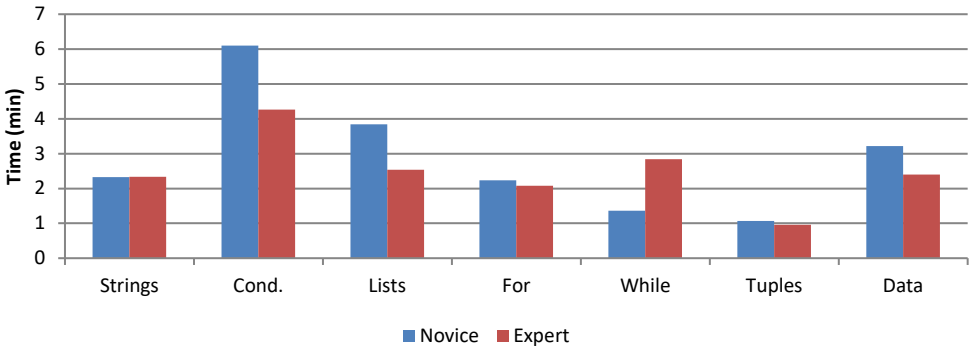


Figure 4.9 Average time for each topic

Both groups were advised that the problems were presented in the increasing order of difficulty. I observed that most students started with easier problems, while the tutors randomly picked a problem from each topic without focusing on the difficulty. Using the number of LOCs, I divided problems into *long* (11-16 LOCs), *medium* (7-10 LOCs) and *short* (3-6 LOCs). For long problems, the tutors (avg = 1.18, sd = .25) outperformed the students (avg = 2.76, sd = 1.58), by completing problems with fewer submissions (U = 5,

p = .03). There was a significant difference for difficulty (U = 34, p = .045) and the number of LOCs (U = 35.5, p = .019) for short problems. For shorter problems with 3-6 LOCs, tutors attempted more complex problems with an average of 5.17 LOCs (sd = .47) and 2.37 difficulty (sd = .44). On the other hand, students completed easier problems which had averages of 4.53 LOCs (sd = .68) and 1.78 difficulty (sd = .97). For tutors, the short problems completed had averages of 5.17 LOCs (sd= .47) and 2.37 difficulty (sd = .44). There were no significant differences between students and tutors on medium length problems.

Table 4.5. Results by the number of distractors in a problem (* denotes significance at the .05 level)

Number of Distractors	Measure	Students (8)	Tutors (5)	U, p
Many Distractors	Time	6.76 (2.77)	3.18 (.52)	U = 1, p = .003*
Many Distractors	Submissions	3.85 (2.35)	1.22 (.22)	U = .5, p = .002*
Medium Distractors	Moves	11.6 (9.82)	5.97 (.74)	U = 6.5, p = .045*
Few Distractors	Distractors	0 (0)	.21 (.04)	U = 40, p = .002*
Few Distractors	Viewed	1.65 (.49)	2.96 (.95)	U = 38, p = .006*

I also compared results in terms of the number of distractors in the completed problems (Table 4.5). Problems were divided into *many* (4-5 distractors), *medium* (2-3 distractors) and *few* (0-1 distractors). The tutors also outperformed the students in terms of the problems solved based on the number of distractors in the problem. The tutors needed significantly fewer moves for problems with a medium number of distractors (2-3 distractors). Furthermore, the tutors were more efficient than students in solving Parsons problems with many distractors (in terms of time and the number of submissions). Another significant difference was that the tutors viewed the problem details more often in the case of problems with few distractors (0 or 1).

4.8 Questionnaire Responses

Overall, the participants were enthusiastic about the tutor, as can be seen from the questionnaire responses, summarized in Table 4.6. The participants from both groups found PyKinetic_Pilot intuitive, easy to use and fun (the average ratings ranged from 5 to 5.6). Some participants seemed to appreciate the interface and commented: ***"It's nice how it pops up showing you what to do."*** and ***"Oh wow, that's cool how you can like slide them up.. that's nice.."***. Most of the participants were enthusiastic about using the application. Most of them also think the application is intuitive, handy and easy to use. Other comments from the participants were: ***"This is cool I like it!"*** and ***"Ohh.. That's cool."***. When asked whether they would use the tutor again, seven out of eight students agreed (the student who disagreed specified he/she

was not interested in learning more about programming). Two tutors also stated they would like to use the tutor again since they gained new knowledge from the tutor, specifically on indexing lists. Both groups were also asked to select programming skills they used in the tutor (reading, syntax and structure and/or logical and semantic reasoning skills). Half of the students responded they used all those skills, while the other students selected 2 out of 3 skills. All tutors responded they used all the skills.

Table 4.6. Summary of questionnaire responses

Question (1 Lowest to 7 Highest)	Students	Tutors
Was the tutor's interface intuitive and easy to use?	5.13 (0.83)	5.6 (0.55)
Was the tutor fun to interact with?	5.13 (0.99)	5 (1.41)
Would you say you have learned some new things and/or enhanced your skills by interacting with the tutor?	5.75 (0.89)	2.4 (1.34)
Do you think it is beneficial that this tutor is developed on a mobile platform?	5.25 (1.04)	4.8 (1.92)
Were problem statements clear enough to understand what needed to be done?	4.5 (1.41)	4.8 (1.92)
Please rate the average difficulty of the problems in the tutor.	4.5 (0.53)	4 (0.71)
Do you think there is enough feedback given when attempting a problem?	2.88 (0.99)	4.2 (0.84)

Some of them did not mind either having the tutor on a desktop computer or a mobile, but about half of the participants mentioned something along the lines of ***"I can really see myself practicing Python with this on the bus or if I'm waiting for someone."***. Some participants were confused about the context of the problems and about some terminology used in the tutor. It was not clear to some that the puzzle screen is scrollable and that some problems included additional lines of code that are not seen on the field of view of the screen. It was also not clear to some that the Xs on the right-hand side of the screen meant that these can be tapped to remove lines of code. The problem instructions will also need to be improved to state whether a puzzle includes the correct lines of code or if the puzzle includes extra lines of code. It does not seem enough to show that the puzzles without any X-s on the right-hand side indicate that all lines of code are needed, although these became clearer as the participants learned using the application. Another feature which was unclear was the fact that removed LOCs can be retrieved. The trash icon does not seem to be sufficient to indicate that this could be done. These issues can be easily resolved by adding a short tutorial video on how to answer the puzzles or an infographic explaining the main screen of the application.

One question in the questionnaire was whether or not the problem statements were clear enough to understand. The average ratings for this ranged from 4.5-4.8, possibly because of the unclear and unknown terminology used. There were a handful of terms which the participants were not familiar with, such as

LOC and distractor. One of the buttons was also not clear enough specifically “Skip this problem”. Many participants assumed that this would keep on skipping through a problem on the same topic that they have selected initially, and that it would eventually show a problem from a different topic. “Skip this problem” only provides a problem from the same topic and does not automatically move on to a new topic. In addition, the option to pick a “Random problem” was not to clear for a few. One participant thought this meant that this would result in a random problem from a separate dataset, not included in the list of problems displayed.

The lowest rating was received from the students about the amount of feedback provided by the tutor (2.88). This was expected, since the prototype only provides simple feedback upon submission. Many participants suggested adding hints and more detailed feedback that describes the type of error they have on their solution i.e. syntax error. It is interesting that tutors scored the feedback significantly higher (4.2); ($U = 34$, $p = .045$).

Both groups seemed to perceive that the provided problems have the appropriate difficulty. The tutors were asked whether problems are at the appropriate level of difficulty for student learners (not for themselves). The average ratings for the average difficulty of the problems ranged from 4-4.5. As mentioned earlier in Section 4.3.1, the tutor contains puzzles in which the correct position of the LOCs does not depend on each other. However, two problems included in the pilot study were found to require this, which was not my intention. Some of the participants gave a correct solution, but since the application does not recognise this particular solution/s, this was regarded by the application as incorrect. However, this was easily spotted and can be easily fixed once these constraints are added and defined in the database. These instances were also eliminated in the findings.

Some of the puzzles include a def statement and a docstring. One participant commented that the docstring was missing the information about the return statement. This information can either be included in the docstring or purposely withheld to add an additional level of complexity to these problems. One of the participants appreciated the fact that most of the puzzles were only code snippets and therefore there was no need to think about placing a def statement and a docstring etc. Some participants also liked the fact that indentations were already included in puzzles. The indentations were hints to where a LOC need to be correctly positioned in relation to other LOCs in the puzzle. This was observed in some of the participants who were initially grouping LOCs with similar indentations before attempting to logically rearrange the LOCs. A participant commented: **“*Sometimes with the loops ... the indentations give away a lot and you can just you know*** (re-arrange them based on the indentations) ***without having to read much on what they mean.”***

Most participants commented that the problems seemed difficult at first, since they were different from what they were used to doing in their course i.e. code writing, but it became easier in the long run. There were only two participants who seemed to really struggle in answering the puzzles and did not even

recognise the basics i.e. def statements, docstrings, return statements and what the indentations indicate in relation to the LOCs included in the puzzle and its required output. Most of the student participants liked the idea of Parsons problems since they did not require them to write code from scratch. Some of them also appreciated the nature of the exercises since it was different than what they were used to which was usually writing code from null.

Figure 4.10 Example of a Parsons problem in Lists

There were some aspects of knowledge in certain puzzles which most of the participants never encountered before. This included the indexing of a list with 2 colons i.e. `my_list[start:end:step]` (fourth LOC in Figure 4.10). It seemed like they have only encountered ones with only 1 colon i.e. `my_list[1:3]`. One of the puzzles had an if statement condition that demonstrates an *exclusive or* conditional statement: `if (mood== “happy”) != (temperature > 12):` . Only one of them seem to

have understood this statement without any further explanation. Also, only one of the participants seemed to have never heard of *exclusive or*. A few people have forgotten what `enumerate` does and some people do not know what *mutable* or *immutable* types are. Most of them do not know what a `join()` does for lists and what a `zip()` does for tuples. Lack of knowledge in these might indicate that these were not covered in the course.

When asked whether they improved their skills by interacting with PyKinetic_Pilot, the average response from students was significantly higher than that of tutors (U = .5, p = .002). It is not surprising that the tutors' responses to this question are much lower, as the problems were designed for novices. A few people seemed surprised that they learned something from using the application and a participant remarked ***"I'm actually learning something here!"***. One participant seemed to be really learning and commented ***"Oh cool I didn't know you could do something like that."***. Another participant commented ***"I'm learning stuff while doing it so that's always a plus."*** and ***"I'm clearly learning something.."***. The latter statement was mentioned after successfully attempting a difficult problem and having finished the subsequent ones comparably faster.

Overall the participants enjoyed the variety and context of the problems in the system as well as the topics which are very much similar to the ones, they are used to in COSC121. A participant commented while solving a problem: ***"I think I did this with asterisks…"***. The level of difficulty of the problems also seemed to be similar to what they are used to in the course as most of them commented. This was manifested in the participants' answers in the questionnaires.

4.9 Strategies Observed from Students vs. Tutors

I watched the video recordings of the participants' interactions with PyKinetic_Pilot and manually observed and identified strategies made by the participants. A wide range of strategies was observed, some of which were used by participants in both groups. An example is to focus on a particular type of LOC and move it (referred to as **Selecting a LOC**). The participants usually looked for variable declarations, function calls and print statements, possibly because variable declarations and function calls are normally located somewhere at the beginning of a program, whilst print statements are normally positioned at the end. One participant made a comment along these lines: he/she just started a problem, noticed a print statement and mentioned the following while dragging the LOC in position: **"*Print statements at the end.*"** This strategy was used at least once by each student. One tutor used this strategy. However, it is important to note that this strategy was not used in all problems; it was observed that the participants' strategies changed depending on the nature of the problem and its expected output.

A more specific version of this strategy was used for problems with functions (**Selecting the**

Function Definition First), when the participants moved the function definition first, followed by the docstring. This strategy was very evident in both groups: five students and four tutors used this strategy for all problems that contained functions. The only situations when this strategy was not used were when those statements were already in place (please note that LOCs were presented in a random order), or when the participant was clearly missing the relevant declarative knowledge. The latter was observed only with students who used sub-optimal strategies (discussed in Section 4.9.1).

Another strategy was **Separating Distractors** which was used by both students and tutors was to move distractors (except the very obvious ones) to the end of the solution. Some of the comments that students made while employing this strategy were **"*just in case I still need it*" or "*I don't want to delete the other print lines yet just in case I do need them, but I'll put them down the bottom*."** Only two tutors used this strategy since tutors were generally better at eliminating distractors. Having said that, it seemed that the tutors were only doing so because they were too focused on building their solutions. Some preferred to deal with distractors immediately; therefore, there is no need to use this strategy.

All of these strategies require domain knowledge: knowing relative position for specific types of statements or being able to identify distractors. However, the majority of other observed strategies were used exclusively by one group of participants; those strategies clearly show the difference in domain knowledge between students and tutors. I present those strategies in the following subsections.

4.9.1 Strategies Observed in Students

A common strategy used by students was **Trial and Error**. After solving parts of the problem that the participant was knowledgeable about, the participant then tried to solve the rest of the problem by exploring possible solutions, which resulted in multiple submissions. For example, the participant would move a single LOC and submit the solution immediately, in order to eliminate wrong solutions. In some of the situations, the students asked the researcher for help. This strategy was used when the students were struggling with problems, therefore illustrating lack of knowledge. Additional evidence can be observed from their utterances, such as **"*I'm just gonna get to try all of them and figure out why*"** and **"*This is one of the questions that is probably more complex than my brain … whether or not I give up … I don't know*".** Three out of eight students used Trial and Error, and two other students commented that they could see that Trial and Error can be used as a strategy.

Half of the students used the **Solving by Indentations** strategy where LOCs were grouped on the basis of their indentations. Such a strategy shows lack of knowledge, as students were relying on a superficial feature rather than trying to understand the LOCs. The reliance on the indentations as scaffolding was also mentioned by a participant: **"*Sometimes with the loops … the indentations give away a lot and you can just you know … without having to read much on what they mean*."** This strategy allowed

students to eliminate distractors. The strategy was also useful for arranging the LOCs logically, especially with conditional statements. After applying this strategy, the students either tried to reason about the LOCs in each group or used the Trial and Error strategy. One participant mentioned **"*Okay let's put all the indentations at the same…*"** then tried to read the code, to find the correct lines. Another student mentioned: **"*So I'm like trying to find the systematic way of like sorting it.*"** Following this, the student also mentioned **"*So now I'm gonna work out which ones would make sense.*"**

One student used a unique strategy (**Starting with None**), when he/she deleted all the LOCs, and then retrieved the necessary ones from the trash. The participant eventually abandoned the problem, so this strategy might be due to high cognitive load.

4.9.2 Strategies Observed in Tutors

A common strategy used by tutors was to build the solution from top to bottom (referred to as the **Top-down** strategy). For example, some tutors mentioned that function statements have to be first, so they looked for this line and moved it first, then the docstring and other LOCs, until the return or print statement. This strategy shows that tutors have a mental model of the solution and were working towards matching it. All tutors used this strategy, but not always exclusively. One tutor alternated between this and **Syntactic and Logical** strategy, which consisted of combining syntactically and logically similar LOCs with similar indentations and then logically placing them in the correct order (e.g. similar print statements with similar indentations placed at the bottom).

While the tutors were searching for LOCs according to their model solution, three of them were at the same time deleting distractors which were syntactically incorrect lines of code (**Top-down while Eliminating Distractors**). The other two, on the contrary, left such LOCs and deleted them at the end, although it was clear they understood those LOCs were distractors (**Ignoring Distractors**). Generally, the tutors were good at identifying distractors.

4.10 Discussion and Conclusions

The initial system architecture, application architecture and UI design of the PyKinetic_Pilot were intertwiningly essential for providing a foundation for other versions of PyKinetic. My first research goal (Pilot_R1) was to evaluate the UI of PyKinetic_Pilot and most specifically gain insights on which orientation of the interface is more suitable for solving Parsons problems. Helpful suggestions were received in improving the UI. Overall, the UI was considered to be intuitive and user-friendly. As mentioned in Section 4.5, Parsons problems in PyKinetic_Pilot were presented in either portrait or landscape mode. It was observed that in problems presented in the landscape mode, most LOCs were obscured (participants

only see some of the LOCs), which seemed to increase extraneous cognitive load for many students. Some participants commented that the problems in the landscape mode seemed more difficult because the full view of the Parsons problem was not available.

Another research goal (Pilot_R2) was to identify differences between strategies used by students and tutors when solving Parsons problems with and without distractors. I have observed several effective problem-solving strategies used by both students and tutors, such as using declarative knowledge to focus on particular LOCs and position them first and moving distractors to the end of the code. The strategies used by tutors demonstrated a higher level of knowledge, as they mostly used the top-down strategy. One tutor used an optimal strategy of grouping LOCs with similar indentations, syntax and semantics then logically placing them in their respective positions. I have also observed several strategies when dealing with distractors. Tutors appeared to be better in identifying distractors compared to students.

I have identified six strategies used in solving (rearranging LOCs) for Parsons problems with distractors: **Selecting a LOC**, **Selecting the Function Definition First**, **Trial and Error**, **Solving by Indentations**, **Syntactic and Logical**, and **Top-down**. As mentioned in Section 4.5, my experiment design was similar with the study conducted by Ihantola and Karavirta (2011). Furthermore, the number of tutors in their study were similar to ours. Most tutors in my study followed the Top-down strategy, solving the problem from the function statement through to the return or print statement. Ihantola and Karavirta reported a similar Top-down strategy. However, they have not observed the tutors to move all lines perfectly (in the correct order). This is maybe because of the algorithmic nature of their problems compared to ours, which focused on honing basic Python programming skills for students. Nevertheless, I have confirmed their findings on the Top-down strategy observed in tutors. This shows that tutors solving Parsons problems are working towards a mental model solution.

There were similarities between the problem-solving strategies used by students and tutors. Both groups used two strategies in deciding the first LOC to focus on: Selecting a LOC and Selecting the Function Definition First. Both strategies exhibit some evidence that the learner has a mental model of the solution since he/she immediately knows the correct position of the selected LOC. There were also clear differences between strategies used by students and tutors. The students used sub-optimal strategies such as Trial and Error. None of the students used the Top-down strategy; this contradicts the findings reported by Helminen et al. (2012), where majority of the students were observed to follow the Top-down strategy. The reason for this may be the noticeable difference between the length and complexity of the problems used in their study (five problems with 3-8 LOCs without distractors), compared to my study involving 21 problems with 3-16 LOCs and 0-5 distractors per problem. Helminen et al. (2012) also focused on analysing only three out of five problems, which made their data set smaller. However, they have also observed a more specific strategy for Selecting a LOC which was to select a *for* loop or an *if* statement first (Helminen

et al., 2012).

Another strategy I have observed for students was Solving by Indentations, which is based on superficial scaffolding feature rather than on code logic and semantics. One tutor was observed to have used an optimal variation (Syntactic and Logical strategy). This tutor demonstrated a strategy of grouping syntactically similar statements with the same indentation while also positioning LOCs in place and removing distractors. Both groups were also observed to have strategies on dealing with distractors.

I have observed four strategies used in eliminating distractors: **Separating Distractors**, **Top-down while Eliminating Distractors**, **Ignoring Distractors**, and **Starting with None**. I do not have enough evidence whether one strategy used in eliminating distractors is more optimal than another. However, my suggestion is that the strategy used in eliminating distractors does not contribute to an optimal strategy; rather it is a coping mechanism. For example, the student who used Starting with None strategy was observed to have been overwhelmed with the problem and therefore tried to remove all LOCs first; and ended up abandoning the problem. A more thorough analysis of the data might shed more insight in the role of strategies used in eliminating distractors.

Overall, the pilot study was successful. The amount of enthusiasm, comments and suggestions (both from the video recording and questionnaire) from the participants was very encouraging. Seven out of eight students and two out five tutors agreed that they would use the tutor again. The latter response was expected with tutors since PyKinetic_Pilot is aimed for students. The findings from the study gave us more than enough insights to identify the best aspects of PyKinetic_Pilot and some areas for improvement. The pilot study already gave us encouragement to keep this research going and that students like them may be interested to use PyKinetic_Pilot. The consensus among participants was that they liked the idea of a different type of exercises to practice Python programming and the mobility of the tutor since it is developed on a mobile device. The participants gave very helpful feedback that will be considered for the development of the next version of the tutor, such as adding a short tutorial for first-time users, improved feedback and hints on solving problems.

There were some challenges and limitations to this study. Due to the experimental design, it was not possible to conduct parallel sessions as I was conducting my pilot study by myself. Therefore, conducting the pilot study took 13 hours. However, the one-on-one sessions provided rich observational data which proved vital for my later evaluation studies. A limitation of this study was the small number of participants. There were also buggy problems discovered in the study that have since been fixed. During this time, PyKinetic was still in the early ages of development and several evaluation studies were designed and conducted using later versions of the tutor (Chapters 5-7).

Due to the findings about problem-solving strategies in this study, it is possible to extend PyKinetic_Pilot to provide instruction about optimal problem-solving strategies. For example, the system

could offer instruction on specific topics the students are struggling with, or the system could refer the student to other potential sources. The system could also advise students about more effective problem-solving strategies, observed in tutors. I also plan to include support for self-explanation, an important meta-cognitive skill which improves learning outcomes, and also to introduce game elements to maximise engagement (Mayer and Johnson, 2010).

The next version of PyKinetic, covered in Chapter 5, contained a different variant of Parsons problems (Parsons problems with incomplete LOCs with additional menu-based self-explanation prompts). I have not since developed a version of PyKinetic with support for encouraging optimal problem-solving strategies and game elements. I decided not to go in that direction due to other findings reported in Chapters 5-7. However, implementing instruction for optimal problem-solving strategies and gamification would be invaluable to PyKinetic learners. But unfortunately, these features are out of scope within the context of my research.

5 FIRST EVALUATION STUDY

This chapter covers my first evaluation study and addresses research questions (R1) and (R3-R7). Some of the research questions above were not fully addressed in this chapter but were addressed collectively in all my evaluation studies. This chapter starts by presenting the two versions of PyKinetic (PyKinetic_IncLOCs and PyKinetic_IncLOCs_SE) that I implemented for the first evaluation study. Both versions included one type of activity: Parsons problems with incomplete LOCs. Furthermore, the two versions contained the same Parsons problems, but the difference is that PyKinetic_IncLOCs_SE offers a menu-based self-explanation (SE) prompt for each incomplete LOC. Following that, the architecture and development of PyKinetic_IncLOCs and PyKinetic_IncLOCs_SE are exhibited. Next, details about the evaluation study are presented: experimental design and findings. Finally, I present the discussion and conclusions.

5.1 PyKinetic (PyKinetic_IncLOCs and PyKinetic_IncLOCs_SE)

For my first evaluation study, I developed two new versions of PyKinetic (PyKinetic_IncLOCs and PyKinetic_IncLOCs_SE), containing a different variant of Parsons problems. Instead of using distractors, both versions contained Parsons problems with incomplete LOCs. Solving Parsons problems with incomplete LOCs is closer to code writing than Parsons problems with/without distractors (used in my pilot study in Chapter 4). Furthermore, a recent study found that Parsons problems with distractors decrease learning efficiency of middle-school children aged 10-15 (Harms, Chen, and Kelleher 2016). Both versions are identical in every way (i.e. design, control, and set of Parsons problems), with the additional menu-based SE prompts given in PyKinetic_IncLOCs_SE as the only difference.

In both versions, there were 15 Parsons problems, where the first two problems were used for practice. The remaining 13 problems had between 3 and 16 LOCs, with a maximum of 3 incomplete LOCs. There were six Python topics covered: string manipulation, conditional statements, while loops, for loops, lists, and tuples. Unlike PyKinetic_Pilot used for the pilot study (Chapter 4.2), learners do not have the freedom to choose a problem to solve. Instead, problems were given in a fixed order of increasing difficulty based on a combination of objective measures for difficulty presented in Section 3.13. A problem must be completed before proceeding to the next problem. The first half of the problems focused on a single topic, while the other problems covered at least two topics each.

Each problem consists of a problem description, expected output, and single LOCs for the Parsons problem. All blocks of code contain single LOCs, and correct indentation is provided for all LOCs (as scaffolding). The expected output is the anticipated displayed result when a Parsons problem is solved (all

incomplete LOCs correctly filled in and all LOCs reordered in the correct order). The initial seven problems were code snippets, while the remaining eight problems were functions with function calls. For all problems with functions, there was one function; and function calls are also required to be rearranged to match the problem description and the expected output.

An incomplete LOC contains a blank space, which may contain more than one keyword. For example, the third LOC of the problem shown in Figure 5.1 (middle) is: `____ person in people:`. The alternatives given to fill in the blank space were:

a) `for each`

b) `while`

c) `for`

In this case, the correct answer was c), the incomplete LOC only required one keyword, so the final answer was: `for person in people:`.

Another example of an incomplete LOC from a more difficult problem (not shown in Figure 5.1) was: `for num in range(_____)`*:* with the alternatives given:

a) `number-1, 0, -1`

b) `1, number`

c) `l, number, -1`

d) `number, 1`

Notice that the blank line is longer, indicating that more keywords are required. The correct answer for this incomplete LOC was a), so the completed LOC was `for num in range(number-1, 0, -1):`.

Four incomplete LOCs contained only a blank line each without any code provided; only the correct indentation was given as scaffolding. Seven incomplete LOCs had sets of three options each, while the other fifteen incomplete LOCs had a set of four options each. To solve problems which contain more than one incomplete LOC, learners should select an incomplete LOC one at a time by long tapping on the LOC. Then, the learner chooses the correct choice for the blank from a set of provided options, by tapping between alternatives instead of typing, like in (Ihantola, Helminen and Karavirta 2013). Learners submit their solution to an incomplete LOC by long tapping on the LOC.

There were two types of feedback when solving incomplete LOCs: highlighting the LOC to indicate the change in status and displaying feedback messages. Feedback on solving incomplete LOCs was only triggered by long tapping on the incomplete LOC either to select it or to submit an answer. PyKinetic highlights the selected LOC using three colours: turquoise, red, and green. Turquoise indicates that the learner has selected the incomplete LOC and intends to solve it. Red indicates that the learner's answer is incorrect. Lastly, green specifies that the learner has selected the correct choice for the LOC. When a learner submits an incorrect answer to an incomplete LOC, both PyKinetic_IncLOCs and PyKinetic_IncLOCs_SE

highlight the LOC red and show a feedback message: *"[alternative choice here] is incorrect."* If the learner successfully solves an incomplete LOC, it is highlighted in green. Furthermore, PyKinetic_IncLOCs displays a message *"Correct! Great job!"*. On the contrary, PyKinetic_IncLOCs_SE gives the SE prompt which corresponds to the incomplete LOC solved by the learner. In PyKinetic_IncLOCs_SE, an SE prompt was given every time the learner successfully solves an incomplete LOC. Therefore, if a problem contains three incomplete LOCs, the learner in turn receives three SE prompts. The SE prompt facilitates deeper learning by asking a question specific to the LOC completed.

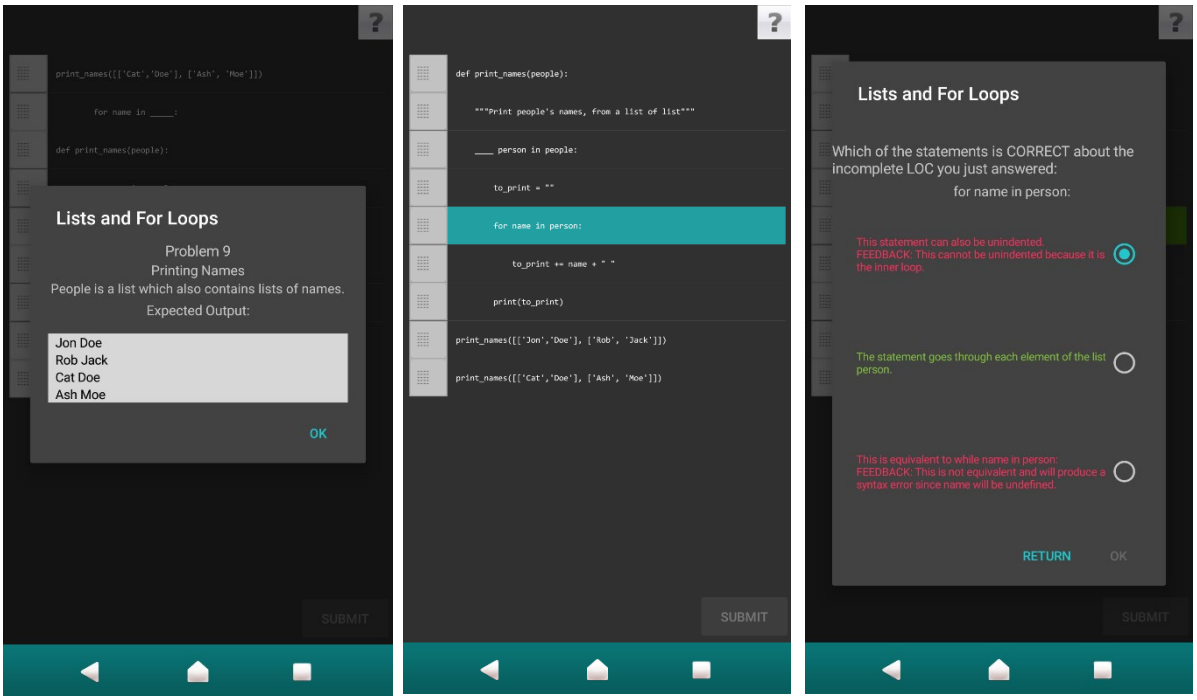


Figure 5.1 Problem description (left); a state of a Parsons Problem with two incomplete LOCs with one LOC which the learner is working on, highlighted in turquoise (middle); a conceptual SE prompt given for the completed LOC (right)

For both versions, all incomplete LOCs must be completed before the learner can submit their solution to the Parsons problem (by clicking on the Submit button). The Submit button was initially disabled and only activates when all incomplete LOCs were solved. Furthermore, this button was only used to put forward a solution for the entire Parsons problem (not for submitting an answer to an incomplete LOC). When reordering LOCs in the Parsons problem, feedback was only given upon clicking the Submit button. There were only two feedback messages given. For an incorrect solution, "Check the order of your solution." was shown. On the other hand, "Correct! Great job!" was given for a completed Parsons problem, same feedback given when an incomplete LOC was solved.

Table 5.1 First example of a conceptual SE prompt (shown in Figure 5.1, right screenshot)

Which of the statements is CORRECT about the incomplete LOC you just answered:
`for name in person:`

SE prompt options	Feedback received by learner for the corresponding SE option on the left column
The statement can also be unindented.	This cannot be unindented because it is the inner loop.
The statement goes through each element of the list person.	No feedback (only highlighted in green since this is the correct option)
This is equivalent to while name in person:	This is not equivalent and will produce a syntax error since name will be undefined.

As mentioned earlier, PyKinetic_IncLOCs and PyKinetic_IncLOCs_SE were identical apart from the additional SE prompts. In PyKinetic_IncLOCs_SE, SE prompts were provided after a learner finishes an incomplete LOC. The SE prompt facilitated deeper learning by asking a question specific to the LOC completed. There were 22 SE prompts in total; 14 prompts were conceptual questions and 8 were procedural questions. Conceptual questions covered declarative knowledge, assessing syntax and theoretical matters. On the other hand, procedural questions evaluated learners' understanding of code execution. Conceptual questions can be answered without necessarily reading the entire program, whereas procedural questions require the learner to mentally execute and trace the code. Half of the conceptual questions had three choices each, while the other half had four choices each. On the other hand, two of the procedural questions had three choices each, while the other six had four choices each. The SE prompts were related to the same topics covered by the Parsons problems. However, there were four additional topics covered: assignment statements, variables, print statements, and functions.

The conceptual SE prompt shown in Figure 5.1 (also presented in Table 5.1) was given after the learner worked on the incomplete LOC `for name in _____:`. The learner chose from the following alternatives:

a) `people`

b) `person`

c) `my_people`

The learner selected the correct option `for name in person:`. PyKinetic_IncLOCs_SE acknowledges the correct choice for the incomplete LOC by highlighting it in green. The learner next gets the SE prompt, which in this case was related to lists and for loops (Figure 5.1, right). In this case, the learner was asked to select all correct statements about the completed LOC, concerning those two Python topics. The learner cannot skip any SE prompts, and only had a single attempt on each prompt. In Figure 5.1 (right), the learner selected the first option. After the learner submits his/her answer for the SE prompt,

PyKinetic_IncLOCs_SE shows the correct options in green and incorrect in red and provides additional feedback for incorrect choices.

The example in Table 5.1 targets the for-loop statement completed by the learner. The SE prompt was considered as conceptual because it was testing the learner on declarative knowledge about for loops. Specifically, it was asking the learner about the role of indentations in a for loop and its mechanism. Also, it was testing the learner on syntax, and explicitly whether the exact same code would work if replaced with a while loop.

Another example of a conceptual SE prompt is shown in Table 5.2. This example assesses the learner's knowledge about the syntax of assignment statements, and about string variables. The learner worked on the incomplete LOC `message = _____`. Then, the learner successfully selected the correct option `"says hello"`. One thing to note is that the example in Table 5.2 presents a negatively phrased question as opposed to the example in Table 5.1. Therefore, the students need to identify the incorrect statements related to the LOC `message = "says hello"`.

Table 5.2 Second example of a conceptual SE prompt

Select all INCORRECT statements about the incomplete LOC you just answered:
`message = "says hello"`

SE prompt options	**Feedback received by learner for the corresponding SE option on the left column**
The variable message has a string value	says hello is in enclosed with quotation marks which makes it a string
`message = says hello` is equivalent to this statement	No feedback (only highlighted in green since this is one of the correct options)
`"says hello" = message` is also equivalent to this statement	No feedback (only highlighted in green since this is one of the correct options)

A third example of a conceptual prompt is given in Table 5.3. The learner was working on a more difficult problem with two incomplete LOCs (Figure 5.2, middle). This problem contained 16 LOCs; note that two LOCs are not visible in the screenshot shown in Figure 5.2 (middle), as the learner needed to scroll through to see them. As observed, the learner was still in the process of rearranging the LOCs. The goal was to rearrange the code and complete the missing keywords to complete a function magic which takes two lists containing integers as parameters. Function magic is expected to return -1 when the given lists are equal in length. Otherwise, the function will check if both lists start and end with the same integer. If so, the function returns the sum of all integers in the first list. But when the lists do not contain the same first and last integer, it will do "magic". "Magic" returns the product of all integers in the second list.

Table 5.3 Third example of a conceptual SE prompt (same as shown in Figure 5.2)

In checking for equality in Python using == select all that apply:
`if len(list) == len(list2):`

SE prompt options	Feedback received by learner for the corresponding SE option on the left column
= can also be used similar to maths	= is for assignment statements
`s1 == s2` is equivalent to `s2 == s1`	No feedback (only highlighted in green since this is one of the correct options)
It can also be used in for loops i.e. `for s1 == s2:`	for loops are mostly used in iterating through lists
It can also be used in while loops i.e. `while s1 == s2:`	No feedback (only highlighted in green since this is one of the correct options)

I first focus on the incomplete LOC selected by the learner (highlighted in turquoise). The learner successfully completes this LOC and is presented by the SE prompt (Figure 5.2, right; also shown in Table 5.3). Compared to the other examples presented so far (Tables 5.1 and 5.2), this question poses a more directly phrased approach in probing the learner about the incomplete LOC answered. It is specifically assessing the learner on the equality operator "==", which was the missing keyword. Based on my experience in teaching Python, learners often have a misconception that "=" can be used interchangeably with "==".

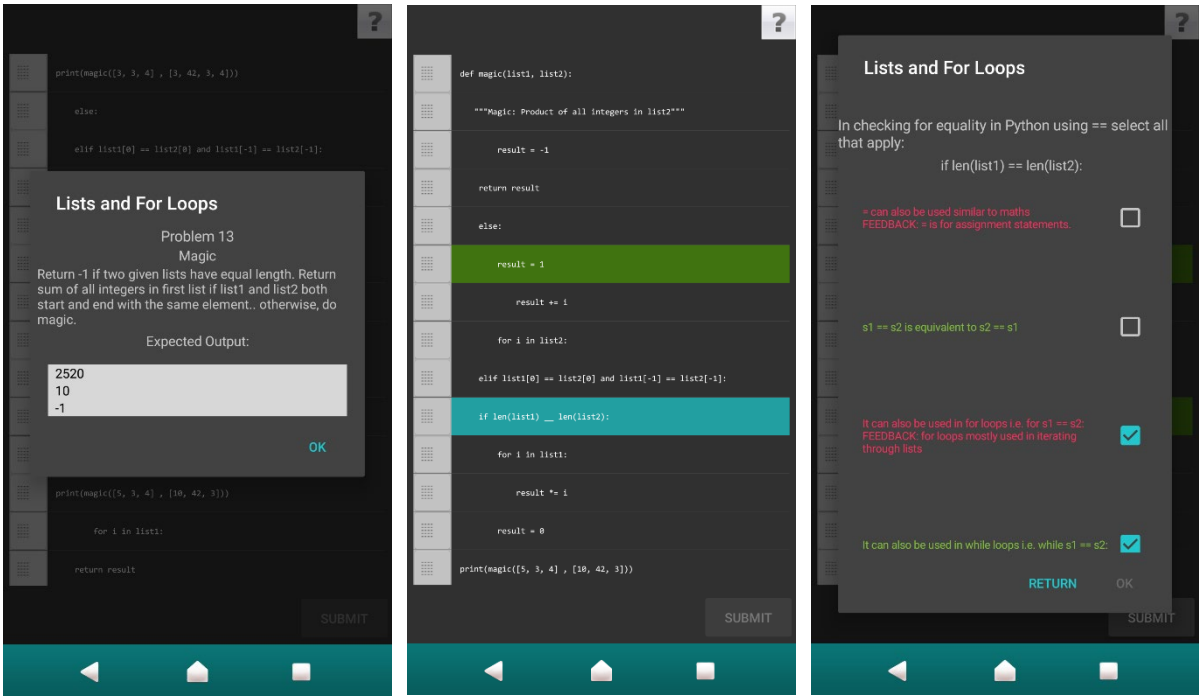


Figure 5.2 Problem description (left); Parsons Problem with two incomplete LOCs, one completed highlighted in green and second selected by the learner and still incomplete highlighted in turquoise (middle); a conceptual SE prompt given when turquoise coloured LOC is completed (right)

PyKinetic_IncLOCs_SE also offered procedural SE prompts, like the one illustrated in Table 5.4. The LOC highlighted in green in Figure 5.2 (middle) corresponds to a procedural question. This LOC (result = 1) started as a blank keyword (_____) with indentation provided. The learner chose between the following alternatives:

a) `result = i`

b) `result = list2[i]`

c) `result = 1`

d) `return result = 0`

As displayed in Figure 5.2 (middle), the learner successfully chosen the correct answer c) `result = 1`. Upon completion, PyKinetic_IncLOCs_SE gave the learner the SE prompt shown in Table 5.4. Like the question presented in Table 5.3, the question was also explicitly phrased to target the incomplete LOC. The question in Table 5.4 was procedural because to answer the question, the learner must think about how the code would be executed and understand why this was needed within the context of the expected output. The content of the question in Table 5.4 aimed to support the reasoning of the learner in understanding the role of this LOC within the entire program.

Table 5.4 First example of a procedural SE prompt

Why was this line necessary with a value of 1? Select all that apply:
`result = 1`

SE prompt options	**Feedback received by learner for the corresponding SE option on the left column**
Magic will work well without this line	This is needed to initialise result
`result` needs to be initialised before assignment inside the for loop.	No feedback (only highlighted in green since this one of the correct options)
Initial value of 1 needed for multiplication	No feedback (only highlighted in green since this one of the correct options)
Value 1 indicates first element in `list2`	Value 1 here indicates an integer number 1

Figure 5.3 shows a relatively easier problem on conditional statements, which contained one incomplete LOC (highlighted in red in Figure 5.3, middle). In this example, the learner had correctly rearranged the LOCs in the problem but incorrectly answered the incomplete LOC, thus PyKinetic_IncLOCs_SE highlighted it in red. The first two test cases were ordered correctly and displays “Go fishing” and “Stay at home” respectively. In this example, the next goal was to complete the last test case to produce an expected output of “Fly overseas”. The options provided for the incomplete LOC were:

a) “clear”, 2500

b) “stormy”, 198

c) “windy”, 1000

d) “sunny”, 1998

This problem contained two other function calls (i.e. the two LOCs above the highlighted LOC). These function calls output “Go fishing” and “Stay at home” respectively, and the learner placed them in the correct order. The learner next must complete the function call to output “Fly overseas”. In this example, the correct answer was “clear”, 2500 and the completed LOC is print_activity(“clear”, 2500), shown in (Figure 5.3, right). After the learner solved the incomplete LOC, the SE prompt was shown. In this case, another procedural SE prompt was given, asking which of the provided statements can replace the incomplete LOC (Figure 5.3, right; Table 5.5). This was a procedural prompt because it specifically asks the learner to supply an alternative test case that would produce the same expected output which was “Fly overseas”. To be successful in solving the question, one must mentally execute the code to find out what works for the program. Therefore, relying solely on declarative knowledge about function calls will not be adequate. The learner successfully identified the two correct options for the SE prompt; but, was still incorrect as he/she also selected one of the wrong options.

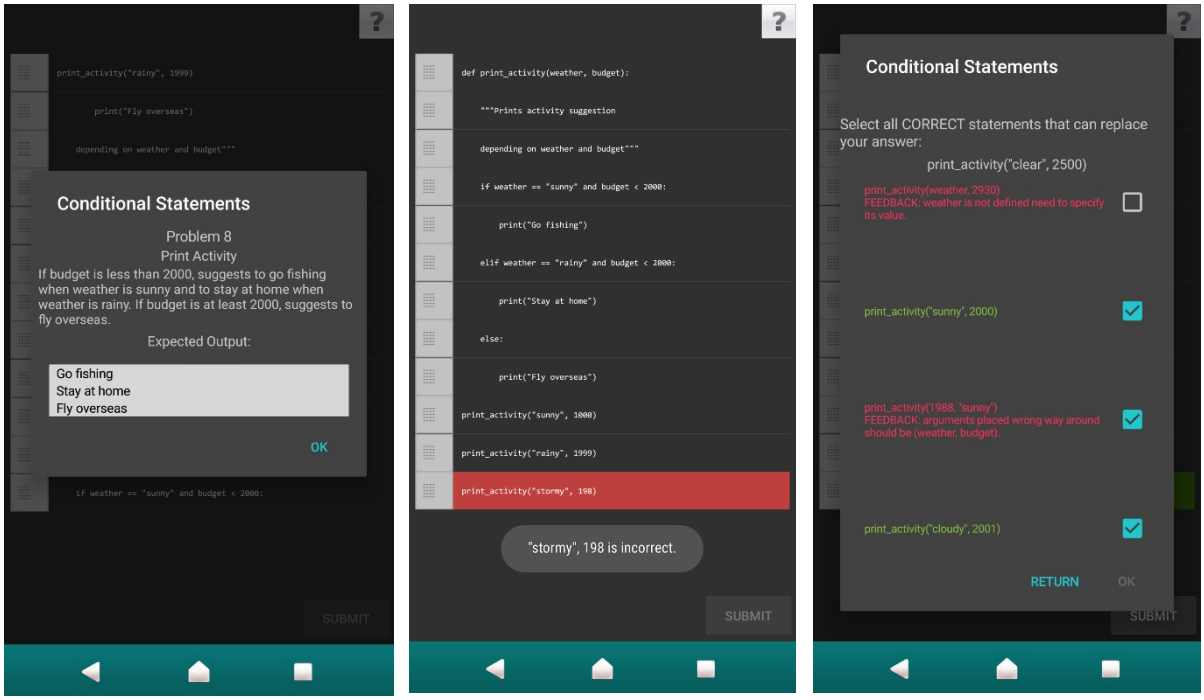


Figure 5.3 Problem description (left); Parsons Problem with one incomplete LOCs incorrectly answered by the learner, and therefore highlighted in red, with simple feedback shown below (middle); a procedural SE prompt given when highlighted LOC is completed (right)

Table 5.5 Second example of a procedural SE prompt (as shown in Figure 5.3)

Select all CORRECT statements that can replace your answer:
`print_activity("clear", 2500)`

SE prompt options	Feedback received by learner for the corresponding SE option on the left column
`print_activity(weather, 2930)`	`weather` is not defined need to specify its value
`print_activity("sunny", 2000)`	No feedback (only highlighted in green since this one of the correct options)
`print_activity(1988, "sunny")`	Arguments placed wrong way around should be `(weather, budget)`
`print_activity("cloudy", 2001)`	No feedback (only highlighted in green since this one of the correct options)

Another example, displayed in Figure 5.4, comes from the second most difficult Parsons problem in both versions of PyKinetic. As portrayed in Figure 5.4, the learner had not finished rearranging the LOCs. In addition, notice that this problem contains three incomplete LOCs. The learner was working on one of them (highlighted in turquoise) which initially was `mix_list._____`. The learner eventually submits the correct answer and was given a procedural SE prompt (Figure 5.4, right; Table 5.6). The given SE prompt was procedural because the first two SE options required the learner to trace the code and figure out the values of the variable `code` and `name` which were part of a tuple `student` which was `(name, code, number)`. Furthermore, the third option in the SE prompt also required procedural thinking as it was asking whether the code can be replaced by another code – `append(student[0])`.

Table 5.6 Third example of a procedural SE prompt

Which of the following statements is CORRECT about the incomplete LOC you just answered:
`mix_list.append(code)`

SE prompt options	Feedback received by learner for the corresponding SE option on the left column
`code` represents names (in student list)	No, it represents codes.
`code` represents all codes (in student list)	No feedback (only highlighted in green since this is the correct option)
`append(student[0])` would've given the same results	Index 0 represents names
`name` represents all numbers (in student list)	`name` represents each name

All SE prompts showed the LOC correctly answered by the learner. Most SE prompts (68%) have multiple correct choices provided, while the others only have a single correct choice (as in Figure 5.4, right). For questions with multiple correct choices, learners were required to identify all correct choices. Furthermore, the SE prompts were phrased in three ways. Some SE prompts were phrased in the positive manner, asking the student to select correct statement(s). Other SE prompts were phrased negatively,

requiring the student to select all options which are incorrect. Lastly, the third form were more directly phrased, as in this example: “In checking for equality in Python using == select all that apply:” and “Why was this line necessary with a value of 1? Select all that apply:”. Feedback *“Correct! Great job!”* is displayed when an SE prompt is answered correctly. If the learner’s response was incorrect, an explanation was shown on all wrong options (Figure 5.4, right). Regardless of the solution, when the learner submits their answer, all wrong options in the SE prompt were shown in red font colour, while the correct options were shown in green font colour (Figure 5.4, right).

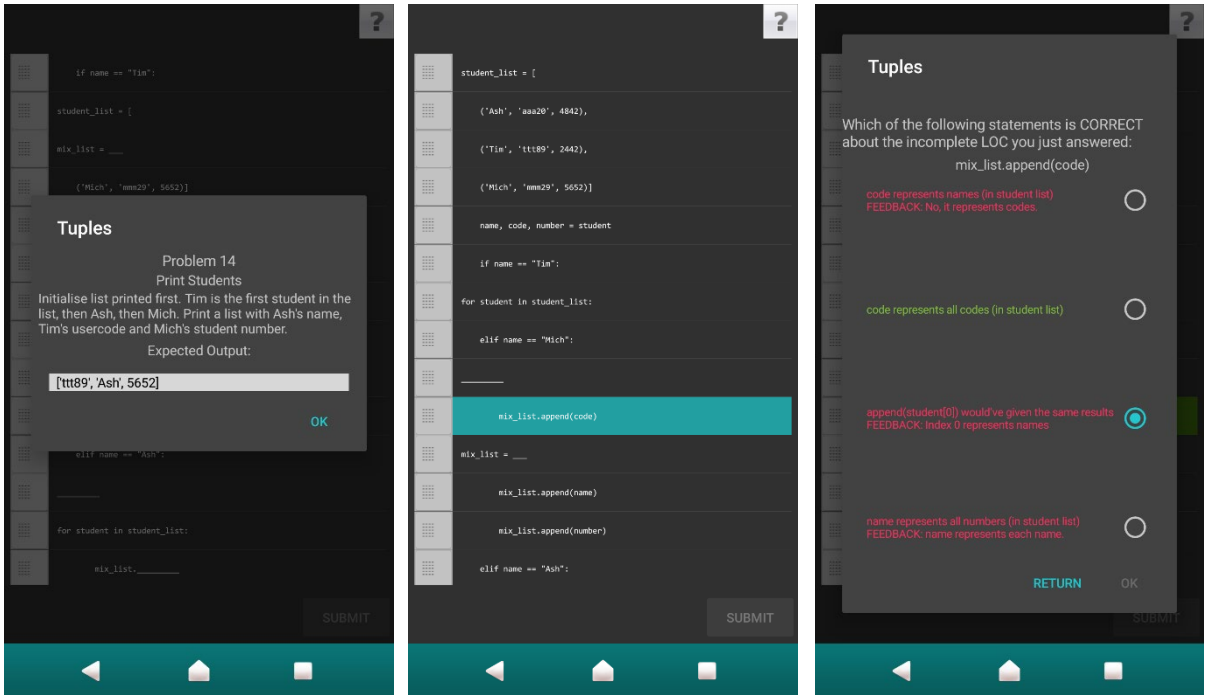


Figure 5.4 Problem description (left); Parsons Problem with three incomplete LOCs, one LOC where the learner is working on, so it is highlighted in turquoise (middle); a conceptual SE prompt given when highlighted LOC is completed (right)

5.2 Architecture and Development

The general architecture of the application in both versions are described in Chapter 3.4.1. However, specific characteristics of PyKinetic_IncLOCs and PyKinetic_IncLOCs_SE are presented here. For PyKinetic_IncLOCs the code statistics are reported in Table 5.7. In comparison to PyKinetic_IncLOCs_SE (Table 5.8), the size of the code in PyKinetic_IncLOCs was less as expected because of the additional SE prompts in PyKinetic_IncLOCs_SE. Only the code that I have written is represented in both Tables 5.7-5.8, other files generated upon compilation, building, and executing both applications are not represented. Furthermore, third-party libraries used in both versions are not included in the statistics shown in the tables below.

Table 5.7 Code Statistics for PyKinetic_IncLOCs (version without SE)

	Java	**XML**
Classes/Files	17	6
Total LOCs without blank lines	2290	225
Source code lines	1836	225
Comment lines	454	0

Table 5.8 Code Statistics for PyKinetic_IncLOCs_SE (version with SE)

	Java	**XML**
Classes/Files	21	7
Total LOCs without blank lines	2897	235
Source code lines	2378	235
Comment lines	519	0

With regards to the manifest XML file in both versions, there were two Android permissions: internet access and access to the state of the network connection. Both the permissions were necessary for recording logs (in real time) during the evaluation study; where the logs contained user actions. The first two permissions regarding network connectivity was necessary for sending logs to a server.

5.2.1 Storage

Both PyKinetic_IncLOCs and PyKinetic_IncLOCs_SE had the same system architecture shown in Figure 5.5. All problems and its details for both versions of the tutor were stored in the SQLite database like in the PyKinetic_Pilot (evaluated in the pilot study). Furthermore, detailed feedback given in the SE prompts for PyKinetic_IncLOCs_SE were also stored in the database. However, simple feedback and other texts used throughout both versions of the tutor were stored in the application in the strings XML file. Unlike in the pilot study, it was necessary for my first evaluation to be connected to a server to send logs. These logs were simply stored as text files. Furthermore, a simple php admin page was made to monitor the progress of the participants during the evaluation study in real time, via the logs received in the server.

The ER diagram of the database (without the attributes) used in PyKinetic_IncLOCs and PyKinetic_IncLOCs_SE are shown in Figures 5.6 and 5.7 respectively. As expected, both versions have similar database structure apart from storing SE prompts for PyKinetic_IncLOCs_SE. For both versions, problem details, expected output, and LOCs were stored similarly as the prototype of PyKinetic used in the pilot study (Chapter 4.3.1). However, in comparison to the pilot study, both versions did not require a table Category for storing Python topics because in the evaluation study in this chapter, I did not intend for learners to choose problems from topics.

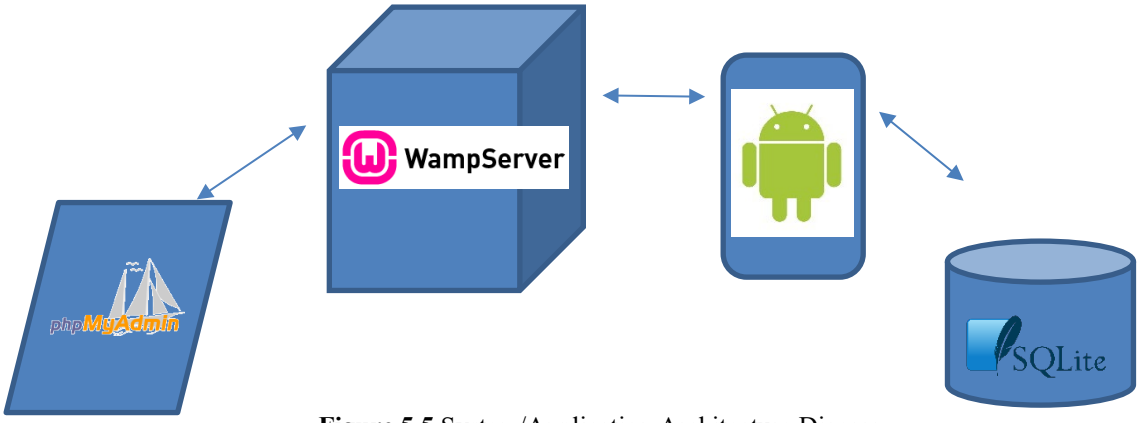


Figure 5.5 System/Application Architecture Diagram

Since both PyKinetic_IncLOCs and PyKinetic_IncLOCs_SE contain Parsons problems, the correct positions for each LOC had to be stored in the database. I stored this similarly as implemented in the pilot study (Chapter 4.3.1), where the correct positions are stored as integers. In the context of storing problems, the difference between Parsons problems with incomplete LOCs and Parsons problems with distractors are the alternatives given specifically for incomplete LOCs. Therefore, I added a table CHOICE_LOC (Figures 5.6-5.7) in the databases of PyKinetic_IncLOCs and PyKinetic_IncLOCs_SE to store choices that will fill in the missing keywords in an incomplete LOC. In a problem, only few LOCs are incomplete, so choices are not strictly required for all LOCs.

For the SE prompts in PyKinetic_IncLOCs_SE, the details including the question in the SE prompt is stored in the SE table (Figure 5.7). All choices for the SE prompt are stored in choices_se including the detailed feedback for the choices. Only the incorrect choices contained detailed feedback.

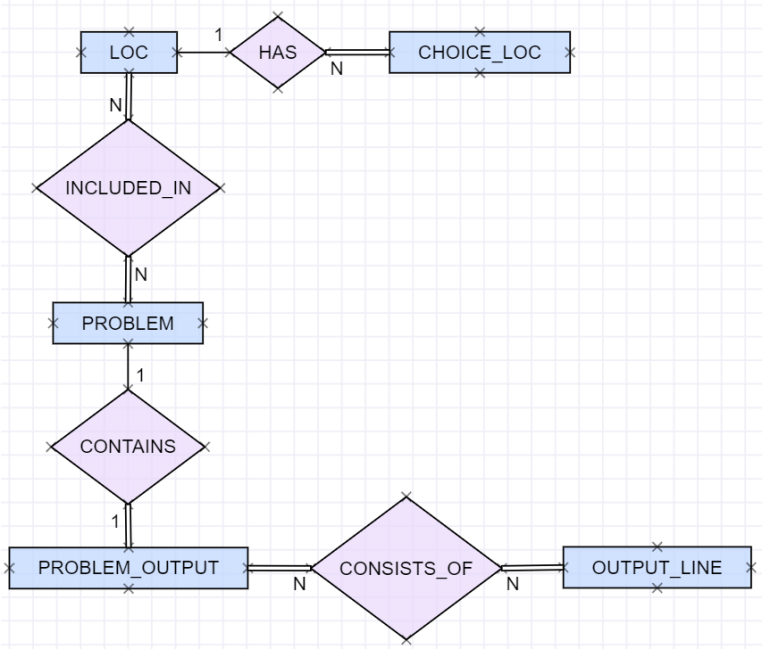


Figure 5.6 ER diagram of the database in PyKinetic_IncLOCs without the attributes

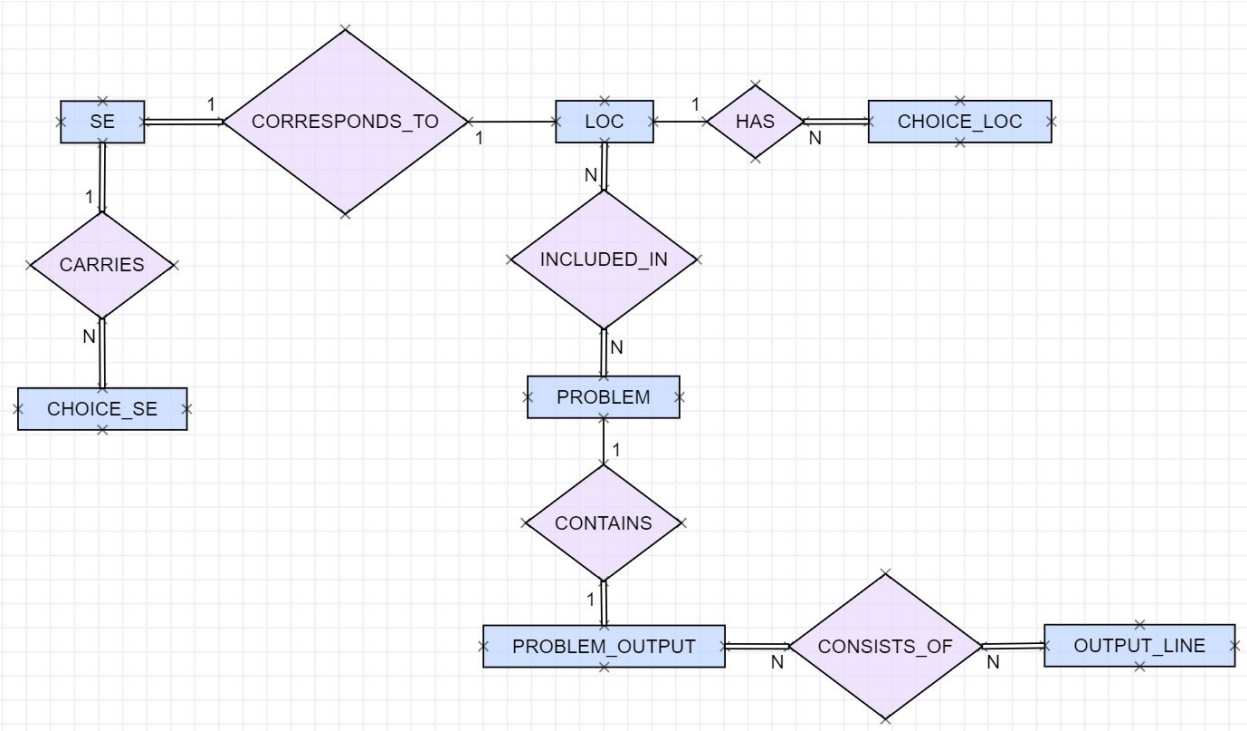


Figure 5.7 ER diagram of the database in PyKinetic_IncLOCs_SE without the attributes

5.2.2 Interaction Elements

The list of navigational elements used in PyKinetic_IncLOCs and PyKinetic_IncLOCs_SE are presented in Table 5.9. PyKinetic_IncLOCs_SE had additional elements for the SE prompts, and these are presented in Table 5.10. All elements from Tables 5.9-5.10 require either single tap or long tap actions.

Table 5.9 List of buttons, icons, and navigational elements for both PyKinetic_IncLOCs and PyKinetic_IncLOCs_SE

Screen/Dialog	Element	Purpose
Start screen	Edit text for participant id	Element where users enter in their participant id for evaluation purposes
	Start button	Navigates to the first problem offered in the tutor
	Device's back button	"Closes" the app (not completely, it will still run in the background)
Parsons problem solving screen	"?" button	Displays the problem details dialog box
	Back button	Screen stays in place and "Sorry, you can't go back to the previous problem." is displayed below the screen.
	Drag icon	By holding on the drag icon and moving it vertically, it moves a LOC from the position it was dragged from to a new position where it was dropped (upon release of the drag icon).
	Incomplete LOC	Long tapping will select the LOC to indicate that the learner wants to solve this LOC. Tapping on a selected incomplete LOC would change the LOC and show the alternative keyword/s to fill the blank. Other notes: An incomplete LOC needs to be selected first before being able to see alternatives. Any changes to an incomplete LOC will undo when LOC is moved or unselected unless the answer is finalised by long tapping on the LOC (like when selecting).
	SUBMIT button	Initially disabled, until all incomplete LOCs are correctly solved. Gives feedback about the Parsons problem.
Problem details dialog box	Ok button	Closes the dialog box
Feedback dialog box	Ok button	Closes the dialog box

Table 5.10 Additional list of buttons and navigational elements for PyKinetic_IncLOCs_SE

Screen/Dialog	Element	Purpose
SE prompt dialog box	Back button	Disabled
	Output choice	Tapping on a choice indicates that the learner has selected this as his/her solution and at the same time SUBMIT button is triggered. Other choices selected before are unselected at the same time.
	OK button	Initially disabled, enables when at least one choice is selected; used by the learner to submit their solution to the SE prompt.
	RETURN button	Only enables after the learner submits their solution, dismisses the dialog box.

5.3 Experimental Design

I conducted a controlled lab study with PyKinetic_IncLOCs and PyKinetic_IncLOCs_SE. There were two conditions in the study; the only difference between the versions presented to the control (PyKinetic_IncLOCs) and experimental group was that the experimental group used PyKinetic_IncLOCs_SE which had additional menu-based SE prompts for every LOC completed. The conditions in which differed by whether SE were provided to participants was similar to other studies (Rau, Aleven, and Rummel, 2015; O'Neil et al., 2014; Hsu, Tsai, and Wang, 2012; Chamberland et al., 2011; Rittle-Johnson, 2006; Aleven and Koedinger, 2002). My hypotheses were:

(H1) Both PyKinetic_IncLOCs and PyKinetic_IncLOCs_SE would be successful in supporting learning

(H2) menu-based SE prompts would result in further learning benefits

(H3) students with low prior knowledge (LP) would learn more from my variant of Parsons problems in comparison to those with high prior knowledge (HP).

Parsons problems are designed for introductory programming students which are my target audience. Literature revealed that Parsons problems are similar to solving code writing exercises (Denny et al., 2008; Ericson, Margulieux, and Rick, 2017; Ericson, Foley, and Rick, 2018). Enhancing code writing skills is essential for learning programming, coupled with incomplete LOCs, I expected that (H1) both PyKinetic_IncLOCs and PyKinetic_IncLOCs_SE would successfully support learning. For (H2), I expected that menu-based SE prompts would result in further learning benefits because SE is known to promote deeper learning (Chi 2000). Lastly, (H3), I hypothesised that LP would learn more from my variant of Parsons problems since these exercises are low-cognitive load exercises. Also, my variant includes indentations as scaffolding to help LP students. Lastly, I expect that most HP students are generally good in code writing, and therefore would probably not learn as much in rearranging LOCs for learning.

5.3.1 Participants

I recruited 47 volunteers from an introductory programming course at University of Canterbury (UC), and 23 volunteers enrolled in an introductory computing course at the Ateneo de Manila University (ADMU). There were 70 participants in total, randomly assigned to two groups: experimental group (with SE prompts) and control group (without SE prompts). The ADMU participants were given free food as compensation. The participants from the University of Canterbury did not receive compensation but were added to a draw for one of four NZ$50 vouchers. The study was approved by the Human Ethics Committees of both universities (Appendix F).

5.3.2 Method and Materials

Each participant joined a group session, which lasted for 1.5-2 hours. There were between one and 13 participants per session. At the start of each session, the participants were introduced to the study and provided informed consent. The participants were advised that they could pause or stop at any time during the study. Then, a 15-minute pre-test was administered, and the participants were instructed on how to download and install the tutor. The participants from both groups (depending on the condition) used either PyKinetic_IncLOCs or PyKinetic_IncLOCs_SE for about an hour. The design and name of the application installed for both groups were labelled similarly to avoid any suspicion or bias. After interacting with the tutor, the participants received a 15-minute post-test, and finally, instructions on uninstalling the application. Both pre- and post-tests were completed on paper. Some participants used their own Android smartphones, while I provided phones to other participants. After the post-test, I asked participants who used their own devices for the study to uninstall the application.

I wrote two tests of comparable complexity (Appendix A). The pre/post-test had a total of eight questions: six conceptual questions (with a maximum of 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 (without providing any choices), 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 sequential order. One participant wrote a slightly different solution by modifying the incomplete LOCs but received full marks on that question as the solution was correct. There were more conceptual than procedural questions in the pre/post-tests, which was similar proportionally with SE prompts.

5.3.3 Data Collection

For both versions of the tutor, logs were recorded for all activities. Every user action (button press, single tap, and long tap) triggers additional data to the log: a timestamp with details about the interaction. Feedback received by the student based on their last solution was also logged, which was useful for identifying how close a student was to completing a problem where they failed to do so. Lastly, the total time taken per problem was also recorded. For the SE prompts in PyKinetic_IncLOCs_SE, the time spent solely on these were also stored.

Evaluation logs were only sent and received through the server. Both PyKinetic_IncLOCs and PyKinetic_IncLOCs_SE did not store logs in the devices because majority of the participants used their own phones for the evaluation. Therefore, I relied heavily in the network as I only received data through the logs sent from the devices via WiFi.

The logs were gathered while the user was using the tutor. A log file was sent to the server when: a learner submits a solution, navigates away from an application screen, and every three minutes. I recorded the logs every three minutes to have a reasonable amount of data upon recording the log, and at the same time circumvent data loss.

5.4 Findings

I have eliminated data about seven participants due to incomplete logs caused by WiFi connection problems. The chapter presents analyses performed on the data collected from the remaining 63 participants. As I had participants from two universities, I compared their pre-test scores before performing further analyses. There was no significant difference on the pre-test scores of the two populations (Table 5.11). Therefore, the two populations had comparable levels of pre-existing knowledge. Figure 5.8 shows the box plots of pre-test scores from both universities.

Table 5.11 Mean pre-test scores (standard deviations in parentheses) of the two populations of participants

	UC (42 students)	**ADMU (21 students)**
Pre-test %	66.73 (13.05)	63.40 (13.53)

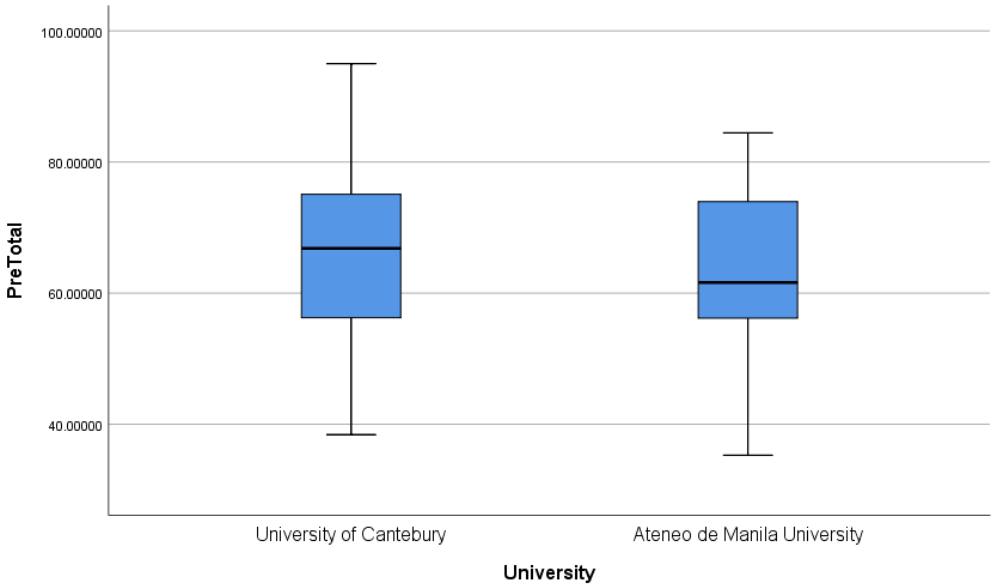


Figure 5.8 Box plots for pre-test total scores of the two populations of participants

As the data was not normally distributed, I used non-parametric tests for all reported analyses. I calculated the normalised gain using two formulas like Marx and Cummings (2007). When the learning gain was positive, I calculated the quotient of gain (post-test score – pre-test score) and (100 – pre-test score). However, when the learning gain was negative, I calculated the quotient of gain and the pre-test score.

On average, the experimental group participants spent 48 minutes using PyKinetic_IncLOCs_SE (sd = 13.1), while control group learners spent slightly less time with PyKinetic_IncLOCs (42 minutes, sd = 14.4). There was no significant difference on the total interaction time between the two conditions. However, there was a significant difference (U = 340, p = .035) between the two conditions on average time spent per problem, with the control group participants being faster (mean = 3.22 minutes, sd = 1.18) compared to the experimental group (mean = 4 minutes, sd = 1.52). This was expected, as they did not have SE prompts. Only 38 participants finished all problems (11 from experimental and 27 from the control group). Out of fifteen problems, the experimental group participants completed 13.2 problems (sd= 1.9), which was significantly less (U = 701.5, p =.001) than the control group (14.4 problems, sd=1.5).

I used the paired Wilcoxon Signed Ranks test to examine hypothesis H1 both PyKinetic_IncLOCs and PyKinetic_IncLOCs_SE would be successful in supporting learning. Table 5.12 reports the pre/post-test scores for the two groups on all questions, and separately on conceptual/procedural questions. Participants from both groups improved their scores significantly between the pre- and post-test (the Improvement on All Questions row), and on conceptual questions (the Improvement Conceptual row), but there was no significant improvement on procedural questions only. These results provide evidence to accept my first hypothesis H1. The fact that the two groups improved on conceptual questions but not on

procedural questions might be explained by the higher proportion of conceptual vs. procedural in the SE prompts. However, further investigation is needed to prove this speculation. Both groups had a positive Cohen's d effect size, but the effect size was higher for the experimental group.

Table 5.12 Pre- and post-test scores in % (* denotes significance at $p < .05$)

	Experimental (29) Mean (sd)	**Control (34)** Mean (sd)	**U, p**
Pre-test	65.22 (15.31)	65.96 (11.33)	ns
Post-test	77.91 (13.26)	72.60 (12.88)	ns
Improvement on All Questions	W = 379, p = .000*	W = 483, p = .002*	
Cohen's d for All Questions	d = .89	d = .55	
Normalised Gain All Questions	34.73 (32.34)	20.42 (25.28)	U = 348, p = .046*
Pre-test Conceptual	62.19 (16.42)	63.24 (13.50)	ns
Post-test Conceptual	78.06 (15.41)	71.19 (15.15)	ns
Improvement Conceptual	W = 375, p = .001*	W = 474.5, p = .002*	
Cohen's d for Conceptual	d = 1.00	d = 0.55	
Normalised Gain Conceptual	41.61 (38.84)	22.08 (30.94)	U = 334, p = .028*
Pre-test Procedural	74.31 (23.36)	74.99 (19.79)	ns
Post-test Procedural	77.45 (16.46)	76.84 (19.50)	ns
Improvement Procedural	ns	ns	
Cohen's d for Procedural	d = 0.16	d = 0.09	
Normalised Gain Procedural	21.99 (47.34)	29.54 (48.15)	ns

To answer H2 (menu-based SE prompts would result in further learning benefits**)**, I used Mann Whitney U test for checking significant differences between the groups (Table 5.12). There was no significant difference on the pre-test scores. The experimental group had significantly higher normalised gains for all questions (U = 348, p = .046). Furthermore, the normalised gain for conceptual questions of those who self-explained was also significantly higher (U = 334, p = .028). However, both groups had comparable normalised gains for procedural questions. I also calculated the Cohen's d effect size of each group. The experimental group had almost double effect sizes compared to the control group on all questions, conceptual, and procedural questions. The Cohen's d effect size for the normalised gains between the groups is 0.493, indicating a moderate positive effect. The Cohen's d effect size for the normalised gains on conceptual questions also indicates a moderate positive effect (d = 0.556). On the other hand, for procedural questions between groups the effect size is a weak negative effect (d = -0.158). These findings were enough evidence to support H2, revealing that participants who self-explained had further learning benefits.

Table 5.13 Pre- and post-test scores (%) for Easier and Harder to Guess questions (* denotes significance at $p < .05$)

	Experimental (29) Mean (sd)	**Control (34)** Mean (sd)	**U, p**
Pre-test Easier-to-Guess	75.57 (23.14)	80.39 (21.89)	ns
Post-test Easier-to-Guess	89.37 (20.03)	87.99 (21.44)	ns
Improvement on Easier-to-Guess	ns	ns	
Cohen's d on Easier-to-Guess	d = 0.64	d = 0.35	
Normalised Gain Easier-to-Guess	42.53 (64.29)	24.02 (55.69)	ns
Pre-test Harder-to-Guess	59.01 (14.00)	57.30 (10.86)	ns
Post-test Harder-to-Guess	71.03 (14.95)	63.37 (12.58)	U = 319, p = .016*
Improvement on Harder-to-Guess	W = 368, p = .001*	W = 472, p = .003*	
Cohen's d on Harder-to-Guess	d = 0.83	d = 0.52	
Normalised Gain Harder-to-Guess	12.02 (46.96)	22.69 (33.08)	ns

To probe further into the effect of SE prompts on deeper learning, I performed a similar analysis as in (Aleven and Koedinger 2002), who also conducted an evaluation with two versions of their system (with and without SE). I classified pre/post test questions into two categories: easier-to-guess and harder-to-guess. The easier-to-guess questions were two True/False questions, and one multiple choice question with a single answer required (3 marks in total). The harder-to-guess questions (5 marks in total) required more knowledge to identify the correct answers. For example, the harder-to-guess questions included an output prediction question, which required the student to analyse the given code and think about its output, as there were no options provided. There were also three multiple-choice questions where the student needed to identify all correct options, and a Parsons problem. I used the Wilcoxon Signed Ranks test and Mann Whitney U test for checking significant differences between the groups. The results in Table 5.13 show that both groups improved significantly on harder-to-guess questions from pre- to post-test. The experimental group outperformed the control group on the harder-to-guess questions. I also calculated the effect sizes for easier and harder-to-guess questions (Table 5.13). The Cohen's d effect size for the normalised gains on easier-to-guess questions between groups reveals a weak positive effect (d = 0.308). For harder-to-guess questions, the between groups Cohen's d effect size is a weak negative effect (d = -0.263).

To investigate H3 (whether Parsons problems with incomplete LOCs are more beneficial for LP learners, we performed a median split on the pre-test scores of the participants like (Adams et al., 2014; McLaren, Adams, and Mayer, 2015). The participants with pre-test scores less than the median (66.28%) were labelled as low prior knowledge (LP) participants, while the rest were considered as high prior knowledge (HP) participants. There were 16 LP learners and 13 HP learners in the experimental group; and 16 LP learners and 18 HP learners in the control group. There were no significant differences between pre-test scores of LP learners from the two groups (Table 5.14). On the other hand, the pre-test scores of HP

students from experimental and control groups was significantly different (U = 67.5, p = .046). Furthermore, I found no significant difference between the time spent per problem by LP learners in experimental group compared to those in control group. Similarly, no significant difference was found between the average time spent per problem by HP students in experimental group compared to those in control group.

Table 5.14. Effect sizes calculated by abilities

Group	**Subgroup**	**Pre-test** Mean (sd)	**Post-test** Mean (sd)	**Cohen's d**
Experimental	LP (16)	53.63 (8.65)	75.47 (13.74)	d = 1.90
	HP (13)	79.48 (7.34)	80.90 (12.51)	d = .14
Control	LP (16)	56.24 (7.40)	66.50 (14.65)	d = .88
	HP (18)	74.60 (5.72)	78.03 (8.15)	d = .49

As presented in Table 5.14, there was a substantial difference between the Cohen's d effect sizes of LP vs. HP participants in both groups. More importantly, the LP in the experimental group had a more than double effect size than LP in the control group. The results in Table 5.14 show evidence of H3, that Parsons problems with incomplete LOCs are more beneficial for novice learners, specifically those with low prior knowledge.

We were also interested in more detailed analyses of the performance of LP learners in each condition. In the experimental condition, the LP learners spent a statistically comparable amount of time per problem compared to HP students (LP: 3.97 minutes, sd = 1.79; HP: 4.04 minutes, sd = 1.18). We expected the pre/post-scores of LP and HP students to be significantly different, due to how the two subgroups were selected. Although there was a significant difference on the pre-test scores, the performance of LP learners on the post-test was not significantly different from the performance of HP students (Table 5.15). There were also no significant differences on the post-test scores for procedural, conceptual, easier or harder to guess questions between LP and HP students. LP students also had a significant improvement their pre-/post-test scores on all questions, conceptual, procedural, easier-to-guess and harder-to-guess questions. This is contrary to HP students which did not reveal any significant improvement from any of their pre-/post-test scores. Furthermore, a stronger evidence is that LP students had a significantly higher normalised gain than HP students for all questions and easier-to-guess questions. However, the pre- to post-test scores of HP students who self-explained might be due to the ceiling effect. Overall, these results show that SE prompts better support LP than HP students.

Table 5.15. LP Students and HP Students from the Experimental Group (* denotes significance at $p < .05$)

Measure (%)	**Experimental LP (16)** Mean (sd)	**Experimental HP (13)** Mean (sd)	**U, p**
Pre-test	53.63 (8.65)	79.48 (7.34)	U = 0, p = .000*
Post-test	75.47 (13.74)	80.90 (12.51)	ns
Improvement on All Questions	W = 135, p = .001*	ns	
Normalised Gain All Questions	46.65 (28.27)	20.07 (31.92)	U = 150, p = .045*
Pre-test Conceptual	50.54 (11.86)	76.54 (6.99)	U = 8, p =.000*
Post-test Conceptual	76.49 (15.39)	79.98 (15.84)	ns
Improvement Conceptual	W = 131, p = .001*	ns	
Normalised Gain Conceptual	50.47 (34.30)	30.71 (42.62)	ns
Pre-test Procedural	62.93 (24.18)	88.31 (12.55)	U = 31.5, p = .000*
Post-test Procedural	72.40 (18.94)	83.67 (10.38)	ns
Improvement Procedural	ns	ns	
Normalised Gain Procedural	30.39 (49.13)	11.66 (44.73)	ns
Pre-test Easier-to-Guess	58.33 (14.91)	96.79 (9.34)	U = 6, p = .000*
Post-test Easier-to-Guess	89.58 (20.07)	89.10 (20.80)	ns
Improvement Easier-to-Guess	W = 94.5, p = .006*	ns	
Normalised Gain Easier-to-Guess	75.00 (48.31)	4.49 (46.97)	U = 171, p = .003*
Pre-test Harder-to-Guess	50.82 (8.69)	69.09 (12.77)	U = 23, p = .000*
Post-test Harder-to-Guess	67.00 (16.75)	75.98 (11.08)	ns
Improvement Harder-to-Guess	W = 122, p = .005*	ns	
Normalised Gain Harder-to-Guess	31.37 (34.63)	23.07 (30.10)	ns

We were also interested in more detailed analyses of the performance of LP learners in each condition. In the experimental condition, the LP learners spent a statistically comparable amount of time per problem compared to HP students (LP: 3.97 minutes, sd = 1.79; HP: 4.04 minutes, sd = 1.18). We expected the pre/post-scores of LP and HP students to be significantly different, due to how the two subgroups were selected. Although there was a significant difference on the pre-test scores, the performance of LP learners on the post-test was not significantly different from the performance of HP students (Table 5.15). There were also no significant differences on the post-test scores for procedural, conceptual, easier or harder to guess questions between LP and HP students. LP students also had a significant improvement their pre-/post-test scores on all questions, conceptual, procedural, easier-to-guess and harder-to-guess questions. This is contrary to HP students which did not reveal any significant improvement from any of their pre-/post-test scores. Furthermore, a stronger evidence is that LP students had a significantly higher normalised gain than HP students for all questions and easier-to-guess questions. However, the pre- to post-test scores of HP students who self-explained might be due to the ceiling effect. Overall, these results show that SE prompts better support LP than HP students. The students' answers to SE prompts in the tutor were marked the same way as the multiple-choice questions in the pre/post-test. The HP students' scores for SE prompts were significantly higher overall, for conceptual, and for procedural SE prompts (Table 5.16).

Table 5.16. Experimental Group SE Scores (* denotes significance at p < .05)

Score (%)	**Experimental LP (16)** Mean (sd)	**Experimental HP (13)** Mean (sd)	**U, p**
SE All Questions	57.18 (12.52)	73.06 (11.25)	U = 34, p = .001*
SE Conceptual Questions	59.69 (13.08)	73.98 (11.76)	U = 43, p =. 007*
SE Procedural Questions	51.74 (16.84)	71.47 (16.08)	U = 41.5, p = .005*

Table 5.17. LP Students and HP Students from the Control Group (* denotes significance at p < .05)

Measure	**Control LP (16)** Mean (sd)	**Control HP (18)** Mean (sd)	**U, p**
Pre-test	56.24 (7.40)	74.60 (5.72)	U = 0, p = .000*
Post-test	66.50 (14.65)	78.03 (8.15)	U = 70, p = .010*
Improvement on All Questions	W = 113, p = .020*	W = 138, p = .022*	
Normalised Gain All Questions	23.92 (31.55)	17.32 (18.45)	ns
Pre-test Conceptual	52.92 (10.93)	72.41 (7.69)	U = 18.5, p =.000*
Post-test Conceptual	65.31 (17.62)	76.42 (10.50)	ns
Improvement Conceptual	W = 109, p = .034*	W = 131.5, p = .045*	
Normalised Gain Conceptual	25.93 (38.45)	18.67 (22.97)	ns
Pre-test Procedural	68.02 (18.36)	81.19 (19.39)	U = 84, p =.039*
Post-test Procedural	70.05 (16.96)	82.87 (20.06)	U = 81.5, p =.030*
Improvement Procedural	ns	ns	
Normalised Gain Procedural	15.52 (43.10)	42.01 (50.12)	ns
Pre-test Easier-to-Guess	64.58 (19.12)	94.44 (12.78)	U = 37.5, p = .000*
Post-test Easier-to-Guess	81.25 (27.13)	93.98 (12.72)	ns
Improvement Easier-to-Guess	ns	ns	
Normalised Gain Easier-to-Guess	44.79 (64.04)	6.94 (35.5)	U = 200.5, p =.050
Pre-test Harder-to-Guess	51.23 (9.63)	62.70 (9.05)	U = 57, p = .002*
Post-test Harder-to-Guess	57.65 (11.05)	68.46 (11.88)	U = 63.5, p = .004*
Improvement Harder-to-Guess	W = 108, p = .039*	W = 133, p = .039*	
Normalised Gain Harder-to-Guess	13.37 (22.87)	18.74 (27.86)	ns

Looking at the control condition, the LP learners spent similar amount of time per problem compared to HP students from the same group (LP: 3.21 minutes, sd = 1.31; HP: 3.22 minutes, sd = 1.10). Contrary to results of the experimental group, the HP students scored significantly higher both on the pre- and the post-test in comparison to LP students (Table 5.17), apart from the post-test scores on conceptual and easier-to-guess questions. Moreover, in comparison with the LP students in the experimental group, LP students also improved significantly on pre- to post-test scores on all questions, conceptual, and harder-to-guess questions; but not on easier-to-guess questions. Interestingly, the HP students in the control group also improved significantly on pre- to post-test scores in a similar manner as LP students in the control group (all questions, conceptual, and harder-to-guess questions). Contrary to the HP students in the experimental group, who did not show signifant improvements in any pre- to post-test scores. When normalised gains of

LP and HP in the control group was compared, only a marginal difference was found between easier-to-guess questions; the rest were not significant.

5.5 Discussion

My results demonstrate the effectiveness of both PyKinetic_IncLOCs and PyKinetic_IncLOCs_SE in supporting learning for introductory programming students. Although the participants only interacted with the tutor for roughly 45 minutes, my results still revealed positive learning effects.

One of the contributions of my research is the innovative design of Parsons problems with incomplete LOCs which contained four features that are uncommon when compared to other work. However, it is important to note that each characteristic is not unique by itself, rather it is the combination of these features below that I consider as unique.

Firstly, Parsons problems in both versions of the tutor are composed of only one area acting as both a problem and solution space. As discussed in Chapter 2.7, in other implementations of Parsons problems learners drag the blocks of code from the problem area onto the solution area. Although other implementations also allow learners to rearrange blocks of code within the solution area, the main difference lies on the initial state of the problems. Combining the problem and solution area makes better use of the space in smartphones which allows for longer problems. However, having only one area means that learners might be overwhelmed because all the LOCs are displayed at once in the same space, and learners need to mentally separate their solution from the rest of the LOCs in the problem since there is no separate solution area. I recognise that combining the problem and solution area might make it difficult for learners to keep track of the LOCs they have moved. However, I did not observe this to be an issue, probably because I limited the length of the problems to have maximum of 16 LOCs.

Secondly, since I designed both PyKinetic_IncLOCs and PyKinetic_IncLOCs_SE for novice learners, I provided scaffolding for indentations, contrary to other implementations of Parsons problems (Parsons and Haden 2006; Ericson, Margulieux and Rick 2017; Ihantola and Karavirta 2011; Karavirta, Helminen and Ihantola 2012; Ihantola, Helminen and Karavirta 2013; Kumar, 2018; Harms, Chen, and Kelleher 2016). Thirdly, my design required students to move individual LOCs instead of moving blocks of code like Garner (2007) and Kumar (2018), thus encouraging students to think about each LOC. Lastly, I introduced menu-based SE prompts for every incomplete LOC, to provide support for conceptual knowledge. I acknowledge that acquisition of both procedural and conceptual knowledge is important for novice learners. Solving Parsons problems with incomplete LOCs supported procedural knowledge. Therefore, I have introduced menu-based SE prompts for every incomplete LOC, to provide support for conceptual knowledge. The combination of Parsons problems and SE prompts is also one of my main

contributions, as this has not been done earlier to the best of our knowledge. Once again, I stress that my design of Parsons problems combined SE prompts as an entity is one of my contributions rather than the specific differences of my implementation with other work.

Self-explanation has previously been proven effective in many domains, implemented on personal computers and I found that it is also beneficial when learning programming in a mobile tutor. We are not aware of other work who had evaluated menu-based SE on smartphones. Thus, another one of my contributions is designing and evaluating SE on smartphones. Furthermore, I found that SE prompts are effective when combined with activities like Parsons problems which are less demanding than problem solving but more difficult than studying worked examples. My results are consistent with Rittle-Johnson (2006) and Matthews and Rittle-Johnson (2009) who also evaluated SE prompts after filling in a blank line but in the domain of mathematical equations.

In my study, the experimental group participants had to respond to all SE prompts. I did not allow the students to skip the SE prompts to support possible missing gaps in the learners' knowledge regardless of their abilities. I suspect that if I had allowed participants to skip the SE prompts, poorer students would probably choose to not attempt the prompts, thus losing chances to deepen knowledge. Aleven and Koedinger (2000) reported that students did not always follow through their tutor's SE prompts.

I presented evidence showing that menu-based SE prompts further improved students' learning. Experimental group participants completed less problems on average, with most not completing the last two problems (i.e. the most difficult problems). Despite completing fewer problems, the experimental group still learned more. Students who self-explained had a significantly higher normalised gain for all questions than those who did not. The Cohen's d effect size on the treatment revealed a moderate effect (d= 0.493).

My findings revealed that students from both groups, regardless of whether they solved SE prompts, improved more on conceptual questions than procedural questions. The experimental group had a significantly higher normalised gain on conceptual questions. Moreover, a moderate positive effect size was found for conceptual questions (d = 0.556). On the contrary, procedural questions yielded a weak negative effect (d = -0.158). Further evaluations are needed to investigate why procedural questions seemed to have had a negligible effect. My findings show that learners who self-explained improved their conceptual knowledge more than procedural knowledge. However, I do not have enough evidence to show that Parsons problems have a specific influence in enhancing conceptual knowledge more than procedural knowledge. Because, to the best of our knowledge there are no studies comparing the effectiveness of Parsons problems in enhancing conceptual knowledge questions to procedural knowledge. Therefore, I advise further evaluations to verify the specific impact that Parsons problems have with harnessing conceptual and procedural knowledge.

I have enough evidence to show that SE prompts enhance conceptual knowledge. Based on my

results, learners who self-explained performed significantly better on conceptual questions than those who did not. My results are supported by literature showing self-explaining enhances conceptual knowledge (Najar, Mitrovic and McLaren, 2016). Furthermore, I have provided evidence that this also holds for learning with a smartphone tutor. Thus, combining Parsons problems with SE prompts helped learners in the experimental group to perform better on conceptual questions. However, I did not find any evidence that self-explaining is beneficial for enhancing procedural knowledge.

My results verified that Parsons problems are suitable for novice learners, specifically for those with low prior knowledge. The reason behind this is most likely the amount of scaffolding provided in Parsons problems, as they contain correct syntactic structures. Morrison et al. (2016) also have similar insights that Parsons problems require lower cognitive load than other activities like code writing. Students with high prior knowledge are usually well-versed with syntax compared to those with low prior knowledge, so they often do not require scaffolding for syntax. I also found distinctive evidence that PyKinetic_IncLOCs_SE was more effective for LP students who had SE compared to LP students who used PyKinetic_IncLOCs (without SE provided). This result was expected, as weaker students often find it difficult to self-reflect and fill gaps in their own knowledge (Chi et al., 1989). LP learners who self-explained had a notably high Cohen's d effect size (d=1.95), more than double the effect size for LP students in the control group (d = .91). Furthermore, LP learners who self-explained had reached the performance of HP students on the post-test. On the contrary, although LP students in the control group improved from their pre- to post-test scores, their post-test scores were still significantly lower in comparison to the HP students from the same group.

A possible explanation for LP students learning more with Parsons problems than HP students is a ceiling effect. It is probable that my HP participants particularly in the experimental group might have shown a ceiling effect. However, in the control group HP learners had significant improvement on procedural questions despite having high pre-test scores. Furthermore, it is thought-provoking that HP learners in the control group acquired significantly higher normalised gains on procedural questions compared to LP students. Due to low numbers in these subgroups, I do not propose that HP students are better without SE. However, I am inclined to suggest that this might be possible.

I do not postulate that a ceiling effect is an accurate explanation for LP students benefitting more than HP students when solving Parsons problems (especially with SE prompts). This is due to results from my other study (Chapter 6) wherein I evaluated a different version of PyKinetic containing output prediction and debugging problems without Parsons problems. In this study, I have found the opposite outcome, where LP students had negligible learning effects, but HP students had significant learning benefits. Therefore, I have reasons to believe that Parsons problems are indeed more suitable for LP students, while output prediction and debugging problems for HP students.

Although learning effects were positive, there are certain aspects of the SE prompts which could be

improved. I have chosen to utilise a mixture of questions written in a positive or negative manner, to increase the difficulty of the questions. However, during the study, some of the participants did not read the questions properly and mistakenly answered some SE prompts. For example, a question which had "Select all INCORRECT statements….", confused some participants, where they instead selected all *correct* statements. This issue may or may not be concerning, as I have observed that these questions compelled most participants to be more cautious after this occurrence. Some participants were observed to have completed the incomplete LOC first before reading the rest of the problem. This made the SE prompts to appear more difficult for them, as answering some of the prompts requires students to understand the entire code, not just the incomplete line. It may have helped to display a message advising learners to rearrange the LOCs first, before filling in the missing keyword/s for the incomplete LOC.

5.6 Conclusions

My variant of Parsons problems with a pedagogically-guided design is one of my main contributions, which included 1) one area acting as both problem and solution areas; 2) scaffolding for indentations; 3) all blocks of code each with single LOCs; and 4) incomplete LOCs with menu-based SE prompts. There is some evidence that Parsons problems are positively correlated with code writing (Denny et al., 2008); a skill considered as procedural knowledge in programming. Therefore, I have added incomplete LOCs to further support the acquisition of procedural knowledge. I also introduced menu-based SE prompts for every completed LOC to support the acquisition of conceptual knowledge. These SE prompts were proven to be effective in the version of PyKinetic in this study which contains Parsons problems that are puzzle like exercises, consistent with work by Johnson and Mayer (2010). My research revealed that menu-based SE prompts are also effective on a mobile platform. I have also addressed a research gap in the knowledge of self-explanation by combining it with an activity which is more challenging than learning with worked examples, but less demanding than problem solving. My results revealed that solving SE after solving incomplete LOCs is effective, consistent with work by Rittle-Johnson (2006) and Matthews and Rittle-Johnson (2009) who also evaluated SE prompts after filling in a blank line but in the domain of mathematical equations. More specifically, I found that SE is also effective when combined with activities such as Parsons problems which requires lower cognitive load than problem solving, but higher when compared to worked examples. The evaluation of a combination of Parsons problems and SE, as well as menu-based SE prompts on smartphones have not been done before to the to the best of our knowledge. Further research on the effectiveness of other types of SE prompts in mobile tutors is needed.

The goals of the evaluation study presented in this chapter were to evaluate three hypotheses. The findings supported my first hypothesis H1, with both groups having improved significantly from the pre-

to the post-test. Both groups also improved significantly on the harder-to-guess questions. I also showed enough evidence to support my second hypothesis H2, with a moderate effect size on the participants who self-explained, and their normalised gains were also significantly higher than those in the control group. Moreover, the evidence was more distinctive with LP learners. I have also shown enough evidence to accept my hypothesis H3, that LP students will learn more than HP students. LP learners from both treatments have learned significantly more than HP students. However, LP students who self-explained benefitted the most, as they achieved similar scores to those of the HP students in the post-test and had a very high Cohen's d effect size.

I observed that participants were more engaged than expected, possibly because most of them have not experienced learning Python on smartphones before. Some of them also asked if they can download the version of PyKinetic they used in the evaluation to continue learning. Although the participants were informed that they were free to stop the session whenever they wanted, all participants stayed until the time was up. Some participants commented that they wished they could stay longer and finish all the problems in the tutor. On the downside, as participation in the study was voluntary, this may have affected some of the learners' performance on the pre- and post-tests. Participants may or may not be motivated in answering the exercises as they were informed that they will be compensated similarly regardless of their performance.

One limitation of my study was the larger proportion of conceptual vs. procedural questions in the SE prompts, pre-test and post-test. This is not alarming, as the proportions in all three were consistent. However, the results may have been different if the proportion of conceptual vs. procedural questions were equal. Another limitation on my study was the session length. Longer sessions may be considered in my future evaluations, as well as a delayed/transfer test, which may yield more interesting results. The number of participants could have also been improved. Since my study was on a mobile device, running controlled experiments was not as straight-forward as running it on personal computers. I found that the WiFi connection on some of the phones were not always reliable, which led to some data loss. I plan to conduct future studies using only my development phones, to be able to save the data in two places (on the server sent through WiFi, and on the smartphones for backup). Another avenue for future work is to develop a version of PyKinetic with additional game features, in order to increase student motivation.

Both versions of PyKinetic used in this study were designed specifically for novice programmers. My findings support this, and more specifically revealing that PyKinetic is more beneficial to LP students than HP students. A later version of PyKinetic is covered in the next chapter (Chapter 6) which consists of debugging and output prediction activities instead of Parsons problems.

6 SECOND EVALUATION STUDY

This chapter covers my second evaluation study and addresses research questions (R1) and (R3-R8). Similar to what I mentioned in Chapter 5, some of the research questions above were not fully addressed solely in this chapter, rather were addressed collectively in all my evaluation studies. However, amongst the research questions above, (R8) was tackled exclusively in this chapter. In this chapter I implemented a version of PyKinetic (PyKinetic_DbgOut) with a combination of debugging and output prediction activities. For (R4), I evaluated PyKinetic_DbgOut to investigated whether the combination was effective. For (R5), I investigated whether combining exercises with different natures is an effective pedagogical strategy for PyKinetic. Apart from the combination, I regard the sequencing of the exercises as a pedagogical strategy like McArthur et al. (1988). Lastly for (R1), the evaluation study in this chapter investigated the usability of PyKinetic by getting feedback from participants.

The following section exhibits the activities implemented in this version of PyKinetic_DbgOut. Next, I present PyKinetic_DbgOut which comprised with the activities as discussed. Following that, the architecture and development of PyKinetic_DbgOut is addressed. After that, details about the evaluation study are presented: experimental design and findings. After revealing the findings, is the section on discussion. Finally, this chapter ends with the conclusions.

6.1 Activities

The problems in PyKinetic_DbgOut consist of the problem description, code (containing 0–3 incorrect LOCs), 1–3 activities, and 1–3 questions for each activity. There are five types of activities (Table 6.1): three types of debugging activities and two types of output prediction activities.

In Dbg_Read activities, the learner was given test cases with the actual output; the learner's task was to specify whether the given code was correct or not. Figure 6.1 shows an example of Dbg_Read where a learner was first given the problem description (Figure 6.1, left). Function `go_to_work` should display True or False if a person is required to go to work, based on the value of two Boolean parameters: `off_day` and `public_holiday`. Variable `off_day` is True if today is the person's day off work, and False if it is a normal working day. Variable `public_holiday` is True if today is a public holiday, and False if it is regular working day. The learner then needs to respond on whether the code is error-free or not, based on the given code and the problem description. If the code contains error/s, it is not mandatory to determine the part of the code which causes the error. The learner was also given test cases where PyKinetic_DbgOut displayed the actual output of a test case when the learner long-presses on the test case. Figure 6.1 (middle) shows

that for the first test case, the output is True when the parameter `off_day` is True and `public_holiday` is False. The same output of True is also returned when `off_day` is False and `public_holiday` is True. The test cases show that the function contains an error since the function should only return True if the person needs to go to work. In Figure 6.1 (right), the learner answered incorrectly, so PyKinetic_DbgOut highlighted the LOC causing the error in red. In cases where the learner responded correctly to a problem where the given code is incorrect (like shown in Figure 6.1), the LOC causing the error is not highlighted. Regardless of the solution, the only feedback displayed when a learner rightfully answers a code reading problem was "Correct! Great job!" (LOC causing error not highlighted).

Table 6.1 Five Types of Debugging and Output Prediction Questions in PyKinetic_DbgOut

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

The second type of debugging activities (Dbg_Ident) provided similar information to the learner but instead of asking whether the code was error-free or not, the learner was made aware that the code contained errors by specifying the number of incorrect LOCs to be identified. The learner was required to identify one to three incorrect LOCs per problem. An example is shown in Figure 6.2 (middle), where the student needs to identify one incorrect line (the line the student selected is highlighted in turquoise). In this example, the student had successfully identified the incorrect line. The incorrect line in this problem produces a syntax error because the LOC is using '`x`' as an operator for multiplication when in fact it should have been '`*`'. Therefore, all test cases in Figure 6.2 (right) results in a syntax error (message displayed below screenshot).

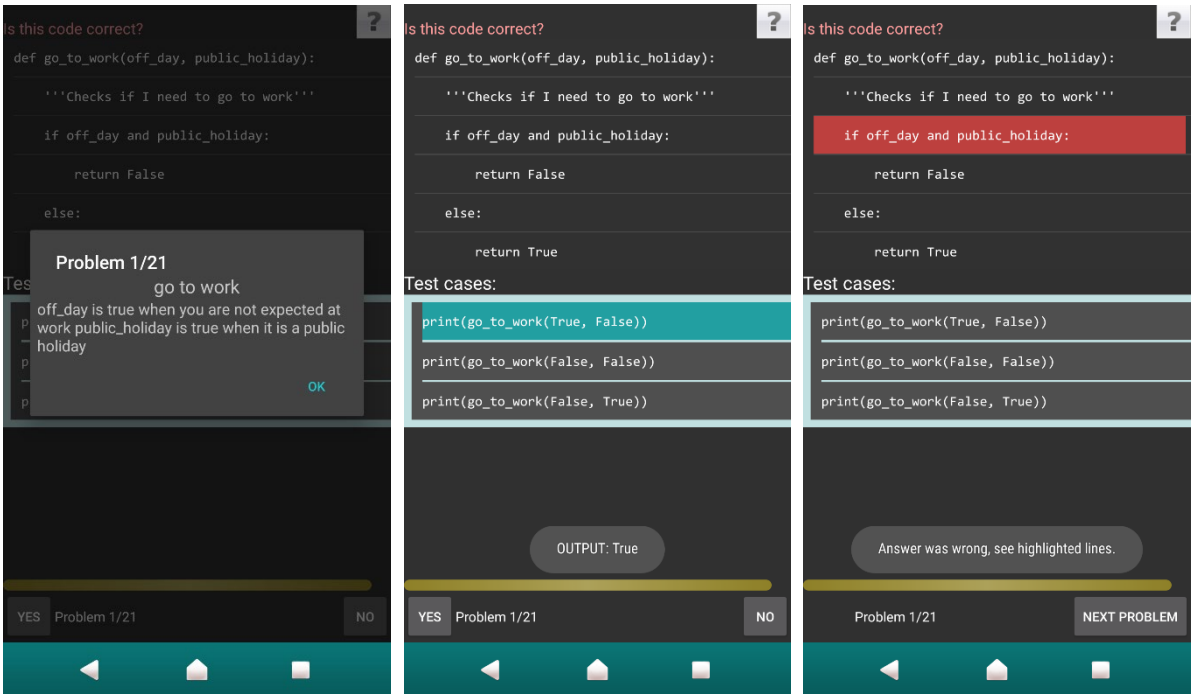


Figure 6.1 Example of a Dbg_Read activity in PyKinetic_DbgOut (left: problem description; middle: code with test cases, output shown is the actual output of the highlighted line; right: feedback when learner answered incorrectly)

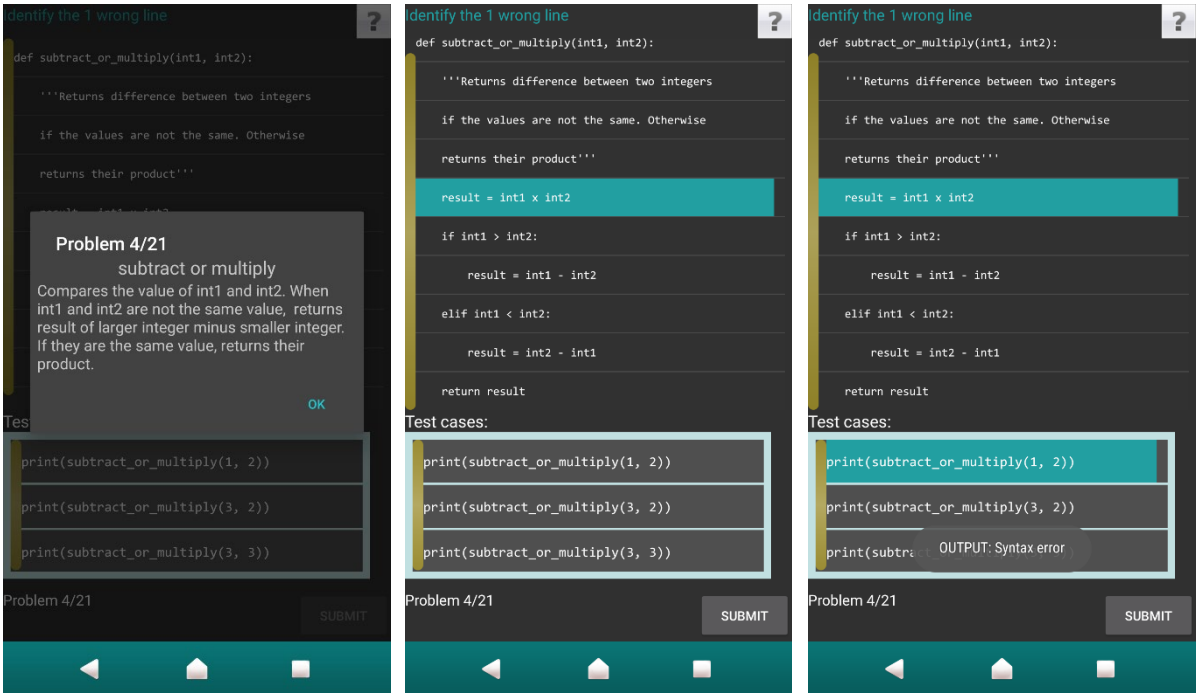


Figure 6.2 Example of a Dbg_Ident activity in PyKinetic_DbgOut (left: problem description; middle: code with test cases, where the highlighted line is the correct answer; right: similar screenshot to the one on its left, but with the actual output of the first highlighted test case shown – "OUTPUT: Syntax error")

The third type of debugging activities is Dbg_Fix, which started by requiring the student to identify incorrect lines (Dbg_Ident), and then to fix them (Figure 6.3, right). To fix incorrect LOCs, the student needed to select the correct option from given choices by tapping on the LOC. Initially, LOCs that required fixing were highlighted in orange. In the screenshot shown in Figure 6.3 (right), the student completed the line highlighted in green, and needed to work on the other line. If the learner fails to fix a LOC, it is highlighted in red.

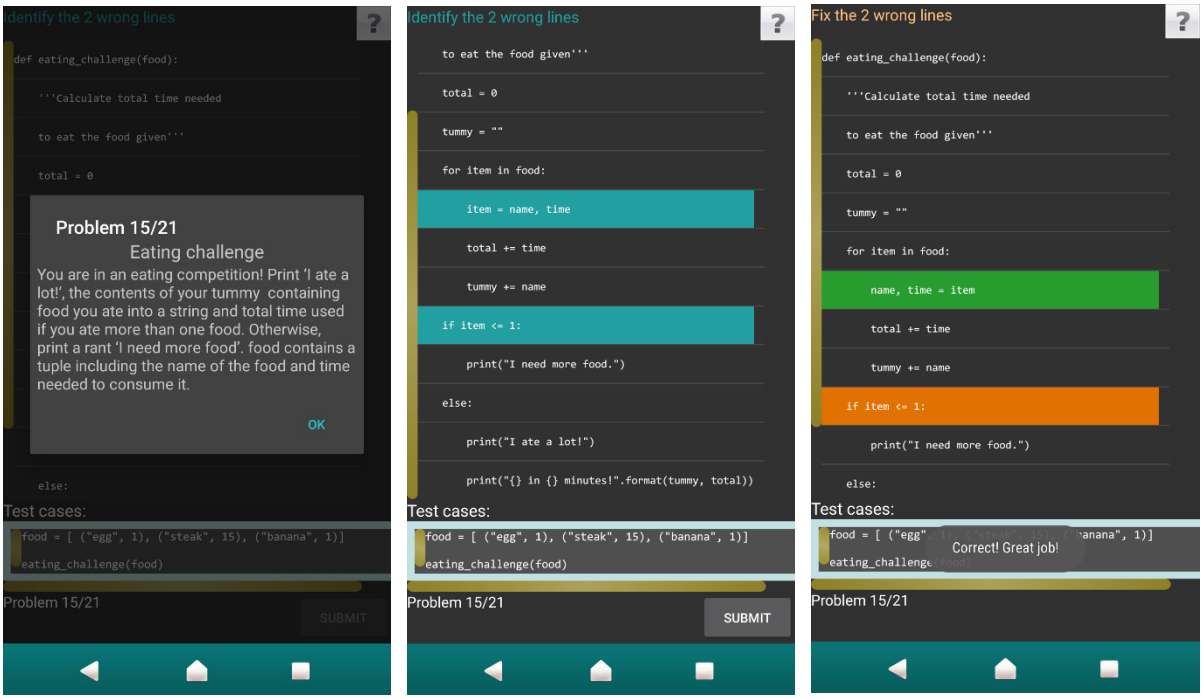


Figure 6.3 Example of a Dbg_Ident –> Dbg_Fix problem in PyKinetic_DbgOut (left: problem description; middle: Dbg_Ident activity with the erroneous LOCs highlighted; right: Dbg_Fix activity with the highlighted lines that need to be fixed)

Each output prediction activity contains 1–3 test cases. In the first type (Out_Act), the student needs to specify the actual output of the code for each given test case (Figure 6.4, right). The learner successfully identified the output for one of the test cases (Figure 6.4, middle). In response to this, PyKinetic_DbgOut highlighted the test case in green (Figure 6.4, right), and the rest of the test cases needs to be solved to complete the Out_Act activity. In the example provided in Figure 6.4, the given code does not contain any errors. Therefore, the actual output and expected output are synonymous. However, in some Out_Act activities the given code is erroneous. All given Out_Act activities with erroneous code starts with a Dbg_Ident activity where the learner is required to first identify the erroneous LOCs (Figure 6.5, middle), then proceed to the Out_Act activity (Figure 6.5, right). If the code is erroneous, the actual output may be none with an error displayed.

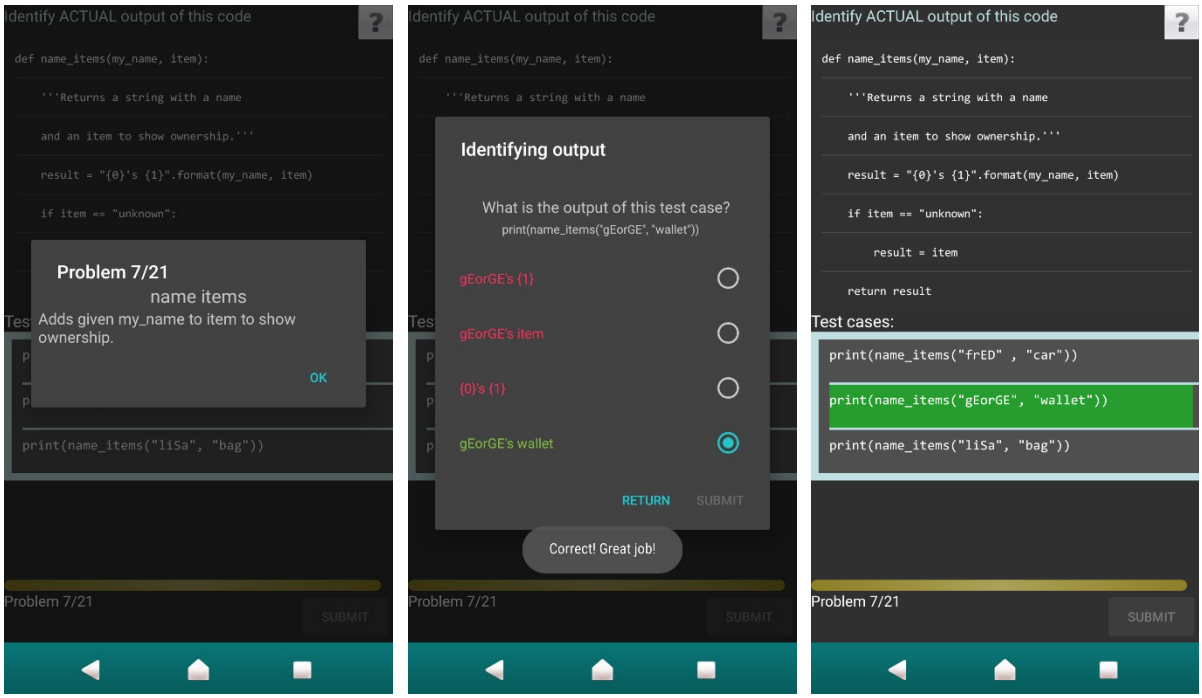


Figure 6.4 Example of a Out_Act activity in PyKinetic_DbgOut (left: problem description; middle: Out_Act activity with the last option displayed in green as it is the correct answer; right: Out_Act activity with one of the test cases highlighted in green as it was completed)

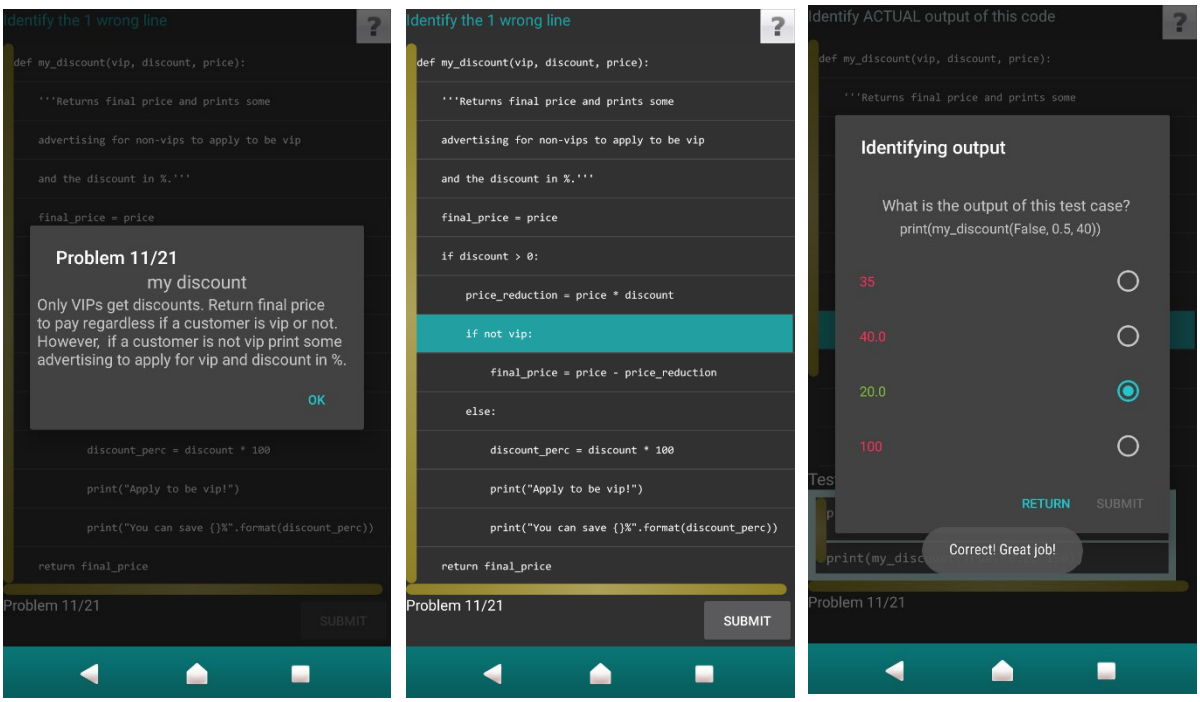


Figure 6.5 Example of a Dbg_Ident –> Out_Act activity in PyKinetic_DbgOut (left: problem description; middle: Dbg_Ident activity with the erroneous LOC highlighted, right: Out_Act activity, output prediction question for one of the test cases)

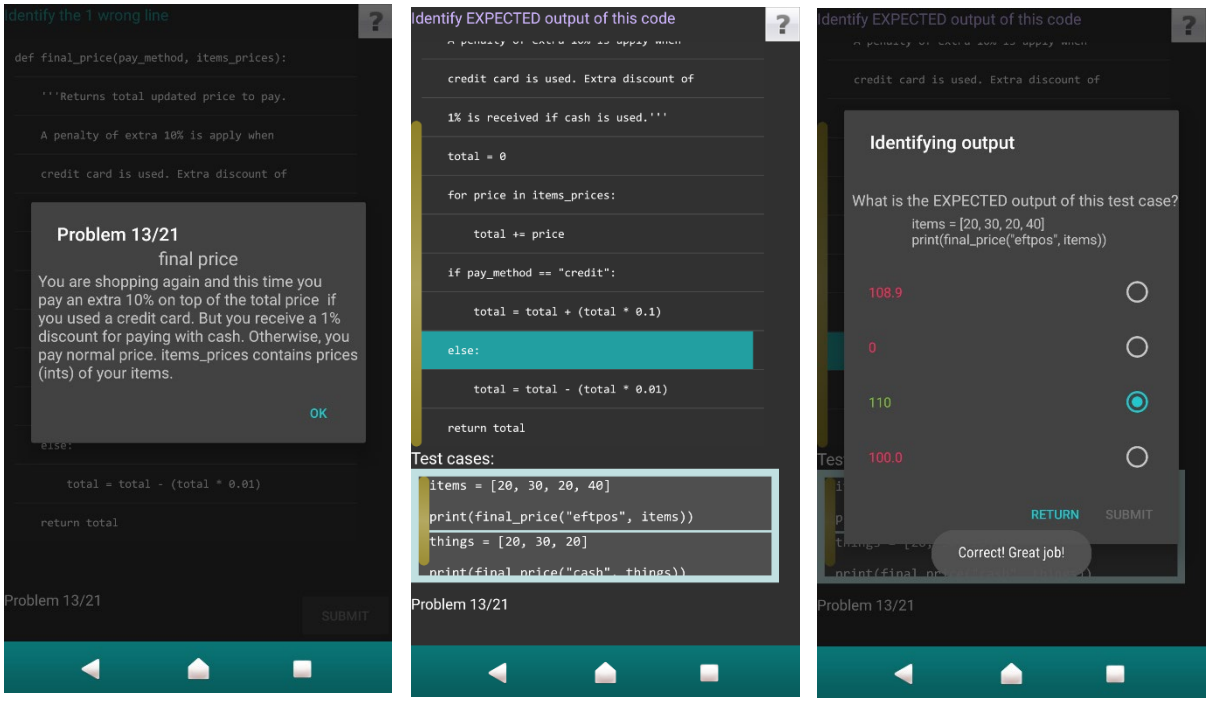


Figure 6.6 Example of an Out_Exp activity in PyKinetic_DbgOut (left: problem description; middle: code with erroneous LOC already identified, right: output prediction question for the first test case)

In Out_Exp activities, the student specified the expected code output matching the problem description regardless of the error in the code. Like Dbg_Ident –> Out_Act problems, all Out_Exp activities first started with a Dbg_Ident activity where the student needs to identify the error in code before predicting the output. Figure 6.6 shows an example of a problem with Out_Exp where PyKinetic_DbgOut first showed the problem description (left). The problem contains a function with two parameters: a string parameter `pay_method`, and a list parameter `items_prices`. The problem requires to sum all the values in the given list `items_prices` and to put the sum into the variable `total`. Afterwards, the value of `total` is adjusted based on the string parameter `pay_method`. An additional 10% is added to variable `total` if the variable `pay_method` contains the string "`credit`". However, if `pay_method` contains the string "`cash`", the variable total should be deducted by 1%. In all other cases, the value of the variable `total` should remain the same. Lastly, the code should return the value of `total`. In Figure 6.6 (middle), the learner successfully identified the erroneous LOC in the code (highlighted in turquoise) and then asked to identify the expected output of the code for each given test case. In Figure 6.6 (right), the learner correctly identified the expected output in the given test case (shown in green font). Notice that because the given first parameter (`pay_method`) contained "`eftpos`" the expected output should be the sum of all integers in the given list without any changes. If the question asked for the actual output, because of the error in the code (highlighted in turquoise in Figure 6.6, middle), the answer would have been 108.9. However, the question was asking about the expected output, so the correct answer was 110.

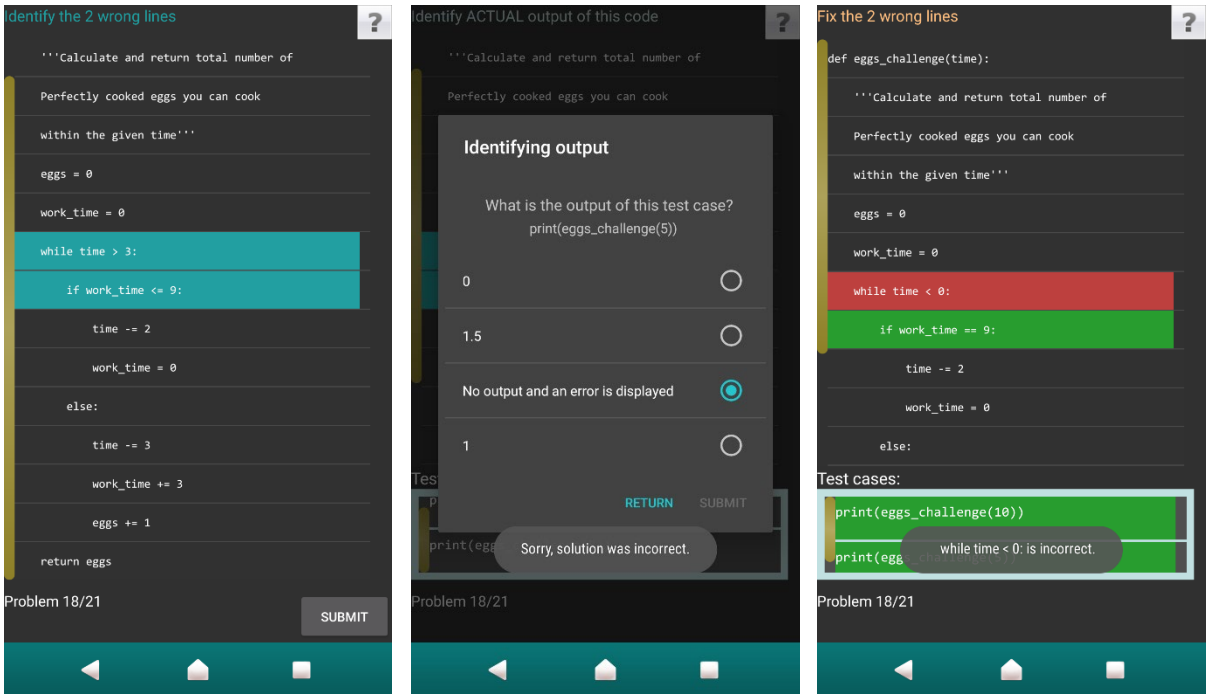


Figure 6.7 Example of Dbg_Ident –> Out_Act –> Dbg_Fix activities in PyKinetic_DbgOut (left: Dbg_Ident; middle: Out_Act; right: Dbg_Fix)

6.2 PyKinetic_DbgOut

PyKinetic_DbgOut had 21 problems provided in a fixed order. There were seven levels of complexity, each containing 2–4 problems (Table 6.2). 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 activity. For example, problem one was a code reading problem, containing only one activity – Dbg_Read. The complete code, problem description, and test cases with function calls were given; the task was to identify if the code is correct or not.

Table 6.2 Combinations of Questions in Levels 1-7

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

Each problem in levels 4–7 contains 2–3 activities and covered same topics as in the earlier levels as well as for loops, while loops, and importing a module. Problems on these levels started by requiring the student to identify incorrect LOCs (Dbg_Ident). After that, levels four and five were followed by output prediction activities: identifying the actual output (Out_Act) for level four and identifying the expected output (Out_Exp) for level five. Level six targeted code writing skills, by requiring the student to fix erroneous LOCs (Dbg_Fix) in the second activity. The code fixing was achieved by tapping through the options with the lines changing for each tap, instead of showing a separate dialog for the options like work of Ihantola et al. (2013). Lastly, level seven contained three types of activities in each problem: identifying

erroneous LOCs (Dbg_Ident), identifying actual output (Out_Act), and fixing erroneous LOCs (Dbg_Fix). The problem illustrated in Figure 6.7 belongs to level seven. It is important to note that the ordering of the problems is based on factors mentioned in Section 3.13; it was 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 has fewer lines and topics.

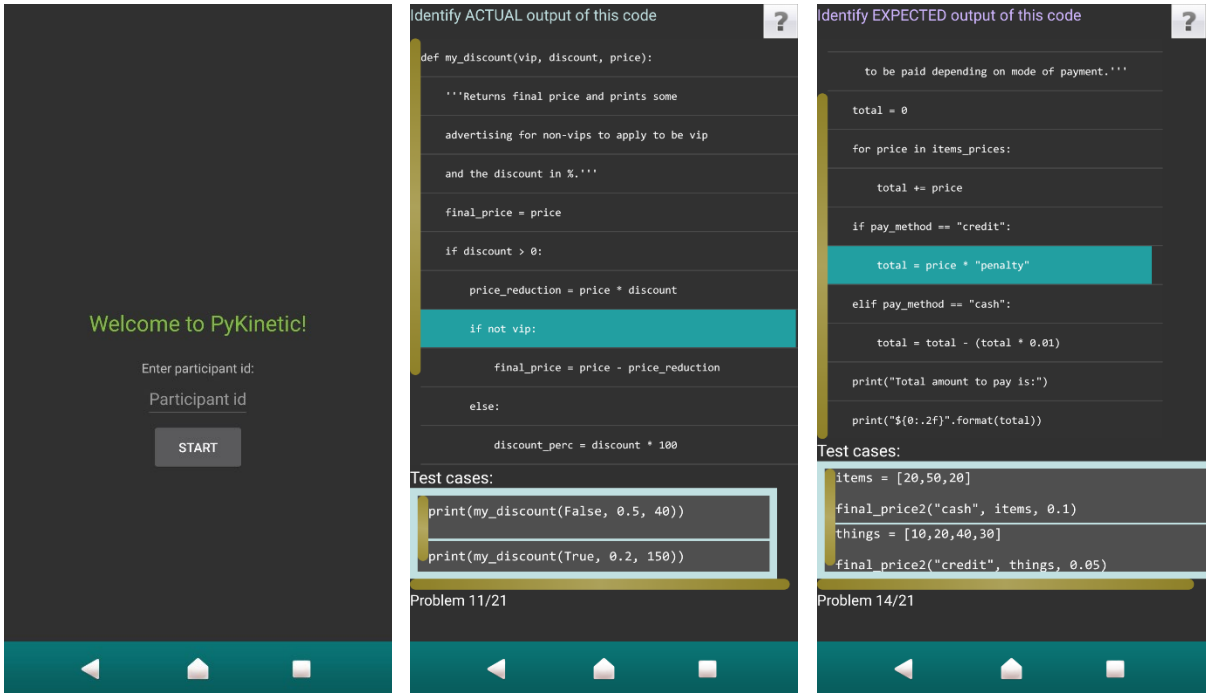


Figure 6.8 Screenshots (from left to right): start screen, problem-solving screen (Out_Act), and problem-solving screen (Out_Exp)

Like the versions used in my first evaluation study (Chapter 5), there were only two screens in PyKinetic_DbgOut: start screen and problem-solving screen. The *start screen* was identical to the versions PyKinetic_IncLOCs and PyKinetic_IncLOCs_SE and was utilised in the same manner – for participants to enter their unique participant id given to them (Figure 6.8, left). The *problem-solving screen* was used for all activities in PyKinetic_DbgOut (Figure 6.8: middle and right; Figure 6.9). Despite offering five different activities, all activities were portrayed to learners using what it seems to be like only one multi-purpose screen. Technically speaking these were implemented using three separate Android activity classes for the code reading activities, debugging activities, and output prediction activities. However, my design depicts only one screen to promote a seamless user experience. All versions of PyKinetic including PyKinetic_DbgOut were intricately designed to adhere to the user interface design guidelines covered in Chapter 3 as I recognise that it is vital for learning effectively.

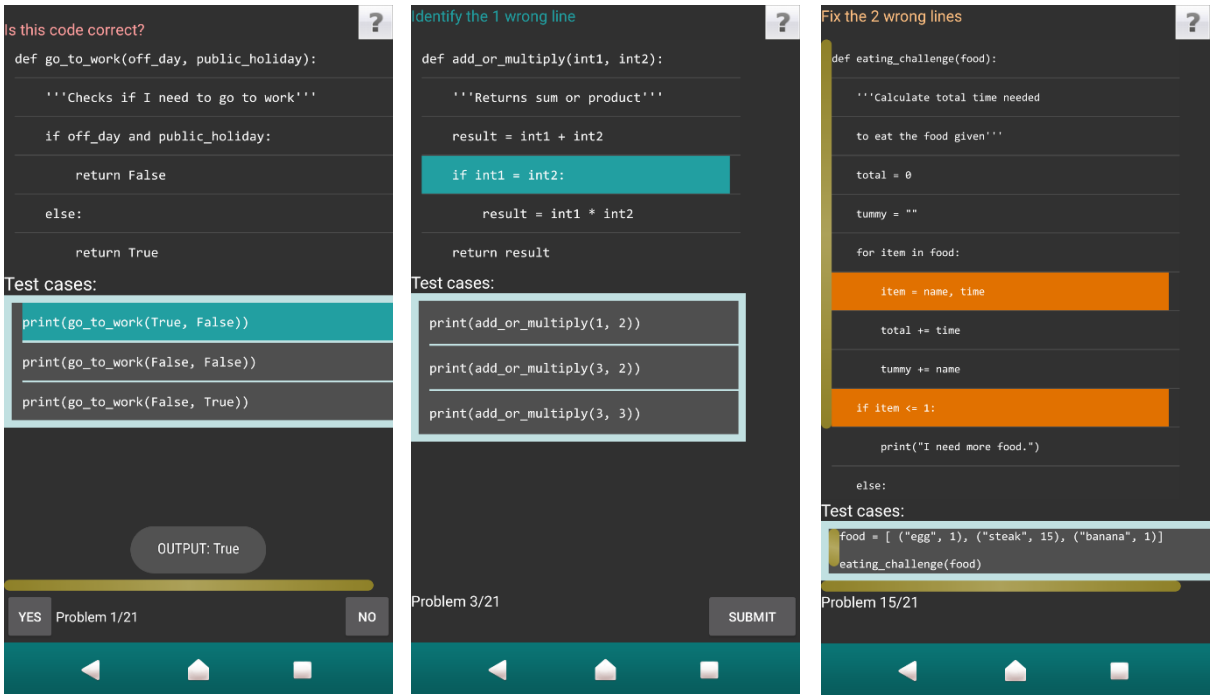


Figure 6.9 Screenshots (from left to right): problem-solving screen (Dbg_Read), problem-solving screen (Dbg_Ident), and problem-solving screen (Dbg_Fix)

6.3 Architecture and Development

Like in other chapters, the general architecture of the application is described in Chapter 3.4.1. Specific characteristics of PyKinetic_DbgOut are described here. The code statistics are reported in Table 6.3. Like in my other evaluations, note that Table 6.3 only includes code that I wrote. Other files like third-party libraries are not represented in Table 6.3 even though used by PyKinetic_DbgOut. Moreover, files generated upon compilation, building, and executing of the application is not represented in Table 6.3.

Table 6.3 Code Statistics for PyKinetic_DbgOut

	Java	**XML**
Classes/Files	23	8
Total LOCs without blank lines	5707	555
Source code lines	4721	555
Comment lines	986	0

With regards to the manifest XML file in this version, there were four permissions in PyKinetic_DbgOut: internet access, access to the state of the network connection, reading and writing data in the device. All the permissions were necessary for recording logs (in real time) during the evaluation study; where the logs contained user actions. The first two permissions regarding network connectivity was necessary for sending logs to a server. The latter permissions for reading and writing data were to record

the logs in the devices used in the evaluation study.

6.3.1 Storage

Like previous versions of PyKinetic, the problem set was stored within the SQLite database in the application. Since only simple feedback was provided, the feedback was not stored in the database, but instead in the strings XML file together with the other texts used throughout the application. Logs were recorded and sent through a server. These logs were simply stored as text files. Furthermore, a simple php admin page was made to monitor the progress of the participants during the evaluation study in real time, via the logs received in the server.

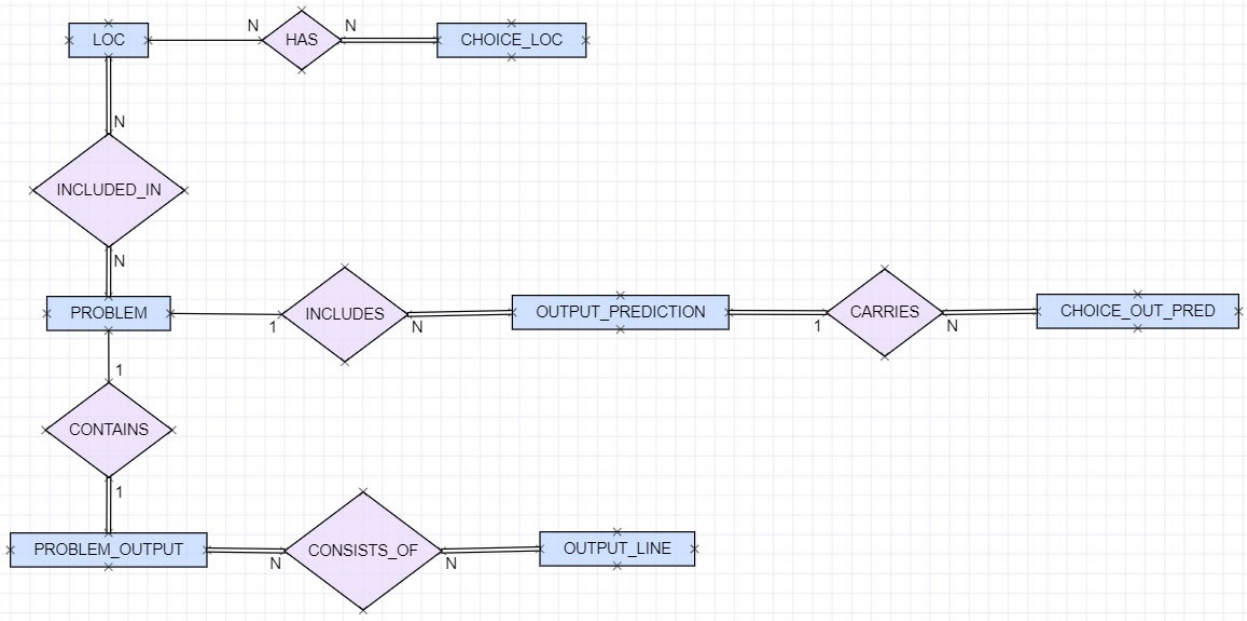


Figure 6.10 ER diagram of the database in PyKinetic_DbgOut without the attributes

The ER diagram of the database used in the evaluation study is shown in Figure 6.10 (without the attributes). Only LOCs that were used in the Dbg_Fix contains choices. All problems regardless of the type of activity contained at least one LOC. Some problems contained either Out_Act or Out_Exp activities, and for each output prediction question, a set of choices were stored. Furthermore, only problems without output prediction activities provided an output description. Like the database structure in other versions, each line in the output may have been used in multiple problems. For instance, output lines displaying "0" may also be contained in the output of another problem.

6.3.2 Interaction Elements

The list of navigational elements used in the prototype is reported in Table 6.4. Elements require either single tap or long tap actions.

Table 6.4 List of buttons, icons, and navigational elements

Screen/Dialog	Element	Purpose
Any problem	"?" button	Displays the problem details dialog box
Problem solving screen problems 2-21	Back button	Screen stays in place and "Sorry, you can't go back to the previous problem." is displayed below the screen.
Problem solving screen for all problems expect the last	NEXT PROBLEM button	Navigates to the next problem in the sequence. Note that this button is only displayed when the learner has completed the current problem.
Problem details dialog box (Same for all problems)	Ok button	Closes the dialog box
Feedback dialog box (Dbg_Read activity)	Ok button	Closes the dialog box
Start screen	Edit text for participant id	Element where users enter in their participant id for evaluation purposes
	Start button	Navigates to the first problem offered in this version of PyKinetic_DbgOut
	Device's back button	"Closes" the app (not completely, it will still run in the background)
Dbg_Read problem-solving screen	YES button	The learner taps on this button to respond that the given code is correct.
	NO button	The learner taps on this button to respond that the given code is wrong.
	Test cases	A long tap on a test case shows its actual output when it is executed.
Dbg_Ident problem-solving screen	Non-test case LOC	A long tap highlights the LOC in turquoise, indicating that it is selected.
	Test cases	A long tap on a test case shows its actual output when it is executed.
	SUBMIT button	Submits the LOC/s selected as the solution to receive feedback. (Only enabled when there is at least one LOC selected)
Out_Act and Out_Exp problem-solving screen	Test cases	A long tap on a test case shows the output prediction dialog for the chosen test case.
Out_Act and	Device's back	Disabled

Out_Exp dialog box	button	
	Output choice	Tapping on a choice indicates that the learner has selected this as his/her solution and at the same time SUBMIT button is triggered. Other choices selected before are unselected at the same time.
	SUBMIT button	Initially disabled, used by the learner to submit their solution to the output prediction activity.
	RETURN button	Dismisses the dialog box
Dbg_Fix screen	LOC which requires fixing	Long tapping on LOC that needs to be fixed will be highlighted in turquoise. A LOC in turquoise is activated, indicating that the learner is working on this LOC. Tapping on an activated LOC reveals the alternative choices (the entire LOC changes on tap). Long tapping on an activated LOC indicates that the learner is submitting this LOC as their answer (to replace the old LOC to be fixed). Tapping on a regular LOC does not do anything. Similarly, tapping on an unselected LOC to be fixed does not do anything.

6.4 Experimental Design

I conducted a controlled lab study with PyKinetic_DbgOut, focused on investigating the effectiveness of the activities in the tutor. I conducted a study with PyKinetic_DbgOut to address the following research questions:

(Study2_ R1) Is the combination of coding activities effective for learning programming?

(Study2_ R2) How do the activities affect the skills of students with lower prior knowledge (LP) compared to those with higher prior knowledge (HP)?

(Study2_ R3) How can I improve the usability of PyKinetic?

For (Study2_ R1), my hypothesis (H1) was that the combination of coding activities is effective for learning programming. I have combined output prediction and debugging activities to implement PyKinetic with a component-skills perspective (McArthur et al., 1988), aiming to target multiple programming skills. For (Study2_ R2), my hypothesis (H2) was that lower prior knowledge (LP) students would have higher learning gains in comparison to (HP) students. I expected LP students to have significantly higher learning gains than HP because they have more room to improve than HP students. Secondly, although debugging is considered a higher order of skill than code-writing (Fitzgerald et al., 2008; Ahmadzadeh, Elliman, and

Higgins, 2005), the debugging exercises I designed in PyKinetic_DbgOut are simple exercises for novices. Therefore, I expected that it would be more suitable for LP students than HP.

6.4.1 Participants

In the controlled study, I had 37 participants recruited from an introductory programming course (COSC121) at the University of Canterbury. I eliminated data about two participants as they did not finish the study and present the findings from 35 participants (23 males and 12 females). The study included a short survey on demographics which was given before after interacting with PyKinetic_DbgOut, and two more questions after interacting with PyKinetic_DbgOut (Appendix B). Four participants did not disclose their ages, so the average age of the rest of the participants was 21 (sd = 6.9). There was one older participant aged 54, who was found to perform similarly to other participants.

English was the first language of 23 participants (66%). Only eight participants (23%) had previous programming knowledge before enrolling in the COSC121 course. I asked students to rate themselves on their programming experience on a Likert scale from one (less experienced) to ten (more experienced), and the average rating was 3.43 (sd = 1.9). I also asked students if they have previously used any other Python programming tutor which is not necessarily on a mobile device, and only five participants (14%) said that they have used one before.

Sixteen participants (46%) owned Android smartphones, 16 participants (46%) owned iOS smartphones, and one participant owned both an Android smartphone and an iOS smartphone. Interestingly, two participants said that they did not own a smartphone. For those 33 participants who owned smartphones, I asked the average daily time that participants spend on their devices; 16 participants (48.5%) claimed to use their smartphones for 3–8 hours, 12 learners (36.5%) use their smartphones for 1–3 hours, 3 participants (9%) use their smartphones for less than an hour, and 2 participants (6%) use their smartphones for more than 8 hours.

6.4.2 Method and Materials

Each student participated in a single session. The sessions were 2 hours long, with 1–9 participants per session. The participants provided informed consent, followed by an 18-min pre-test (Appendix A), which included questions on demographics and programming background. I then gave brief instructions on using the tutor and provided Android smartphones which were mostly running on Android 6.0 (Marshmallow) and had screen sizes of 5–5.2 in with resolutions of at least 720 × 1280 pixels. Some phones had higher screen resolutions, but this was unlikely to have had any effects since the interface elements were implemented to scale relative to the screen resolution. The Android phones given to the participants already had PyKinetic_DbgOut installed. Participants interacted with the tutor for roughly an hour. Lastly,

participants were given an 18-minute post-test (Appendix A), which included open-ended questions for comments and suggestions about the tutor (Appendix B). The post-test was given either when time had run out, or when a participant had finished all problems. There were two tests of comparable complexity that were alternatively given as the pre-test for half of the participants. The study was approved by the Human Ethics Committee of the University of Canterbury (Appendix F).

The topics covered in the study have previously been covered in the lectures of the introductory programming course (COSC121). The pre/post-tests had six questions each and were administered on paper. The tests contained same types of questions in PyKinetic_DbgOut worth one mark each: code reading (Dbg_Read), identifying erroneous LOCs (Dbg_Ident), fixing erroneous LOCs (Dbg_Fix), prediction of actual output (Out_Act), and prediction of expected output (Out_Exp). Additionally, a code-writing question worth five marks was given. The maximum mark for both tests is ten marks. The participants were not accustomed to doing any programming exercises on paper, because all lab quizzes and assessment in the course are completed using computers. Therefore, the code syntax on their pre/post-test were not strictly penalised. 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 questions had an ideal solution of five LOCs (without any comments), which was the reason for a maximum of five marks on this question. The participants did not receive scores for the problems completed in the tutor.

6.4.3 Data Collection

Logs were recorded for all activities. Apart from recording logs for analysis, they were also used for debugging purposes. Every user action (button press, single tap, and long tap) triggers additional data to the log: a timestamp with details about the interaction. Feedback received by the student based on their last solution was also logged, which was useful for identifying how close a student was to completing a problem where they failed to do so. Lastly, the total time taken per problem was stored in the logs.

Evaluation logs were sent through the server and (in parallel) recorded directly on the smartphone device. The logs were gathered while the user was using the tutor. A log file was sent and recorded to the device when: a learner submits a solution, navigates away from an application screen, and every three minutes. I saved the logs every three minutes to have a reasonable amount of data upon recording the log, and at the same time circumvent data loss. Other than that, three minutes was not a “magic number”, rather my educated estimate based on experience from previous evaluation studies.

6.5 Findings

I present the results from the 35 participants who completed the study in Table 6.5. There were no significant improvements from the pre-test to post-test scores. The problems in PyKinetic_DbgOut are of different nature to the problems the participants were used to in the course, where they were mostly asked to write code. For that reason, I investigated whether there was a difference between the participants based on their code-writing skills. Before the study, the participants were assessed in a lab test, which consisted of 20 code-writing questions. The median score on the lab test was 79%. I therefore divided the participants post-hoc into two groups based on the lab test median. The median split used in classifying the prior knowledge of students is like in work by (Adams et al., 2014; McLaren, Adams, and Mayer, 2015). I refer to the 16 participants who scored less than 79% as lower prior knowledge students (LP), and to the 19 participants who scored 79% or higher as the higher prior knowledge students (HP).

Table 6.5 Pre/post-test scores (%)

Question	Pre-test% (sd)	Post-test%(sd)
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)

6.5.1 LP vs. HP Participants

Table 6.6 presents the pre- and post-test results for LP and HP students. I used non-parametric statistical tests to compare the results, as the data was not distributed normally. The LP and HP students spent comparable time on solving each problem. However, HP students outperformed LP by completing more problems ($U = 35$, $p = .037$), and by getting higher pre/post-test scores. Although the overall pre-test score was significantly different for the two subgroups of students, there was no significant difference on the score for the code-writing question alone. Furthermore, only HP students improved their score on the code writing question ($W= 75$, $p = .039$). On the other hand, I did not find a significant improvement for LP on the code writing question. Lastly, it seemed that output prediction questions were unfavourable for the learning of HP students, since HP students had a significantly higher score than the LP on the pre-test but no significant differences between LP and HP students in the post-test.

I calculated the Spearman's correlations between the pre- and post-test scores of LP students. LP had a significant positive correlation on pre- and post-test scores on all questions (r_s = .53, p = .033). However, LP had a significant negative correlation on their pre- and post-test scores on output prediction questions (r_s = − .53, p = .036). Furthermore, I found no significant correlation between the pre- and post-test scores of LPs on debugging questions. There were no significant correlations between pre- and post-test scores for HP students.

Table 6.6 LP vs. HP Students (* denotes significance at the .05 level)

Measure	**LP (16)**	**HP (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	
Normalised gain	.09 (.53)	.30 (.46)	ns
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	W = 75, 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 Output Questions	ns	W = 10, p = .022*	

Table 6.7. LP vs. HP Students Performance Measures (* denotes significance at the .05 level)

	Identifying Error	**Output Prediction**	**Code Fixing**
Attempts			
LP	5.33 (2.98)	3.26 (.51)	8.07 (4.02)
HP	4.21 (.94)	2.63 (.28)	5.37 (1.52)
	ns	p = .000, U = 256*	p = .002, U = 243*
Time per activity (min)			
LP	2.01 (0.69)	1.48 (0.63)	1.63 (0.82)
HP	2.23 (0.43)	1.30 (0.30)	1.59 (0.47)
	ns	ns	ns
Time per attempt			
LP	0.87 (0.47)	0.56 (0.27)	0.25 (0.14)
HP	0.99 (0.32)	0.56 (0.15)	0.38 (0.14)
	ns	ns	p = .003, U = 65*
Score			
LP (16)	0.47 (0.16)	0.63 (0.13)	0.25 (0.16)
HP (19)	0.60 (0.16)	0.75 (0.09)	0.47 (0.15)
	p = .020, U = 82.5*	p = .006, U = 70.5*	p = .000, U = 40*

I present a comparison of the performance measures of LP and HP participants within PyKinetic_DbgOut in Table 6.7. As some problems contain multiple activities, I report the performance measures for each type of activities, not per problem. I did not calculate the performance of the participants on the code reading activity (Dbg_Read), since I only provided two of these questions, and I only allowed

participants to attempt these once, as these are true or false questions. The average attempts reported on Table 6.7 were based on the number of submissions on each activity. Like in my other evaluation studies, I calculated the normalised gain using two formulas (Marx and Cummings, 2007). When the learning gain was positive, I calculated the quotient of gain (post-test score – pre-test score) and (100 – pre-test score). However, when the learning gain was negative, I calculated the quotient of gain and the pre-test score.

The LP students made significantly more attempts in comparison to HP students on output prediction and code fixing activities, but not for identifying errors. There was no significant difference on the time spent per activity between two groups of students. The average times per attempt for LP and HP students were similar apart from the average time on the code fixing activity. The score on each activity was calculated based on the number of correct answers in their first attempt. The maximum score for each activity is 1. For example, in an identifying error activity (Dbg_Ident), if there were three errors to be identified, all three should have been identified correctly on the first attempt to get a score of 1. If only two out of three were correct, a score of 0.67 was given, and a score of 0.33 if only one out of three was correct. However, if the activity only had one required answer, then the score given will be either 0 or 1. The HP students outperformed the LP on the average score based on their first attempts, on all three activities in Table 6.7.

I also investigated whether there were any correlations between the scores on the first attempt on an activity and the post-test performance. Table 6.8 presents significant correlations for average scores based on the first attempt on the following activities: identifying error, output prediction, and code fixing. I also present significant correlations between the average scores and the normalised gains. All reported correlations for HP students are moderate to high positive correlations. Results revealed that the average scores of HP students on identifying errors strongly correlated with their scores on output prediction (r_s = 0.68, p = .001). Their scores on identifying errors also showed a moderate positive correlation with their scores on fixing code (r_s = 0.56, p = .014). On the contrary, there were no significant correlations for LP. There were no significant correlations between the average scores on output prediction and average scores on code fixing for both the HP and LP students. Based on the results presented on Table 6.8, HP students benefitted most with identifying error activities as it revealed statistically significant medium to strong positive correlations with their post-test performance on several types of questions: all questions together, code writing, and output prediction questions. Furthermore, I found a medium positive correlation between scores of HP students on output prediction and their normalised gains on output prediction (r_s = 0.55, p = .016). This was unexpected since HP students showed a significant decrease with their scores on pre- to post-test on output prediction questions (Table 6.6).

The results shown in Table 6.8 shows that the disparity between LP and HP students is astoundingly evident. So, I was wondering whether HP students were just better on their first attempts because of higher

prior knowledge. Therefore, I delved deeper and considered average performance on the entire duration of each activity rather than focusing only on first attempts. More specifically, I calculated correlations between the average time per attempt on each activity type with the average normalised gains (Table 6.9). I found significant high positive correlations between time per attempts on identifying error, output prediction, and code fixing. However, there were clear differences between the correlations found in LP and HP students. For LP, time per attempt on identifying error was positively correlated with their time per attempt on output prediction ($r_s = 0.81$, p = .000). Furthermore, their time per attempt on identifying error was also positively correlated with their time per attempt on code fixing ($r_s = 0.70$, p = .002). Finally, their time per attempt on code fixing was also positively correlated with their time per attempt on output prediction ($r_s = 0.67$, p = .004). For HP students, I only found one significant correlation between their time per attempt within the three activities which was between identifying error and output prediction ($r_s = 0.66$, p = .002). Notice that the correlations between time per attempt on identifying error and time per attempt on output prediction were both significant for LP and HP students. However, for LP, I obtained a stronger positive correlation ($r_s = 0.81$, p = .000) compared to HP students ($r_s = 0.66$, p = .002). To sum up, all three activities were positively correlated with each other for LP, but only identifying error and time per attempt was significant for HP students (Figure 6.11).

Table 6.8. Spearman's Correlations between Scores in PyKinetic_DbgOut, Post-test and Normalised Gains (* denotes significance at the .05 level)

	LP (16)	**HP (19)**
Correlation on *score on identifying error* and:		
Normalised gain	ns	$r_s = 0.55$, p = .016*
Normalised gain on output prediction	ns	$r_s = 0.58$, p = .009*
Normalised gain on code writing	ns	$r_s = 0.46$, p = .048*
Post-test code writing	ns	$r_s = 0.59$, p = .008*
Post-test all questions	ns	$r_s = 0.66$, p = .002*
Post-test output prediction	ns	$r_s = 0.55$, p = .015*
Correlation on *score on output prediction* and:		
Score on identifying error	ns	$r_s = 0.68$, p = .001*
Normalised gain on output prediction	ns	$r_s = 0.55$, p = .016*
Post-test output prediction	ns	$r_s = 0.55$, p = .015*
Correlation on *score on code fixing* and:		
Score on identifying error	ns	$r_s = 0.56$, p = .014*

A similar disparity (like reported values in Table 6.8) was revealed between LP and HP students when correlations between time per attempts and normalised gains were calculated (Table 6.9). However, unlike the values reported in Table 6.8, I found one significant correlation for LP between time per attempt on output prediction and normalised gain on output prediction ($r_s = 0.52$, p = .038). On the other hand, for HP students, similar significant correlations are reported on Tables 6.8 and 6.9. However, in Table 6.9, all

significant correlations for HP students were only on their time per attempt on identifying errors. Furthermore, most of the significant correlations were on normalised gains apart from one correlation with post-test output prediction (r_s = 0.48, p = .037).

Table 6.9. Spearman's Correlations between Time per Attempt in PyKinetic_DbgOut and Normalised Gains (* denotes significance at the .05 level)

	LP (16)	**HP (19)**
Correlation on *time per attempt on identifying error* and:		
Time per attempt on output prediction	r_s = 0.81, p = .000*	r_s = 0.66, p = .002*
Normalised gain on all questions	ns	r_s = 0.50, p = .031*
Normalised gain on output prediction	ns	r_s = 0.55, p = .014*
Normalised gain on code writing	ns	r_s = 0.61, p = .005*
Post-test output prediction	ns	r_s = 0.48, p = .037*
Correlation on *time per attempt on output prediction* and:		
Time per attempt on code fixing	r_s = 0.67, p = .004*	ns
Normalised gain on output prediction	r_s = 0.52, p = .038*	ns
Correlation on *time per attempt on code fixing* and:		
Time per attempt on identifying error	r_s = 0.70, p = .002*	ns

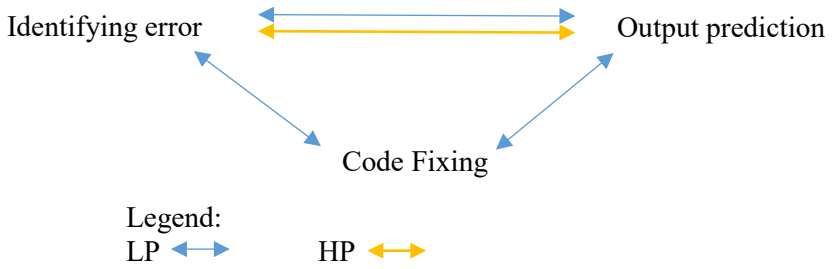


Figure 6.11 Relationship between activities for LP vs. HP students

Since I used both scores and time per attempt to calculate correlations, I also calculated correlations between these measures (Table 6.10). I found related results for HP students as in Table 6.9. Table 6.10 shows significant positive correlations for HP students, but only between activities of output prediction and identifying errors (r_s = 0.51, p = .026) as visualised in Figure 6.11. Furthermore, correlations were found between scores and time per attempt of HP students for both activities of identifying errors (r_s = 0.71, p = .001) and code fixing (r_s = 0.55, p = .015). For LP, values in Table 6.10 also show significant correlations between identifying errors and output prediction (r_s = 0.56, p = .025; r_s = 0.50, p = .049), as well as identifying errors and code fixing (r_s = 0.53, p = .036). Moreover, a correlation was found between score and time per attempt on output prediction (r_s = 0.56, p = .024). There were no correlations found between

code fixing and output prediction for LP when looking at scores and time per attempt, unlike in Table 6.9 (r_s = 0.67, p = .004).

6.5.2 Analyses based on Demographics

I used the Mann-Whitney U test to check differences between the participants based on demographics, focusing on gender, whether English was their first language, and personal smartphone device. There were 11 HP and 12 LP male students, and there were 8 HP and 4 LP female students. When results of male and female students were compared, no measures were significantly different.

I performed the Mann-Whitney U tests between participants who had English as their first language (23) to those who had another language (12). The participants with English as their first language had significantly higher scores on the following: pre-test on expected output prediction (p = .041, U = 79.5), pre-test on all output predictions (p = .037, U = 78.5), post-test scores on code writing (p = .049, U = 81.5), score on code reading in PyKinetic_DbgOut (p = .007, U = 61), score on output prediction in PyKinetic_DbgOut (p = .037, U = 78.5), and score on code fixing in PyKinetic_DbgOut (p = .002, U = 53). Note that all scores within PyKinetic_DbgOut were based on the correctness of their first attempt on each activity. Furthermore, 15 (65%) participants who had English as their first language were HP students, whereas only (33%) 4 participants had another first language who were also HP students.

Table 6.10 Spearman's Correlations between Score and Time per Attempt (* denotes significance at the .05 level)

	LP (16)	**HP (19)**
Correlation on *score on identifying error* and:		
Time per attempt on identifying error	ns	r_s = 0.71, p = .001*
Time per attempt on output prediction	r_s = 0.56, p = .025*	ns
Time per attempt on code fixing	r_s = 0.53, p = .036*	ns
Correlation on *score on output prediction* and:		
Time per attempt on identifying error	r_s = 0.50, p = .049*	r_s = 0.51, p = .026*
Time per attempt on output prediction	r_s = 0.56, p = .024*	Ns
Correlation on *score on code fixing* and:		
Time per attempt on code fixing	ns	r_s = 0.55, p = .015*

When comparing results of participants based on their personal smartphone devices with Android (16) compared to iOS (16) users, participants who used Android devices had a significantly higher pre-test score on code fixing questions (p = .002, U = 48), and pre-test score on debugging questions (p = .002, U = 47.5). The Android users also had a significantly higher score on identifying errors within PyKinetic_DbgOut based on their first attempt (p = .035, U = 72). Contrary to that, Apple users were significantly faster on their average time per problem with an average of 2.49 min (sd = .52) and average of 2.91 min (sd = .67) for Android users (p = .023, U = 68). However, Apple users were not significantly

faster than Android users on their average time per activity. Furthermore, I examined whether there was a correlation on the length of average use of participants with their smartphones with their normalised gain scores, but I did not find any significant correlations.

When comparing LP with HP students, their estimated programming experience was statistically comparable. No significant correlations were found between indicated programming experience by participants and their normalised gains. Also, no significant correlations were found between indicated programming experience by participants and their scores within PyKinetic_DbgOut. However, I found correlations between programming experience as stated by participants and some of their pre-test scores: on code writing ($r_s = 0.36$, $p < .05$).

6.5.3 Comments and Suggestions from Participants

The post-test included a few open-ended questions about PyKinetic_DbgOut. I asked participants whether they would like to use PyKinetic_DbgOut again and to state their reasons (Q1). Three participants did not respond to the question. Twenty-seven participants (77%) said that they would like to use it again (including 1 participant who gave a condition that he/she will only do so if PyKinetic_DbgOut was improved with more feedback), and 5 participants (14%) said that they would not. Some of the participants who refused to use PyKinetic_DbgOut again mentioned that they learn better by writing their own code. However, despite their preference to practice on a personal computer, three out of these five participants showed an improvement in their learning in using PyKinetic_DbgOut based on their scores from pre- to post-test.

The questionnaire also asked participants for any comments or suggestions for PyKinetic_DbgOut (Q2). Since I already asked the participants to state their reasoning (Q1) on wanting (or not) to use PyKinetic_DbgOut again, some participants wrote their comments and suggestions as an answer to (Q1). For this reason, I combined all the comments and suggestions from both (Q1) and (Q2) and classified them. Some participants wrote multiple comments, hence the reason for the numbers below. I classified the comments into following categories:

(C1) PyKinetic_DbgOut was helpful and/or had a good range of content

(C2) PyKinetic_DbgOut is good for debugging and code understanding

(C3) PyKinetic_DbgOut is intuitive and has a good user interface

(C4) Some questions were too difficult.

(C5) PyKinetic_DbgOut is convenient especially when not near a computer

Category C1 contains comments from 23 participants (63%), such as *"It was fun and helpful"* and *"There are very good questions with a good degree of challenge."*. Category C2 contains comments from six participants (17%), examples of which are *"It is good for extra practice and I know how to point out mistakes and fix them after using it."* and *"It was good practice at reading a program and understanding*

what it does." Category C3 contains comments the usability and user interface of the tutor from four participants (11%), such as *"It is quick to pick up and very intuitive"*. Three participants provided negative remarks (C4), who found some of the questions in PyKinetic_DbgOut too difficult. Lastly, in category C5, there are comments from two participants (6%), who noted that PyKinetic_DbgOut is convenient to use when not near a computer. There were some interesting comments. One of the participants commented about the interface: *"I felt the app made good use of the limited screen space."* I also combined the suggestions and classified them as follows:

(S1) Feedback needs to be improved

(S2) Include code writing questions

(S3) Usability can be improved particularly in code fixing

(S4) Implement gamification on PyKinetic_DbgOut

The most common suggestion (S1), made by ten participants, was to improve feedback, such as *"Give some guide notes when people made mistakes."* Seven participants suggested to add some code writing questions (S2), as some of them mentioned that the activities in PyKinetic_DbgOut does not fit their style of learning. Six participants mentioned that the usability of the PyKinetic_DbgOut could be improved particularly for the code fixing activity (S3). Some of them mentioned that they found it easy to accidentally submit a random answer by mistake when fixing LOCs. Finally, three participants suggested to add gamification (S4), specifically to add a scoring and penalty system, as it was easy to cheat via trial and error, specifically on the code fixing activity. A notable suggestion from one participant was the following: *"Potential integration into a lecture environment would be very cool."*. Another participant mentioned a similar suggestion: *"The tutor would be good to use for testing students where small simple tests might be appropriate, such as during lectures when the lecturer may want to ask students to interact with the content."*

6.6 Discussion

I found several differences between participants with lower prior knowledge (LP) and those with higher prior knowledge (HP). I reported differences between their learning improvement, performance in PyKinetic_DbgOut, and relationships between their coding skills. I discuss those differences between LP and HP participants separately in the three subsections to follow. In the last subsection "Analysis with demographics," I discuss findings when participants were grouped based their demographics.

6.6.1 Learning Improvement

My results revealed significant correlations on the pre- and post-test scores of LP students implying that they mostly relied on their prior knowledge to answer the post-test. Significant positive correlations were found on the following: pre- and post-test on all questions, and pre- and post-test on output prediction questions. However, for the HP students, there were no significant correlations on any of the pre- to post-test scores, including pre- and post-test scores on debugging questions. Since most activities in the tutor were mostly composed of debugging activities, I suspect that the reason why only HP students showed significant improvement in learning was because of the hierarchy of coding skills. As discussed in the "Coding skills" section, researchers found that code tracing needs to be mastered before being able to be proficient in code writing (Lopez et al. 2008; Thompson et al. 2008; Harrington and Cheng 2018). Furthermore, Fitzgerald et al., (2008) and Ahmadzadeh et al. (2005) found that debugging code written by someone else requires higher order of knowledge than code writing. Therefore, I have seen evidence of the coding hierarchy of skills in my study. I propose that this may not necessarily mean that the activities were too difficult for LP. Instead, it seems to be that learning with activities on the higher end of hierarchy of programming skills proved to be not beneficial for LP. This is because they are most likely lacking the understanding of the core concepts of programming, and unless these are learned, students may not be able to benefit. Like what Anderson (1982) mentioned, both declarative and procedural knowledge is needed by students.

I presented more evidence showing that HP students learned more than LP with PyKinetic_DbgOut, based on the correlations between their scores in the tutor and their post-tests, and their scores in the tutor when correlated with their normalised gains. The correlations revealed that identifying error activities benefitted the HP students most, as it positively correlated with their post-test scores and normalised gains in various activities. Similar correlations were found when I used scores as a measure, and time per attempt. It appeared that HP students performed worse on pre- to post-test on output prediction questions. I propose that the learning benefit was most likely not reflected on the post-test due to the ceiling effect on output prediction problems (pre-test avg. = 97.37%, sd = 7.88). The score of HP students for identifying errors was also correlated with their normalised gain for all questions and their normalised gain on code writing. These results are consistent with the literature as (Ahmadzadeh et al. 2005) found that most learners with good debugging skills are also advanced programmers. In this study, I consider HP students as "advanced programmers" in comparison to LP students.

6.6.2 Performance in PyKinetic_DbgOut

The differences between the performance of LP and HP students in PyKinetic_DbgOut were interesting. Firstly, one might expect that HP students will be faster than LP, but I found that there were no significant differences with the average time they spent in the three activities: identifying errors, fixing errors, and output prediction. It might have been because, regardless of their abilities, the participants were not accustomed to doing activities in PyKinetic_DbgOut in their course, as they were mostly doing code writing. However, the HP students outperformed the LP when I calculated their scores based on their first attempts. Furthermore, LP had significantly more attempts on output prediction and code fixing activities. This leads me to postulate that some LP students might have used trial and error strategies. Furthermore, a trial and error strategy are most likely being utilised more in the code fixing activity as it was revealed that LP students had a significantly lower time per attempt than HP students in this activity. This result suggests that they completed the code fixing activities with more attempts but with less time, in comparison with the HP students. Furthermore, although they did not admit to using a trial and error strategy, some participants mentioned in their suggestions that I should improve the usability specifically for the code fixing activity. Participants found that the controls in the code fixing activities made it easy to use a trial and error strategy. Therefore, I should improve the tutor by providing support for optimal strategies, to prevent learners from guessing.

6.6.3 Relationships Between Coding Skills

I presented correlations between time per attempt in PyKinetic_DbgOut and normalised gains for both the LP and HP students. Bearing in mind that learners were not allowed to skip any activity, the average time per attempt is indicative of the performance of the learners. I showed evidence that the performance of LP on debugging code is positively correlated with their performance on tracing code. Moreover, their performance on code tracing is also positively correlated with their performance on fixing code. Lastly, their performance on fixing code is also positively correlated with their performance in debugging. All these are significant strong correlations, showing evidence that students with lower prior knowledge perform similarly across various coding activities of debugging, tracing and fixing code. However, I did not yield the same result between code tracing and code fixing when I considered both score and time per attempt. However, I think that this might be related to the possible issue of controls in code fixing activities mentioned by the participants (Section 6.5.3). Therefore, I have reasons to accept that my illustration in Figure 6.11 still holds. My results showing relationships between skills is consistent with literature as Harrington and Cheng (2018) also found indications that underachieving students usually have a large gap between coding skills as they are most likely struggling in core programming concepts. Core programming

concepts are essential in learning coding skills of debugging, tracing, and fixing. I see further confirmation of this when I looked at correlations between the same measures for students with higher prior knowledge. For HP students, only their time per attempt on debugging revealed a significant correlation with code tracing. The performance of HP students in identifying errors showed a strong positive correlation with their performance on output prediction, which demonstrated the relationship between debugging and code tracing skills. Similarly, a moderate positive correlation was shown between debugging and code tracing when I had taken both scores and time per attempts into consideration. My results demonstrated how various coding skills are interrelated and how it differs between students of varying prior knowledge.

6.6.4 Analysis with Demographics

I also analysed various subgroups of participants based on demographics. I did not find any significant result when comparing male and female students, most likely because of the lower proportion of female students. When I compared participants based on whether English was their first language, it seemed like participants who had English as their first language performed better than the other students. However, it was probably due to the English group having more HP learners compared to the other group. I also compared the participants based on their personal devices, Android users compared to iOS users. Android users achieved a significantly higher score on identifying errors within PyKinetic_DbgOut but this was most likely because Android users had a significantly higher pre-test score on debugging questions. Lastly, one might expect regardless of the smartphone that they are using, learners who often use their smartphones would learn more in using a mobile tutor such as PyKinetic_DbgOut. However, I found that was not the case. This might be an indication that PyKinetic_DbgOut can be useful to a learner despite their limited smartphone experience. I also did not find any significant correlations on their average use of their smartphones with their scores within PyKinetic_DbgOut.

6.7 Conclusions

I presented a controlled lab study which investigated the effectiveness of activities in PyKinetic_DbgOut and my findings from the study. I had three research questions for this study and two hypotheses. Firstly, I investigated whether the combination of coding activities is effective for learning programming (Study2_R1). My hypothesis (H1) for this research question was that the combination of coding activities will be effective for learning programming. However, I found that the group learned from pre- to post-test, but the learning gain was not statistically significant. So, I did a post-hoc division of the participants by their median scores from their course lab test, as I believe this is gives a better representation of their knowledge in programming. Participants who scored less than the median were labelled as LP (16), and the

rest as HP students (19). When I repeated the analysis, I found that only HP students showed significant learning improvement. Therefore, I found enough evidence to answer (H1) and found that the combination of coding activities was effective, but primarily for HP students.

In my second research question (Study2_R2), I explored how the activities affected LP and HP students. My hypothesis (H2) in conjunction with this research question was that lower prior knowledge (LP) students would have higher learning gains in comparison to (HP) students. I did not have any hypothesis on other effects to LP and HP students which are not related to learning gains. As I mentioned (Study2_R1) and (H1), there was evidence of learning benefits in using PyKinetic_DbgOut. However, I did not expect this to only be significant for HP students. Therefore, there was enough evidence to reject (H2), since HP students revealed higher learning gains than LP students.

To further discuss Study2_R2, I reported evidence that for LP, all three activities (identifying errors, output prediction, and code fixing) are interrelated with each other. I showed that these activities were positively correlated with each other, showing evidence that if a LP learner is proficient at identifying errors, he/she will also be skilful in output prediction, and fixing code. This can also be interpreted that debugging, code tracing, and a subset of code writing skills are interrelated for LP. I found that although their performance on the activities seem to be equally associated, LP did not show any evidence that it was correlated to their learning gains. On the other hand, HP students showed evidence that only their performance on identifying errors and output prediction was positively correlated. Therefore, debugging and code tracing skills are interrelated for HP students. Furthermore, I found evidence that the performance of HP students in PyKinetic_DbgOut was correlated with their normalised gains. More importantly, activities on debugging (identifying errors) seemed to be most beneficial for HP students. I presented evidence that their performance on those activities were positively correlated with their normalised gain for all questions, and for specific activities: output prediction, and code writing. Therefore, one of my contributions is that I confirmed results from the literature that there is a hierarchy in programming skills; evident even when learning via a mobile tutor. Firstly, as mentioned by Anderson (1982), learners need both declarative and procedural knowledge. Specifically, code tracing needs to be learned before writing code (Lopez et al. 2008; Thompson et al. 2008; Harrington and Cheng 2018), and debugging someone else's program requires higher order of skill than code writing (Ahmadzadeh et al. 2005). I reported evidence which supports work of Ahmadzadeh et al. (2005). My second contribution is that I also confirmed that debugging someone else's programs is beneficial for HP students. Moreover, another contribution is that I have provided evidence showing that programming skills can also be improved by using a mobile phone, even when (in my study) they only learned with PyKinetic_DbgOut for an hour. I also reported sufficient evidence that the same programming hierarchy of skills applies when learning through a mobile.

I also asked the research question of how I can improve the usability of PyKinetic_DbgOut

(Study2_R3). Firstly, I found that the daily smartphone usage of the participants was not correlated with their learning gains in PyKinetic_DbgOut. Furthermore, their smartphone experience was also not correlated with their scores within the app. This could be an indication that the tutor already has a user-friendly interface since I did not find any evidence that their performance and learning gains are dependent on their experience with smartphones. Moreover, I asked the participants a few questions on whether they would use it again and asked for comments and suggestions. Overall, I received a positive response from the participants. Four participants complimented PyKinetic_DbgOut that it was intuitive and easy to use. Furthermore, 63% (22) of the participants said that the tutor was helpful and/or had a good range of content. However, three participants said that they found some of the questions too difficult.

I also found helpful suggestions to improve PyKinetic. The most popular suggestion was that the feedback needs to be improved. I expected this, as I had time limitations for implementation which only allowed us to provide one pre-determined feedback for each activity. The second most popular suggestion from six participants was to include code writing questions. Although work was done in providing code writing exercises on a phone (Mbogo et al. 2016), the evaluation presented qualitative results without investigating learning effects. Therefore, I am not convinced that it can be effective for learning in a smartphone. More work needs to be done in this area. Furthermore, six participants also remarked that the code fixing exercise was easy to cheat, and it was easy to submit an answer by mistake. This was an unexpected suggestion and will be taken it into consideration for improvement. Enhancing the code fixing activity is relevant to the last suggestion which was to implement game features in PyKinetic. If I add game features, I can for example add limitations on the number of tries when solving a code fixing activity. The limitations of this study include the small set of participants, and limited feedback provided by the tutor. Another limitation is that I found that the activities for predicting expected output (Out_Exp) was not ideal since a learner can easily guess the expected output mostly from reading the problem description.

Results from my first evaluation study (Chapter 5) and the findings in the study covered in this chapter, allowed me to discover different benefits that can be acquired for certain activities. Parsons problems are more suitable for LP students (Chapter 5), while debugging and output prediction activities are more beneficial for HP students (Chapter 6). Furthermore, my results from my first and second evaluation studies (Chapters 5-6) specifically on the difference between learners with lower and higher prior knowledge enabled us to develop a version of PyKinetic with adaptive problem selection (Chapter 7). The latest versions of PyKinetic are covered in the next chapter (Chapter 7) which both consist of a combination of activities used in previous studies. Chapter 7 covers my last evaluation study encapsulating the entire research project.

7 THIRD EVALUATION STUDY

This chapter covers my third evaluation study and addresses research questions (R1) and (R3-R7). Similar to what I mentioned in other chapters the other of the research questions above were not fully addressed solely in this chapter, rather were addressed collectively in all my evaluation studies. However, since this was my last evaluation study, I acquired more evidence to answer research questions (R3-R7) towards drawing conclusions for my entire research project (Chapter 8).

For (R4), I evaluated two versions of PyKinetic (PyKinetic_Fixed and PyKinetic_Adaptive) with a different combination and sequence of activities as an improvement from Chapter 6 (PyKinetic_DbgOut). PyKinetic_Fixed and PyKinetic_Adaptive contained two variants of Parsons problems, debugging, and output prediction exercises. The versions were identical apart from the personalised problem selection offered in PyKinetic_Adaptive. For (R5), I investigated two other pedagogical strategies:

1) the combination and sequence of activities in PyKinetic_Fixed and PyKinetic_Adaptive;

2) adaptive problem selection in PyKinetic_Adaptive.

This chapter starts by presenting the activities implemented in PyKinetic_Fixed and PyKinetic_Adaptive. Next, I present both versions which consists of the activities as discussed. Following that, the architecture and development of both versions are discussed. After that, details about the evaluation study are presented: experimental design and findings. Lastly, the discussion and conclusions are presented.

7.1 Activities

The problems in both PyKinetic_Fixed and PyKinetic_Adaptive consist of the problem description, code containing 0–3 incorrect LOCs. Both versions of PyKinetic contain 1–3 activities for each problem. Furthermore, there are five types of activities offered (Table 7.1): two variants of Parsons problems, one output prediction activity, and two types of debugging activities.

The simplest activity in PyKinetic_Fixed and PyKinetic_Adaptive is a regular Parsons problem (*Reg_Pars*), which is the version of Parsons problems used in the pilot study (Chapter 4) but without distractors (extra LOCs). Reg_Pars contains correct and complete LOCs, which only require the LOCs to be reordered to produce the expected outcome. LOCs can be reordered by dragging and dropping using the drag handle icons on the left of each LOC (Figure 7.1, right).

Table 7.1 Five Types of Activities in PyKinetic_Fixed and PyKinetic_Adaptive

Type of Activity	Task	Additional Information Given
Reg_Pars	Regular Parsons Problem	Test cases with actual output
Pars_Inc	Parsons Problem with incomplete LOCs and SE prompt (Pars_Inc)	Expected output
Dbg	Identify *n* erroneous LOCs (*n* is given)	Test cases with actual output
Fix	Fix erroneous LOCs (by tapping through given choices)	Test cases with and without actual output depending on problem level (See Section 7.2)
Out	Select actual output of the code	Test cases

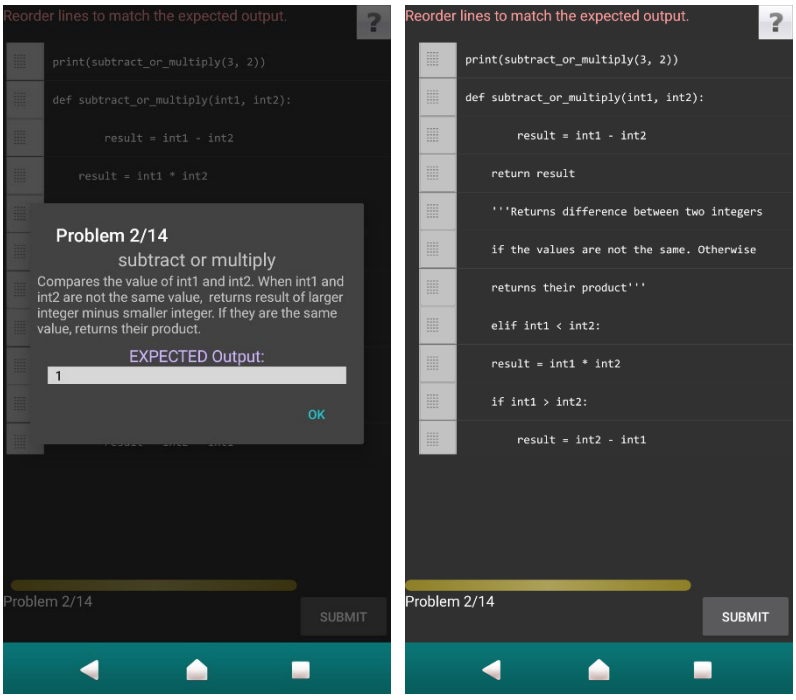


Figure 7.1 Example of a Regular Parsons problem (Reg_Pars); problem details (left) and Parsons problem screen (right)

My regular Parsons problems had similar characteristics with my Parsons problems with incomplete LOCs in Chapter 5 but without menu-based self-explanation (SE) prompts. Firstly, in my design students solve Parsons problems in the same area where the LOCs are presented. So, there is no separation at all between the blocks of code in the problem and the student's solution. Just like in the example shown in Figure 7.1 (right), learners solve the problem in the same area where they receive all the randomised LOCs. I propose that combining the two areas makes better use of the limited screen resource in smartphones. Secondly, in my Parsons problems all lines are provided with the correct indentation which are necessary to mark the start and end statements in Python. Lastly, all blocks of code contain only single LOCs, similar to work by Garner (2007) and Kumar (2018). Moving blocks of LOCs could potentially prevent the student

from thinking about individual LOCs; for that reason, I require the student to move single LOCs.

In the example in Figure 7.1, the learner is working on a problem about conditional statements. The learner has not finished the problem since the LOCs are still ordered incorrectly, just by observing the indentations of the LOCs as these marks the start and end of statements in Python. The problem contains a function `subtract_or_multiply`, which takes two integers (`int1` and `int2`) as parameters. If `int1` and `int2` have the same value, the function returns the product of the two. Otherwise when one of the integers is larger than the other, the function returns the difference between the larger and the smaller integer. Compared to my Reg_Pars activities used in the pilot study (Chapter 4), the Reg_Pars in PyKinetic_Fixed and PyKinetic_Adaptive were improved with additional feedback. On the learners' first attempt, if the solution is incorrect, the tutor gives simple feedback. For Reg_Pars with more than one test case, the tutor will first check whether the test cases are ordered correctly based on the expected output. If the test cases are ordered correctly, but the rest of the LOCs were not, the feedback is "*Check the order of your solution.*" But, if the test cases are ordered incorrectly, the feedback given is "*Test cases must also be reordered to match the expected output.*" to alert the learner that the order of the test cases are also checked. For Reg_Pars activities with only one test case (like in Figure 7.1), on a first incorrect attempt the learner receives "*Check the order of your solution.*". If the learner submits an incorrect solution for the second time, a hint is provided regardless of the number of test cases in the activity. For example, the hint given for the Reg_Pars activity displayed in Figure 7.1 was *"Remember elif always comes after if."*, to remind the learners of the order in which conditional statements are written. This hint is an example of a conceptual hint, to remind learners about Python syntax and other theoretical knowledge. Another example of a conceptual hint given for a different Reg_Pars activity (not shown in the figure) was "*If, elif, else statements can be nested i.e. contain another 'if' within."* Although Reg_Pars activities were simple as correct syntax and indentations were provided, it was not always enough to have conceptual knowledge to solve them. Some learners have good conceptual knowledge but need improvement on their code understanding and code tracing skills. Therefore, to aid learners further, I also provided procedural hints which provided help about certain execution points in the code. Two examples of procedural hints for different Reg_Pars activities (also not shown in the figure) were: "*We only include minutes left when there is extra time*." and "*Variable result should be initialised first.*" From the third subsequent incorrect attempts, feedback toggles between simple feedback and a hint. For each problem, there was only one predefined hint that could be either a conceptual or procedural hint. When the learner successfully rearranges the LOCs in the correct order, PyKinetic_Fixed and PyKinetic_Adaptive provides the following feedback: "*Correct! Great job!*".

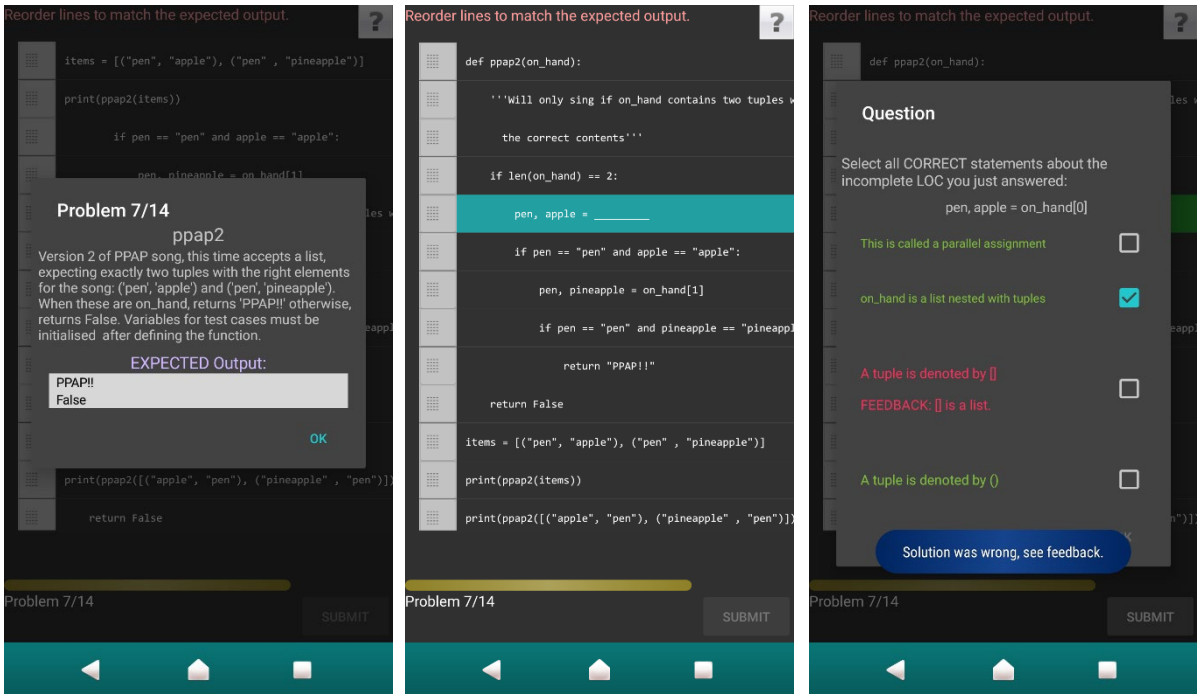


Figure 7.2 A Parsons problem with one incomplete LOC and menu-based SE prompt. Left: problem details and expected output; Middle: problem solving screen with the incomplete LOC highlighted in turquoise as it is selected by the learner; Right: menu-based SE prompt (answered incorrectly), the SE prompt was given when the incomplete LOC was solved.

A second activity in PyKinetic_Fixed and PyKinetic_Adaptive was another variant of Parsons problems – Parsons problems with incomplete LOCs and menu-based SE prompts (Pars_Inc). Pars_Inc was first used in my first evaluation study (Chapter 5). Like in Reg_Pars, my design of Pars_Inc differs in three ways compared to other common implementations: 1) one area was utilised for receiving blocks of code for the Parsons problem and as a workspace to rearrange the blocks; 2) indentations were provided for scaffolding 3) all blocks of code contained single LOCs. For Pars_Inc, there was a fourth key difference compared to other work on Parsons problems: 4) my incomplete LOCs were accompanied by menu-based SE prompts (Figure 7.2, right) that learners were required to answer upon successful completion of an incomplete LOC such as the LOC highlighted in Figure 7.2 (middle). To the best of our knowledge, I was the first one to integrate Parsons problems with SE prompts.

An example of Pars_Inc is illustrated in Figure 7.2. Initially, the student was given the description of the problem and the expected output (Figure 7.2, left). The problem covered conditional statements, tuples, and lists. The function `ppap2` has one parameter `on_hand` which is a list which contains two tuples. `ppap2` checks whether `on_hand` contains the correct values for the song (`'pen', 'apple'`) and (`'pen', 'pineapple'`) and returns a string "`PPAP!!`" if the values are correct. When `on_hand` contains incorrect values for the song, function `ppap2` returns a boolean value of False. The learner already rearranged the LOCs correctly in Figure 7.2 (middle) but was still tasked to complete the incomplete LOC (highlighted in

turquoise). Each Pars_Inc problem contained up to three incomplete lines. The example provided in Figure 7.2, contains only one incomplete LOC. An incomplete LOC contains a blank space, which may require one or more keywords. A longer blank line represents more than one keyword just like the Pars_Inc displayed in Figure 7.2 (middle). In the example shown in this screenshot, the student selected the incomplete LOC to work on, acknowledged by the tutor by highlighting it in turquoise. The learner next selects the correct choice to complete the LOC from a set of provided choices, by tapping between alternatives instead of typing, like (Ihantola, Helminen and Karavirta 2013). The choices provided were:

a) `on_hand`

b) `on_hand[0]`

c) `on_hand[len]`

d) `len(on_hand)`

The blank space is medium in length, indicating that the correct solution may contain more than one keywords. The correct answer was b) `on_hand[0]`, so the complete LOC was `pen, apple = on_hand[0]`. Another example of an incomplete LOC for another Pars_Inc was `while count ____:`. Notice that the blank line is shorter, indicating that fewer (or shorter) keywords are required. The choices given for this incomplete LOC is

a) `> 0`

b) `>= 0`

c) `<= 0`

d) `!=0`

The correct answer was a) `> 0`, so the complete LOC is `while count > 0:`. When the learner selects the correct keyword/s for an incomplete LOC and finalises their answer by long tapping on the LOC, PyKinetic_Fixed and PyKinetic_Adaptive recognise the correct choice by highlighting the line in green. An incorrect solution would yield a feedback *"[choice] is incorrect"*, and the line is highlighted in red. No hint was provided when solving incomplete LOCs.

In Figure 7.2 (right), the learner eventually completed the LOC and was given the SE prompt which corresponds to the line just completed (Figure 7.2, right). I still used menu-based SE prompts, which were proven to be effective in my previous study (Chapter 5). Learners were only allowed to attempt the SE prompts once. Moreover, learners were still not allowed to skip a SE prompt (just like in Chapter 5) because as reported by Aleven and Koedinger (2000) students did not always follow through their tutor's SE prompts. However, I improved my SE prompts by only providing positively phrased questions like illustrated in (Figure 7.2, right) and targeted questions. Examples of targeted questions were "*What is split()?*" and "*Why did we need juice < capacity in the incomplete LOC you just answered?*". I decided to not include negatively phrased questions because learners from my previous study (Chapter 5) experienced

confusion when I provided a mix of positively phrased and negatively phrased questions in my SE prompts.

Upon submitting a solution for a SE prompt, PyKinetic_Fixed and PyKinetic_Adaptive show the wrong options in red and correct in green regardless of the accuracy of the solution. In the illustration in Figure 7.2 (right), the learner was asked to select all the correct statements concerning the completed LOC. In the example in the same figure, the learner selected the second option which was one of the correct options but was insufficient since the question asks to select all correct statements. Therefore, the tutor provided additional feedback for the wrong choice.

Like in my first implementation of Pars_Inc (Chapter 5), learners can only receive feedback about the order of the LOCs in the problem by submitting their solution through single tapping the Submit button. However, the Submit button was only activated once all incomplete LOCs are solved. For instance, in the example in Figure 7.2 (middle), although the learner has correctly rearranged the LOCs, he/she is not allowed to check this until the incomplete LOC is solved (highlighted in turquoise). Nevertheless, the feedback about the order of LOCs were improved in a similar manner as Reg_Pars. For Pars_Inc with one test case, the simple feedback provided for an incorrect solution was "*Check the order of your solution".* On the other hand, for Pars_Inc with more than one test case, the tutor will first check whether the test cases are ordered correctly based on the expected output. If the test cases are ordered incorrectly the feedback given is "*Test cases must also be reordered to match the expected output.*". The tutor first gives simple feedback on the learners' first attempt and provides a hint on the second time that the learner submits an incorrect solution. On subsequent incorrect attempts, feedback toggles between simple feedback and a hint. Like in Reg_Pars, Pars_Inc provided conceptual and procedural hints. Examples of conceptual hints are: "*title() or capitalize() changes the first letter of strings to upper case*" and "*for loops can be nested*". For procedural hints, examples are: "*The counter is decreasing, so the while condition should be?*" and "*We can't cook any eggs if time given is less than 3 minutes.*". When the learner successfully rearranges the LOCs in the correct order, PyKinetic_Fixed and PyKinetic_Adaptive provides the following feedback: "*Correct! Great job!*".

The third type of activity is a debugging exercise (Dbg), where the learner was provided with a problem description, a code snippet with error(s), and test cases together with the actual output for each test case. The learner's task was to identify one to three incorrect LOCs per problem. The tutor displayed how many erroneous LOCs existed in the code. The student identifies the LOC(s) with errors by long tapping on the LOC. An example is shown in Figure 7.3 where the learner first received the problem description (Figure 7.3, left), followed by the code in the problem-solving screen (Figure 7.3, middle). In this problem, function `name_items` is provided which takes two string arguments: `my_name` and `item`, and returns a string. `name_items` expects a name value for variable `my_name` and an object for variable `item` to concatenate the two strings together and show possessive ownership of the item (i.e. Joe's coffee). The function capitalises

the first letter of `my_name` and changes the rest of the string into lower case, then it adds "'s" to show possessive ownership of the item variable. However, the function returns a string **"unknown"** if variable item contains numbers, and if the item is **"unknown"**. The learner selected one of the LOCs as erroneous and submitted their solution, but this LOC (highlighted in turquoise in Figure 7.3, middle) was without error. Therefore, the tutor showed feedback and, in this case, provided a hint. The tutor reminded the learner that `isalpha()` is used to check if a string only contains letters. The hint was given to probe the learner into checking the if statement in the code, where `isdigit()` was used instead. Therefore, the erroneous LOC was `if item != result and item.isdigit()`. The learner also gets a hint from long tapping on the test case to see the actual output when the test case is executed (Figure 7.3, right). As seen in Figure 7.3 (right), the learner should notice that the function was returning "unknown" which should not be the case since the test case arguments does not have any digits (specifically for variable `item`).

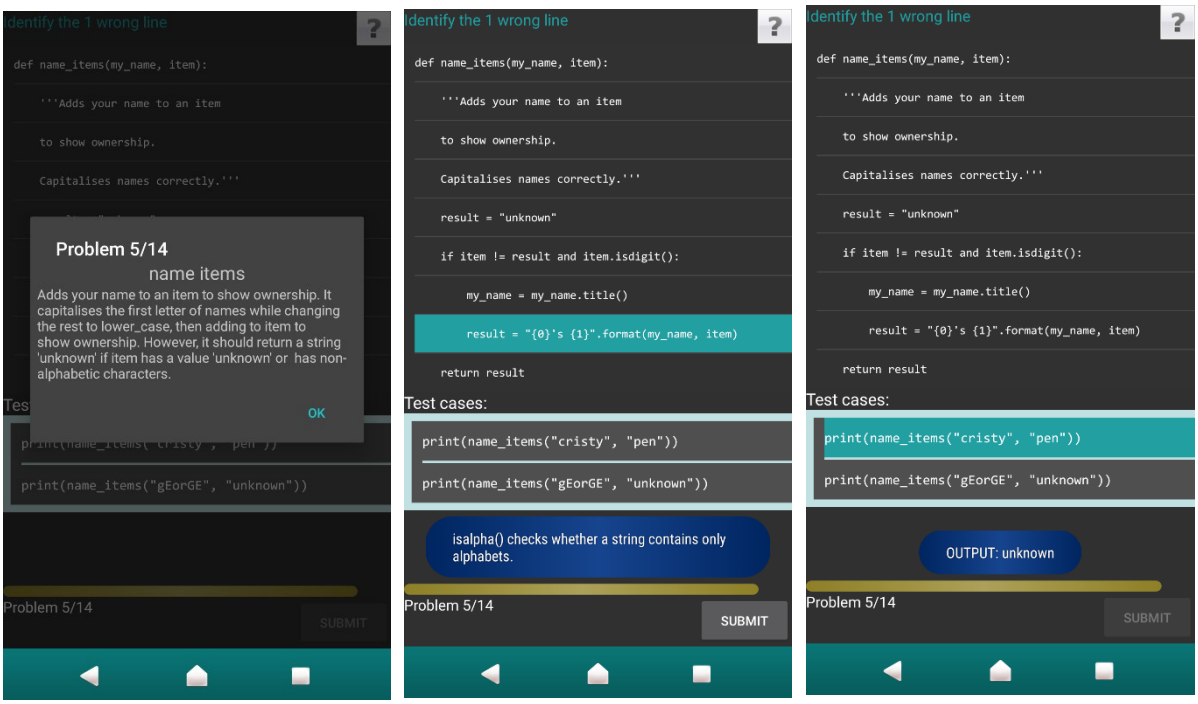


Figure 7.3 Example of Dbg activity; left: problem details; middle: problem with one erroneous LOC; right: same problem where one of the testcases is highlighted because the learner long tapped on it to check its output (OUTPUT: unknown)

Feedback was only given on the student's solution when a learner clicks the Submit button. If the solution is correct, the student receives the positive feedback "*Correct! Great job!"* (same as other activities). When the solution is incorrect, the given feedback depends on the circumstances. Firstly, PyKinetic checks whether the learner selected *n* LOCs (as indicated in the problem). Feedback was also given if the student selected too many or too few incorrect LOCs. If the learner selected exactly *n* lines, but the selections were all incorrect, simple feedback was given "*Sorry, solution was incorrect.*". Moreover, simple feedback of "*Almost there! You are partially correct.*" is given when the selections were partially correct. On a learner's second incorrect try, a hint is given (like in Figure 7.3, middle). Subsequent incorrect attempts result in alternating simple feedback and hint. Similar to my Parsons problems activities, I also provided conceptual and procedural hints for Dbg activities. The example shown in Figure 7.3 (middle) is a conceptual hint. Another example of a conceptual hint is "*The multiplication operator in Python is '*'.*". Examples of procedural hints are: "*Only perfectly cooked pancakes are counted, i.e. half-done pancake doesn't count.*" and "*Which variable contains the number of pancakes?*".

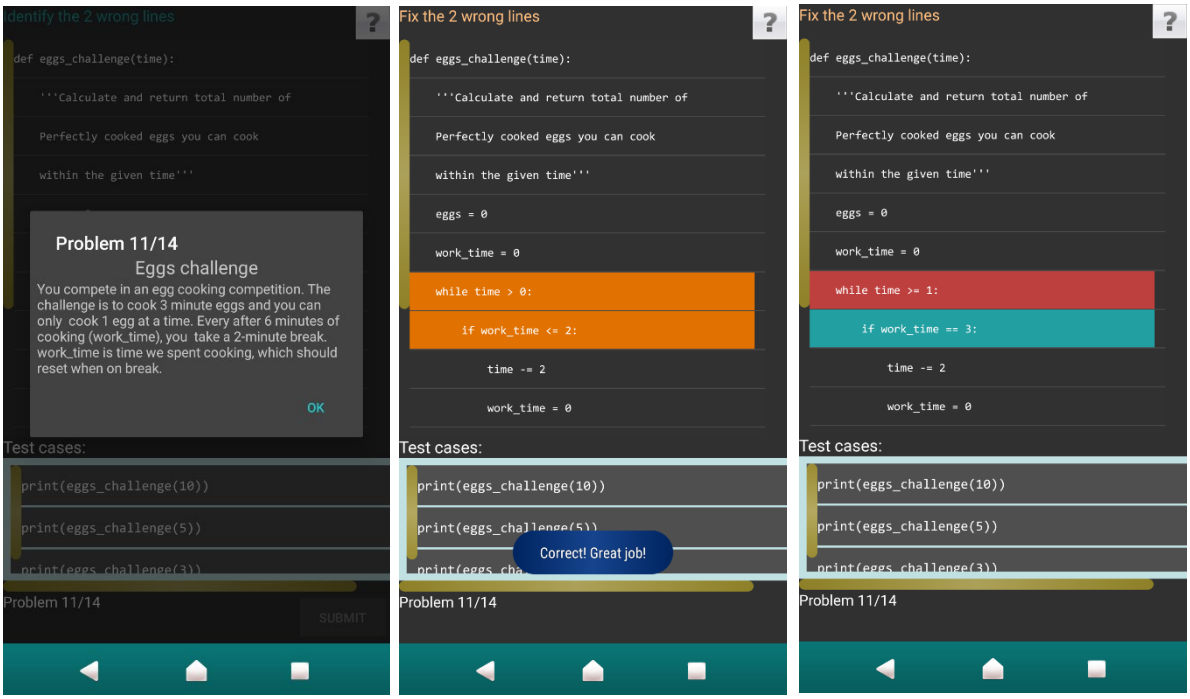


Figure 7.4 Example of Fix activity; left: problem details; middle: problem with two LOCs to fix (highlighted in orange); right: same problem but the student answered the one of the LOCs incorrectly (highlighted in red), and is working on the second one (highlighted in turquoise)

The fourth type of activity (Fix) was also a debugging exercise where the student is asked to fix up to three erroneous LOCs. Fix activity always starts with a Dbg activity, therefore starts with requiring the student to identify incorrect lines (Dbg), and then requires students to fix them (Fix). In Figure 7.4 (middle) the student just finished the Dbg activity and received the feedback *"Correct! Great job!"*. Also, LOCs

that require fixing were highlighted in orange (Figure 7.4, middle). The problem contains function `eggs_challenge` which has an integer parameter `time`. As mentioned in the problem description (Figure 7.4, left), there is a cooking competition where 3-minute eggs are cooked one at a time, and the contestant takes a 2-minute break every six minutes. `eggs_challenge` calculates and returns the number of eggs cooked based on the value of variable `time`. The actual output was given in each test case which was 0 for all test cases. For the first erroneous LOC (Figure 7.4 right highlighted in red), the choices given were:

a) `while time > 0:`

b) `while time >= 1:`

c) `while eggs < 0:`

d) `while time >= 3:`

In this example, the learner answered incorrectly so the tutor highlighted the LOC in red. In this LOC, the correct answer was d) `while time >= 3:` since eggs should only be cooked when there is at least 3 minutes left. In Figure 7.4 (right), the learner did not want to work on the first LOC yet, instead was working on the second LOC he/she selected as indicated by the turquoise highlight. For the second LOC, the alternatives provided were

a) `if work_time <= 2:`

b) `if work_time == 3:`

c) `if work_time == 6:`

d) `if work_time > 1:`.

The correct answer was c) because as mentioned in the problem description, a break is taken every six minutes. To fix incorrect LOCs, the student first selects a LOC to fix by long tapping on an erroneous LOC. Then, the student needs to select the correct option by tapping on the LOC to see alternative choices and finalises their choice by long tapping on the LOC, similar when solving incomplete LOCs for the Pars_Inc activity. If the learner successfully fixes a LOC, the line is highlighted in green and the same feedback of *"Correct! Great job!"* is given. For incorrect attempts, the line is highlighted in red and *"[choice] is incorrect"* is displayed. Also similar to completing LOCs in Pars_Inc, no hints were provided when selecting between alternatives to fix LOCs. To complete a Fix activity, the learner must fix all erroneous LOCs so in the example in Figure 7.4 (right) two LOCs still require fixing.

The fifth type of activity was output prediction (Out) where the student needs to specify the actual output of the code for each given test case (Figure 7.4, right) and contains 1–3 test cases. Out activities were also used in the second evaluation study (Chapter 6) where I had two types of output prediction. There was only one type of Out activity in PyKinetic_Fixed and PyKinetic_Adaptive: identifying the actual output of the given code. There are four choices for all output prediction activities, with only one correct choice. If the learner answers incorrectly, he/she cannot select another choice without first closing the output prediction dialog box (shown in Figure 7.5, middle where submit button is disabled). This was to encourage learners to review the problem and reread the code.

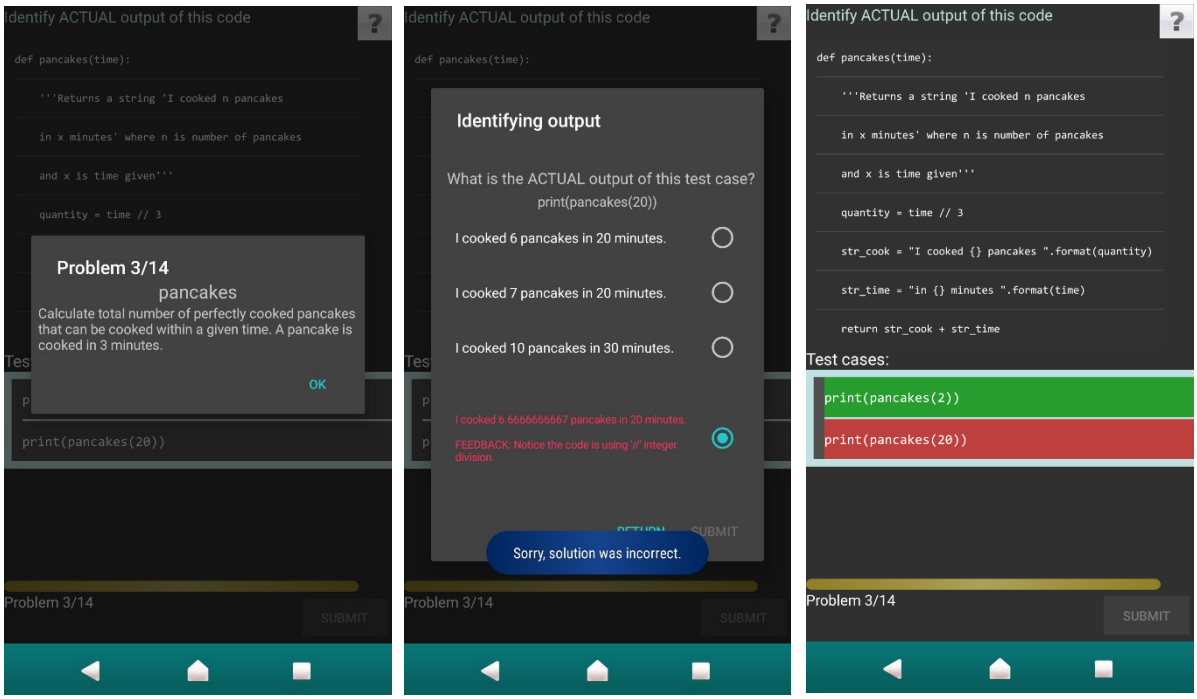


Figure 7.5. Example of Out activity; left: problem details; middle: one of the Out questions where the learner answered incorrectly; right: first test case with the output correctly predicted (highlighted in green) and the second highlighted in red because of the incorrect answer of the learner.

In Figure 7.5, the example shows a problem with function `pancakes` which takes an integer parameter `time`. As mentioned in the problem description (Figure 7.5, left), the function calculates the number of pancakes cooked based on the value of variable `time`; where a pancake is cooked in 3 minutes. The function returns a string which describes the number of pancakes cooked in the given time. In Figure 7.5 (middle), the learner answered incorrectly and chose the last option where a floating-point number was used. In response to this, the incorrect option was coloured in red and a message "Sorry, solution was incorrect." was displayed. Furthermore, a hint was *given "Notice the code is using '//' integer division"* (starts with FEEDBACK: in Figure 7.5, middle). The hint pointed out that integer division was used in the function, therefore the string return value must contain an integer. Therefore, the correct answer was a) I cooked 6

pancakes in 20 minutes. If the learner answered correctly, *"Correct! Great job!"* is displayed and the option will be coloured in green.

All hints in Out activities were procedural hints. Due to the nature of the Out activity, conceptual hints were not provided since I think that these hints might give away too much information and therefore are ineffective. Furthermore, the intention was to encourage learners to reread the code and so only one hint is provided for each Out question. So, selecting the other wrong options b) and c) in Figure 7.5 (middle) would not display any hints. Test cases are highlighted in red if the learner incorrectly identified its output, and green if it is correct (Figure 7.5, right). The Out activity in Figure 7.5 contains code with no errors. However, in some Out activities the given code was erroneous. This is discussed in the next section, where I combine Dbg and Out activities in one problem. If the code is erroneous, the actual output may be none with an error displayed.

7.2 PyKinetic_Fixed and PyKinetic_Adaptive

For this study, I developed a version of PyKinetic (referred to as PyKinetic_Fixed) which combines several activities in order to target multiple coding skills. I implemented PyKinetic_Fixed on a component-skills perspective (McArthur et al, 1988), recognizing that programming requires a set of interrelated skills. As mentioned in other chapters, I regard sequencing of tasks to be an attribute of the pedagogical strategy in PyKinetic. Furthermore, results from my previous studies encouraged us to develop an adaptive version (PyKinetic_Adaptive) which offers the same activities in PyKinetic_Fixed but provides adaptive problem selection. Five types of learning activities are offered (as presented in the previous Section 7.1): regular Parsons problems (Reg_Pars), Parsons problems with incomplete LOCs and menu-based SE prompts (Pars_Inc), identifying erroneous LOCs (Dbg), fixing erroneous LOCs (Fix), and output prediction (Out). With the five types of activities, I defined seven levels of problems covering six Python topics: string manipulation, conditional statements, while loops, for loops, lists, and tuples. These problems were ordered in increasing complexity (Table 7.2) based on factors mentioned in Section 3.13. Problems on levels 1-4 consist of only one activity each, whereas problems on levels 5-7 are combinations of two or more activities. For levels 5-7, problems started by requiring the student to identify incorrect LOCs (Dbg). All problems from levels 1-4 contained non-erroneous code, whereas levels 5-7 contained up to 3 incorrect LOCs since they start with a Dbg activity. Learners were not allowed to skip any activity. Therefore, a problem must be completed to proceed to the next. For levels 5-7 all activities must be solved to complete the problem.

The simplest problem type (**Level 1**) had one activity which was regular Parsons problems (*Reg_Pars*), which only require the LOCs to be reordered to produce the expected outcome. Level 1 targets code understanding and code writing skills. On **Level 2**, Out activity was provided where the problem

description was given, and the student needs to specify the actual output of the given code for each given test case. Level 2 supports code tracing skills. **Level 3** problems consisted of a single activity (*Dbg*), requiring the student to identify *n* erroneous LOCs, where *n* was given. In Dbg activities, the learner was given the problem description and some test cases with the actual output the code produces. I treated Out activities (level 2) as less complex than Dbg activities (level 3), because the results from my previous study revealed that debugging problems were more beneficial for HP students (Chapter 6). **Level 4** contained Parsons problems with incomplete LOCs and SE prompts (*Pars_Inc*). It may be argued that Pars_Inc could be easier than Dbg (level 3); however, in my previous study I found that Pars_Inc were more time consuming and required more effort. Like Reg_Pars in Level 1, Pars_Inc also targets code understanding and code writing skills. However, Pars_Inc further supports code writing and procedural knowledge via the extra task of solving incomplete LOCs. Also, the additional SE prompts are for supporting conceptual knowledge.

Table 7.2 Combinations of Questions in Levels 1-7

Level	Problems	Additional Information Given
1	Regular Parsons problem (Reg_Pars)	Expected output
2	Output prediction (Out)	Test cases
3	Identify erroneous LOCs (Dbg)	Test cases with actual output
4	Parsons Problem with incomplete LOCs and SE prompt (Pars_Inc)	Expected output
5	Dbg -> Out	Test cases
6	Dbg -> Fix	Test cases with expected output
7	Dbg -> Out -> Fix	Test cases

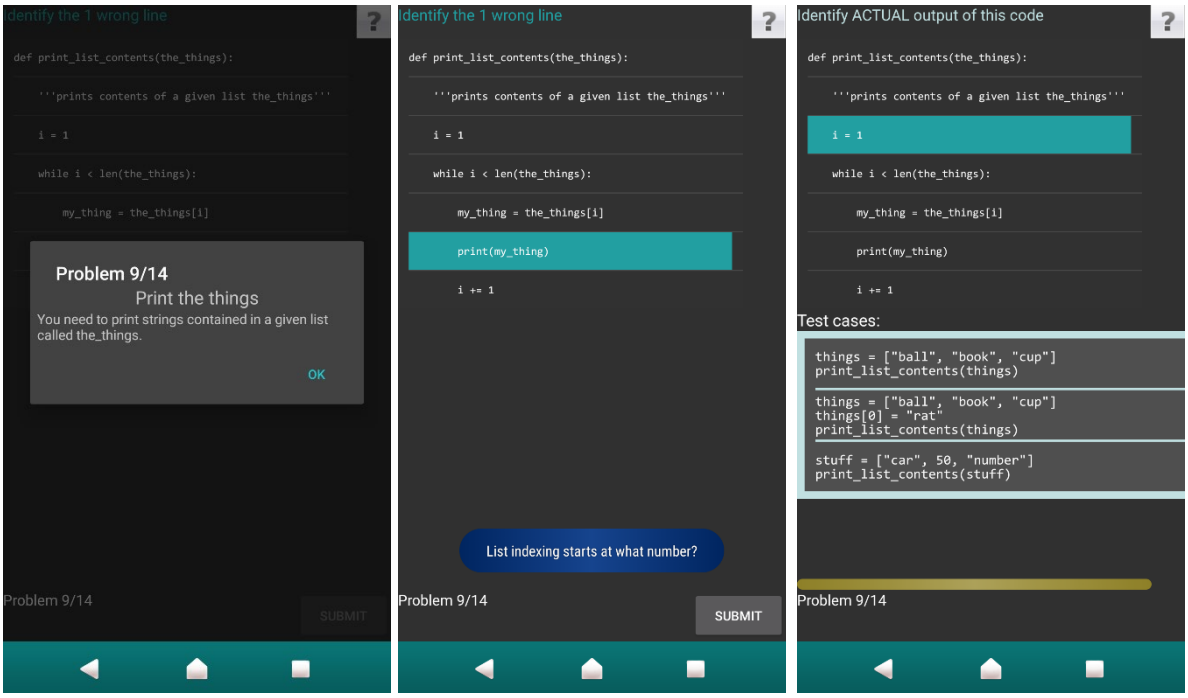


Figure 7.6 Example of Dbg -> Out; Left: problem description; Middle: Dbg activity where the learner submitted an incorrect solution; Right: Out activity with the erroneous LOC already identified (highlighted in turquoise)

Level 5 problems were a combination of debugging and output prediction (*Dbg -> Out*) where only the problem details were given together with the code. **Dbg -> Out** problems started with a Dbg where the learner identified incorrect LOCs (Figure 7.6, middle). In Figure 7.6 (middle), the learner submitted an incorrect answer for the second time, so a hint was given. When the Dbg activity is completed, the learner proceeds with the Out activity to select the output of the code for each test case (Figure 7.6, right). Learners should consider that the code may not have any output due to the errors that they had just identified in the Dbg activity. A Dbg -> Out problem is solved once all the output prediction questions are completed. Level 5 problems target debugging and code tracing skills.

Dbg -> Fix (**Level 6**) is a combination of Dbg and Fix activities. Each problem of this type starts with the learner identifying incorrect LOCs (Dbg), which he/she then needs to fix (Fix). As mentioned in the previous section (7.1), all Fix activities start with a Dbg activity. Dbg -> Fix problems were placed a level above Dbg -> Out since as revealed from my previous evaluation study (Chapter 6), debugging activities were more suitable for students with high prior knowledge. Level 6 targets debugging and code writing skills.

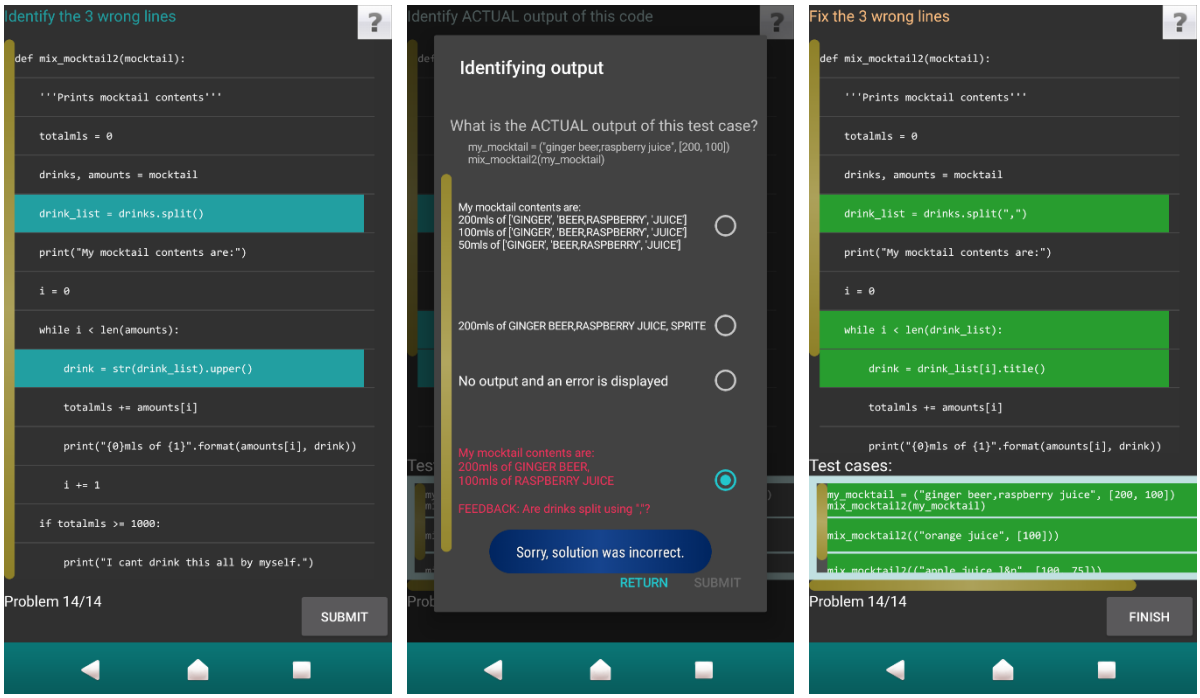


Figure 7.7 Example of Dbg –> Out –> Fix problem; left: Dbg where the learner only identified two out of the three erroneous LOCs; middle: one of the Out questions where the learner answered incorrectly; right: Fix activity completed by the learner

Lastly, the most complex problem type is *Dbg -> Out -> Fix* (**Level 7**), which was a combination of three activities: identifying the erroneous LOCs, predicting the output for the same erroneous code, and finally fixing the errors (Figure 7.7). The first two activities were the same as Dbg -> Out (Level 5), but after identifying the output, the learner was also required to fix the erroneous code. In Level 7, once the learner completes the Out activity, the test cases were highlighted in green. This visual was shown even after moving to the last activity (Fix). As presented in Figure 7.7 (right) when the learner completes all the activities in Level 7, the test cases are highlighted in green, as well as the LOCs that used to be erroneous. The button on the lower right-hand corner in Figure 7.7 (right) shows "Finish" because it was the last problem in the tutor.

Like PyKinetic_DbgOut in Chapter 6, both PyKinetic_Fixed and PyKinetic_Adaptive only had two screens: start screen and problem-solving screen. The *start screen* was used in the same manner for participants to explicitly indicate when they are ready to start using the tutor and to enter their unique participant id given to them for the purposes of the evaluation study. Despite offering multiple activities and problem levels (Table 7.2), the *problem-solving screen* was portrayed to the learners as only one screen to promote a seamless user experience. Technically speaking these are implemented using three separate Android activity classes for Parsons problems, debugging activities, and output prediction. However, my design depicts only one screen to promote a seamless user experience. I recognise that design is vital for

supporting pedagogical activities. Therefore, all versions of PyKinetic including PyKinetic_Fixed and PyKinetic_Adaptive adhere to my user interface design guidelines and authoring guidelines covered in Chapter 3.

7.3 Architecture and Development

The general architecture of the system is described in Chapter 3.4.1. Specific characteristics of PyKinetic_Fixed and PyKinetic_Adaptive are described here. The code statistics are reported in Tables 7.3-7.4. Like in my other evaluation studies (Chapters 4-6), the code statistics presented in Tables 7.3-7.4 only include code that I have written. Other files like third-party libraries are not included. Moreover, files generated upon compilation, building, and executing of the applications are not included.

Table 7.3 Code Statistics for PyKinetic_Fixed

	Java	**XML**
Classes/Files	30	9
Total LOCs without blank lines	8320	558
Source code lines	6831	558
Comment lines	1489	0

Table 7.4 Code Statistics for PyKinetic_Adaptive

	Java	**XML**
Classes/Files	32	9
Total LOCs without blank lines	8547	806
Source code lines	7088	806
Comment lines	1459	0

With regards to the manifest XML file, both applications had four permissions: internet access, access to the state of the network connection, reading and writing data in the device. Like PyKinetic_DbgOut in Chapter 6, all the permissions were necessary for recording logs (in real time) during the evaluation study; where the logs contained user actions. The first two permissions regarding network connectivity was necessary for sending logs to a server. The latter permissions for reading and writing data were to record the logs in the devices used in the evaluation study.

7.3.1 Storage

The application architecture for PyKinetic_Fixed and PyKinetic_Adaptive is the same as in PyKinetic_DbgOut (Chapter 6.3) Logs stored as text files were sent through a server. Furthermore, a simple php admin page was made to monitor the progress of the participants during the evaluation study in real

time, via the logs received in the server. Like in previous versions of PyKinetic, the problem set was stored within the SQLite database in the application. In previous versions of PyKinetic only simple feedback was provided. Therefore, it was not necessary to store any feedback in the database. However, since PyKinetic_Fixed and PyKinetic_Adaptive contained hints for most activities, these hints were stored in the database. Simple feedback was stored in the similar manner as in previous versions, in the strings XML file together with the other texts used throughout the application. Logs were recorded and sent through a server. These logs were simply stored as text files. Furthermore, a simple php admin page was made to monitor the progress of the participants during the evaluation study in real time, via the logs received in the server.

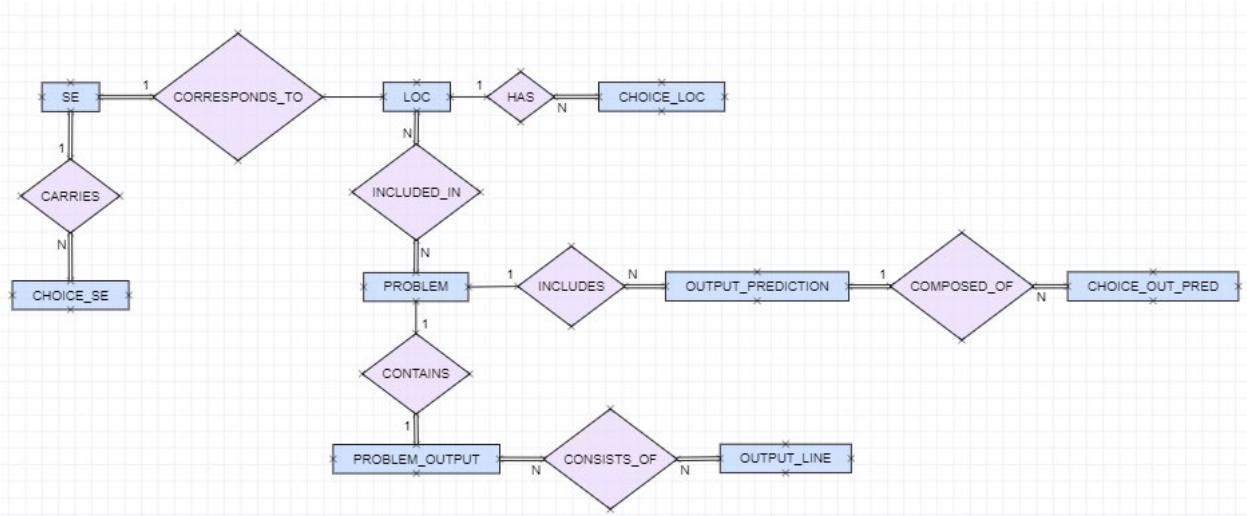


Figure 7.8 ER diagram of the database in PyKinetic_Fixed and PyKinetic_Adaptive without the attributes

The ER diagram of the database used in PyKinetic_Fixed and PyKinetic_Adaptive have identical structures and is displayed in Figure 7.8 (without the attributes). The Parsons problems were stored in a similar manner as previous versions where all possible positions of the LOCs in each problem are stored as integers in the database. For SE prompts a set of choices were stored. Similarly, for Out activities, each question contains a set of choices. Only problems without output prediction activities provide an output description. Like the database structure in other versions, each line in the output may be used in multiple problems. For instance, output lines displaying "0" may also be contained in the output of another problem. All problems regardless of the type of activity contains at least one LOC. Moreover, some LOCs from Pars_Inc and Fix activities contain choices.

7.3.2 Interaction Elements

The list of navigational elements used in the prototype is reported in Table 7.5. Elements require either single tap or long tap actions.

Table 7.5 List of buttons, icons, and navigational elements

Screen/Dialog	Element	Purpose
Any problem	"?" button	Displays the problem details dialog box
Problem solving screen problems 2-14	Back button	Screen stays in place and "Sorry, you can't go back to the previous problem." is displayed below the screen.
Problem solving screen for all problems expect the last	NEXT PROBLEM button	Navigates to the next problem in the sequence. Note that this button is only displayed when the learner has completed the current problem.
Problem solving screen for the last problem (problem number 14)	FINISH button	Displays a dialog box "Awesome work! You have finished all problems!". Then, when "Ok" button is clicked navigates back to the Start screen and shows "CONGRATULATIONS!!! YOU'RE DONE!!".
Problem details dialog box (Same for all problems)	Ok button	Closes the dialog box
Start screen	Edit text for participant id	Element where users enter in their participant id for evaluation purposes
	Start button	Navigates to the first problem offered
	Device's back button	"Closes" the app (not completely, it will still run in the background)
Parsons problem solving screen (Reg_Pars and Pars_Inc)	"?" button	Displays the problem details dialog box
	Back button	Screen stays in place and "Sorry, you can't go back to the previous problem." is displayed below the screen.
	Drag icon	By holding on the drag icon and moving it vertically, it moves a LOC from the position it was dragged from to a new position where it was dropped (upon release of the drag icon).
	Incomplete LOC (only for Pars_Inc)	Long tapping will select the LOC to indicate that the learner wants to solve this LOC. Tapping on a selected incomplete LOC would change the LOC and show the alternative keyword/s to fill the blank. Other notes: An incomplete LOC needs to be selected first before being able to see alternatives.

		Any changes to an incomplete LOC will undo when LOC is moved or unselected unless the answer is finalised by long tapping on the LOC (like when selecting).
	SUBMIT button	Initially disabled, until all incomplete LOCs are correctly solved. Gives feedback about the Parsons problem.
SE prompt dialog box	Back button	Disabled
	Output choice	Tapping on a choice indicates that the learner has selected this as his/her solution and at the same time SUBMIT button is triggered. Other choices selected before are unselected at the same time.
	OK button	Initially disabled, enables when at least one choice is selected; used by the learner to submit their solution to the SE prompt.
	RETURN button	Only enables after the learner submits their solution, dismisses the dialog box.
Out problem-solving screen	Test cases	A long tap on a test case shows the output prediction dialog for the chosen test case.
Out dialog box	Device's back button	Disabled
	Output choice	Tapping on a choice indicates that the learner has selected this as his/her solution and at the same time SUBMIT button is triggered. Other choices selected before are unselected at the same time.
	SUBMIT button	Initially disabled, used by the learner to submit their solution to the output prediction activity. When a learner submits an incorrect solution, this button is disabled again. Upon dismissal of the output prediction dialog, the test case incorrectly answered is highlighted in red. If the learner's answer is correct, the choice is highlighted in green while the rest of the choices in red. When the output prediction dialog is dismissed, the test case successfully answered is highlighted in red.
	RETURN button	Dismisses the dialog box
Dbg problem-solving screen	Non-test case LOC	A long tap highlights the LOC in turquoise, indicating that it is selected.
	Test cases	A long tap on a test case shows its actual output when it is executed.

	SUBMIT button	Submits the LOC/s selected as the solution to receive feedback. (Only enabled when there is at least one LOC selected)
Fix screen	LOC which requires fixing	Long tapping on LOC that needs to be fixed will be highlighted in turquoise. A LOC in turquoise is activated, indicating that the learner is working on this LOC. Tapping on an activated LOC reveals the alternative choices (the entire LOC changes on tap). Long tapping on an activated LOC indicates that the learner is submitting this LOC as their answer (to replace the old LOC to be fixed). Tapping on a regular LOC does not do anything. Similarly, tapping on an unselected LOC to be fixed does not do anything.

7.4 Adaptive Problem Selection

The first problem was selected based on the participant's pre-test score. The score of the participant was manually entered on the start screen of PyKinetic_Adaptive (Figure 7.9) before the smartphone was handed over to the student. If the participant scored 49% and below, a Reg_Pars was given. Participants who scored 50% or more were given an Out (level 2) problem; however, if the participant scored lower on Dbg questions compared to Out questions, a Dbg problem was given (level 3). A Dbg problem was also given to participants who performed equally on output prediction and debugging questions and scored more than 75%.

After the first problem, the adaptive strategy used the performance on the previous problem 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 was selected for scores between 50% and 74%, and a difficulty level 3 if the score was at least 75%. PyKinetic_Adaptive calculated the score for each activity in a problem separately, and the problem score was the average of the activities scores. The score for an activity depends on the time taken (*TimeScore*) and the number of attempts (*AttemptScore*). I used data from previous studies to estimate the ideal time needed to solve each activity. The ideal number of attempts for an activity was the minimum number of submissions needed to complete it (i.e. clicking the Submit button and finalising an incomplete line). The ideal time and ideal attempts were stored in the database. *TimeScore* was the quotient of ideal time and the actual time the student took. *AttemptScore* was calculated as the quotient of the ideal and the

actual number of submissions the student made. Both scores were then combined to compute the score for the activity (Equation 1).

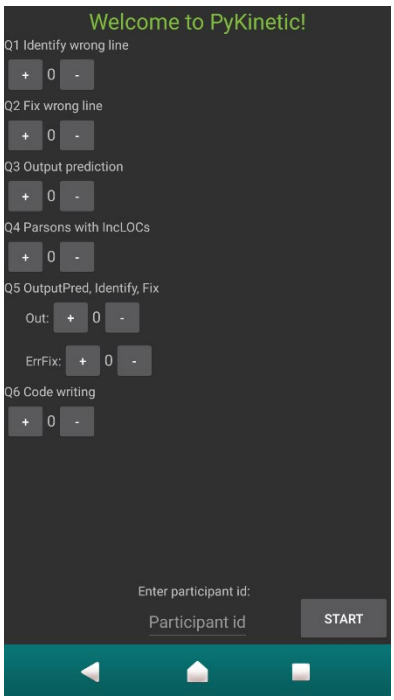


Figure 7.9 Start screen of PyKinetic_Adaptive

$$ActivityScore = (0.5 * TimeScore + 0.5 * AttemptScore) - Penalty \quad (1)$$

The penalty was applied if the time per attempt is less than 10 seconds (0.17 min). In previous studies, students who used the trial-and-error strategy spent less than 10 seconds per attempt. The penalty was calculated based on the time taken per attempt (*AttemptTime*) and on 10 seconds threshold (Equation 2). The shorter the *AttemptTime*, the bigger the penalty.

$$Penalty = 0.17 - AttemptTime \quad (2)$$

7.5 Experimental Design

I conducted a controlled lab study with PyKinetic_Fixed and PyKinetic_Adaptive. My hypotheses for this study were the following:

(H1) the problems in PyKinetic_Fixed and PyKinetic_Adaptive will be effective for learning

(H2) the adaptive selection would be superior to a fixed sequence of problems

(H3) the problems in PyKinetic_Fixed and PyKinetic_Adaptive will be more effective for low prior knowledge (LP) students

(H4), high prior knowledge (HP) students who used adaptive problem selection will benefit more than HP students in the control group.

My first hypothesis (H1), was that both versions would be effective for learning because I have designed both versions based on results from my previous evaluations which were also successful in supporting learning. Secondly, I hypothesised that the adaptive problem selection would be better than a fixed sequence of problems due to results from my previous studies. More specifically, in my first evaluation study (Chapter 5), Parsons problems with incomplete LOCs and SE prompts revealed to be more beneficial for LP students while debugging exercises were more beneficial for HP students (Chapter 6). Therefore, the adaptive problem selection should help in providing the activity more suitable for the student based on his/her abilities, compared to a fixed sequence. For (H3), I hypothesised that both versions will be more effective for LP students since I designed both versions to be more suitable for novices (specifically those who need more practice such as LP). Lastly (H4), I hypothesised that HP students who used PyKinetic_Adaptive will benefit more than HP students who used PyKinetic_Fixed. This is because the adaptive problem selection should provide more debugging activities for HP students, which I found more beneficial for their learning in (Chapter 6).

7.5.1 Participants

I recruited 32 participants from an introductory programming course at the University of Canterbury, of whom 30 participants completed all phases of the study. The participants learnt about all Python topics covered in PyKinetic before the study. The study was approved by the Human Ethics committee of the University of Canterbury.

7.5.2 Method and Materials

The participants were randomly assigned to the control or experimental group. The control group had a fixed set of problems, with two problems on each level (Table 7.6). The experimental group used a version of PyKinetic which provided adaptive problem selection (discussed in Section 7.4). Both groups had 14

problems to solve (increasing difficulty, each covering up to 5 topics). Learners needed to complete one problem before continuing to the next one. The session length was two hours. The participants were first given a brief introduction of the study, then completed a pre-test on lab computers, which had a time limit of 18 minutes. The participants were not allowed to get help and open any integrated development environment to test their answers. Afterwards, an instruction sheet was given for PyKinetic, together with an Android phone with the app installed. For the experimental group, I have entered the pre-test scores into PyKinetic, for the adaptive strategy to select the first problem based on their pre-test scores. The participants interacted with PyKinetic for an hour; a post-test was given afterwards, with the same restrictions as the pre-test.

Table 7.6 Problems in PyKinetic_Fixed vs. PyKinetic_Adaptive

<table>
<tr><th>Problem</th><th>PyKinetic_Fixed (Control)</th><th>PyKinetic_Adaptive (Experimental)</th></tr>
<tr><td>1</td><td rowspan="2">Reg_Pars</td><td rowspan="5">Difficulty level 1: Reg_Pars
Difficulty level 2: Out
Difficulty level 3: Dbg</td></tr>
<tr><td>2</td></tr>
<tr><td>3</td><td rowspan="2">Out</td></tr>
<tr><td>4</td></tr>
<tr><td>5</td><td rowspan="2">Dbg</td></tr>
<tr><td>6</td><td></td></tr>
<tr><td>7</td><td rowspan="2">Pars_Inc</td><td></td></tr>
<tr><td>8</td><td rowspan="3">Difficulty level 1: Pars_Inc
Difficulty level 2: Dbg -> Out
Difficulty level 3: Dbg -> Fix</td></tr>
<tr><td>9</td><td rowspan="2">Dbg -> Out</td></tr>
<tr><td>10</td></tr>
<tr><td>11</td><td rowspan="2">Dbg -> Fix</td><td rowspan="4">Difficulty level 1: Pars_Inc
Difficulty level 2: Dbg -> Out
Difficulty level 3: Dbg -> Out -> Fix</td></tr>
<tr><td>12</td></tr>
<tr><td>13</td><td rowspan="2">Dbg -> Out -> Fix</td></tr>
<tr><td>14</td></tr>
</table>

I had two versions of the test (Appendix A) of similar complexity that were alternatively used as the pre-test for half of the participants. Both tests had six questions (Table 7.7), of the same types as in PyKinetic. However, instead of having two Parsons problems (Reg_Pars and Pars_Inc), there was only one Parsons problem with three extra LOCs (distractors). I refer to this type of Parsons problems as *Pars_Dis*. The problem description for Pars_Dis clearly stated that not all lines were necessary. The three distractors were variants of a single LOC. Pars_Dis problems were used instead of Pars_Inc due to the restrictions in the online quiz system that was used for the pre-/post-tests (the system only allowed fill-in-the-blanks or drag-and-drop activities, but not both at the same time). The Pars_Dis had a drag and drop interface, similar to work by (Ihantola and Karavirta, 2011), where LOCs were dragged from the problem area onto the solution area.

Table 7.7 Questions on Pre/Post-tests

Question	Activities	Question Type	Maximum Mark
1	Out	Multiple choice	1
2	Dbg	Multiple choice	1
3	Fix	Multiple choice	1
4	Pars_Dis	Drag and drop	2
5	Out; Dbg -> Fix	Drop-down choices	2
6	Code writing	Key in solution	5

The code writing question required the learners to type their code without being able to run it. Other questions were given as multiple choice and drop-down list choices (Table 7.7). Questions 1–3 were worth 1 mark each, questions 4 and 5 were worth 2 marks each, and the code writing exercise worth 5 marks. Pars_Dis was worth 2 marks as it requires more effort. Question 5 was also worth 2 marks since there were two problems in this question (Out and Dbg -> Fix). The Pars_Dis question had 8 LOCs including the function definition statement, docstring and the test case. For the code writing question, the student received a 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.

After the pre-test, the control group received problems in the fixed order, as shown in Table 7.6. The problems given to the experimental group were selected adaptively based on each student's performance. Table 7.6 shows the problem types available at different stages of the study. In problems 1-7, the participants could receive a regular Parsons problem (Reg_Pars), output prediction problem (Out) or be asked to identify erroneous LOCs (Dbg). Problems 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 each problem, the adaptive strategy (as discussed in Section 7.4) selected the problem type based on the student's score on the previous problem.

7.5.3 Data Collection

Logs were recorded for all activities. Every user action (button press, single tap, and long tap) triggers additional data to the log: a timestamp with details about the interaction. Feedback received by the student based on their last solution was also logged, which was useful for identifying how close a student was to completing a problem where they failed to do so. Lastly, the total time taken per problem was also recorded in the logs.

Similar to my second evaluation study (Chapter 6), evaluation logs were sent through the server and (in parallel) recorded directly on the smartphone device. The logs were gathered while the user was using the tutor. A log file was sent and recorded to the device when: a learner submits a solution, navigates away from an application screen, and every three minutes. Again, like in Chapter 6, I saved the logs every three minutes to have a reasonable amount of data upon recording the log, and at the same time circumvent data loss.

7.6 Findings

I first present my general findings to address H1 and H2. Then in Section 7.6.2 I present my findings to address H3 and H4, where I divided the participants post-hoc and performed a median split based on their pre-test scores like in my previous studies and like (Adams et al., 2014; McLaren, Adams, and Mayer, 2015). Participants scored less than the median were labelled as low prior knowledge students (LP), while the rest were considered as high prior knowledge students (HP).

7.6.1 General Findings

We eliminated one outlier from the control group, who scored perfect marks on both pre and post-tests 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 used the Wilcoxon Signed Ranks test to investigate our first hypothesis H1. Taken together, the participants improved their scores significantly between the pre- and post-test (the *Improvement* row in Table 7.8, p = .031), and also their scores on code writing questions (the *Improvement code writing* row), which was evidence to confirm that both versions of PyKinetic supports learning (H1). There was no significant difference on the scores and learning gains for other types of questions.

Table 7.8 Pre-test and Post-test results (* denotes significance at the .05 level)

Scores (%)	**All (29)**	**Experimental (15)**	**Control (14)**
Pre-test	72.63 (19.1)	67.22 (19.7)	78.42 (17.3)
Post-test	81.32 (14.7)	76.67 (16.1)	86.31 (11.5)
Improvement	**W = 317** **p = .031***	ns	ns
Pre-test Code Writing	65.86 (35.4)	60.00 (38.5)	72.14 (31.9)
Post-test Code Writing	82.41 (26.00)	74.00 (31.1)	91.43 (15.6)
Improvement Code Writing	**W = 177.5** **p = .03***	ns	**W = 43** **p = .014 ***
Normalised Gain	35.65 (41.46)	30.92 (41.78)	40.71 (42.06)

Table 7.9 Some performance measures

	Experimental (15)	Control (14)
Problems completed	12.67 (1.3)	12.52 (1.6)
Time per problem	3.98 (1)	4.14 (0.9)
Time first half	2.46 (0.9)	3.14 (0.7)
Time second half	5.56 (1.8)	5.18 (1.4)
Problem difficulty level	2.27 (.3)	2.02 (.1)

There were no significant differences between the two groups on the number of completed problems and the average time per problem (Table 7.9). The experimental group participants solved the initial seven problems significantly faster (U=162, p = .012). Furthermore, the experimental group solved significantly more difficult problems (higher average difficulty level) than the control group (U=41.5, p = .004). Figure 7.10 displays the cumulative product of the average problem difficulty level and the problem-solving score until problem 12 (cut-off point set to 73% of participants). Similar to our analysis in our previous evaluation studies in Chapters 5-6, we also used two formulas to calculate normalised gain for this study (Marx and Cummings, 2007).

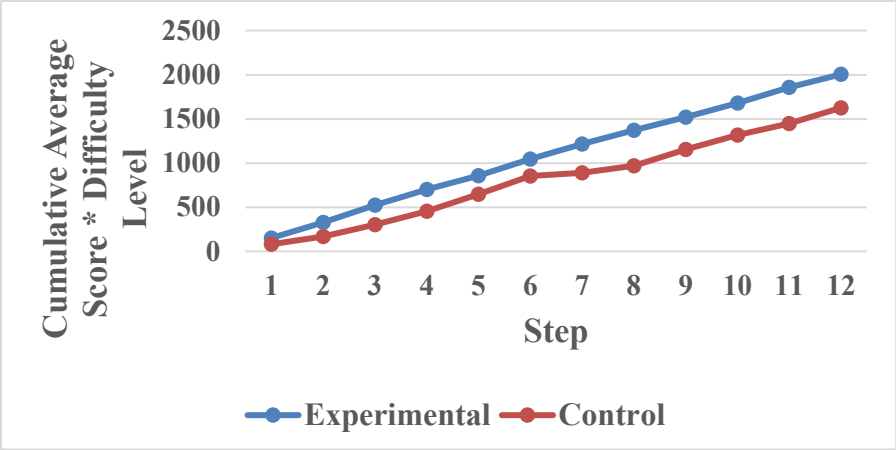


Figure 7.10 Cumulative Average Score * Difficulty Level

The control group received two problems of each type, while the experimental group received problems adaptively, based on their performance. The experimental group solved significantly less (avg = 1.13, sd = 1.2) Reg_Pars, the easiest problem type (U = 154, p = 0.033); at the same time, they solved significantly more Dbg problems (avg = 3.2, sd = 1.6), which were more challenging (U = 42, p = 0.005). Note that these problems were in the first half of the session; therefore, regardless of receiving more difficult problems, the experimental group participants were still significantly faster in completing them. These results support our hypothesis H2.

7.6.2 Analyses of LP compared to HP

I divided the participants based on the pre-test scores: the participants who scored less than the median (72.92%) were labelled as low prior knowledge students (LP), while the rest were considered as high prior knowledge students (HP). Due to random allocation of participants, I discovered that numbers of LP vs. HP students were unbalanced in both groups. There were 15 LP (ten from experimental and five from control), and 14 HP students (five from experimental and nine from control). Table 7.10 reports results of pre/post-test scores of LP and HP students.

Table 7.10 Pre-test and Post-test results LP vs. HP (* denotes significance at the .05 level)

Scores (%)	**LP (15)** **10 Exp. and 5 Cont.**	**HP (14)** **5 Exp. and 9 Cont.**	**Mann-Whitney U test**
Pre-test	57.64 (12.9)	88.69 (8.2)	U = 120, p = .000*
Post-test	74.17 (15.9)	88.99 (8.4)	U = 48, p = .012*
Improvement	W = 102, p = .017* Cohen's d = 1.14	ns	
Normalised Gain	38.41 (37.5)	32.69 (46.6)	ns
Pre-test Dbg, Fix and Dbg -> Fix	60 (28.7)	83.33 (25.3)	U = 57, p =.037*
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 = 58.5, p= .041*
Post-test Out	73.33 (32)	71.43 (32.3)	ns
Pre-test Pars_Inc	89.17 (20.5)	98.21 (4.5)	ns
Post-test Pars_Inc	95 (6.3)	96.43 (5.9)	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 = 57.5, p = .037*
Improvement *Code Writing*	W = 75, p = .039* Cohen's d = .87	ns	

Using the Wilcoxon Signed Ranks test, I found that the LP 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 HP students. I also compared the scores of LP and HP students. I 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 = 57, p = .037), but not on the post-test, therefore showing some indication that LP might have reached the level of the HP students on the post-test questions. Similarly, the output prediction

scores on the pre-test were significantly different (U = 58.5, p = .041) but there was no difference on the post-test. However, the scores on output prediction for HP students might be due to a ceiling effect. Nevertheless, these findings were enough evidence to confirm H3, that this version of PyKinetic was more beneficial for LP.

It is interesting that even though there was a significant improvement on code writing based on the pre- to post-test scores of the LP, there was still a significant difference with their post-test scores when compared with HP students (U = 57.5, p = .037). There were no significant differences between the normalised gains of the LP and HP students. This was slightly surprising since there was only a significant improvement in the learning gains of the LP students. The imbalance of the students between control and experimental group likely affected our results.

I also compared performances of LP and HP students within PyKinetic. There were no significant differences on the average time, the number of attempts per problem and the average problem level between LP and HP students. However, average scores of HP students are significantly higher than scores of LP (U = 41, p = .004).

Table 7.11 Significant Results in Problems Solved by LP and HP Students

Problem Solved	LP Exp. (10)	LP Cont. (5)	HP Exp. (5)	HP Cont. (9)
Reg_Pars	1.5 (1.2)	2 (0)	.4 (.9)	2 (0)
Dbg	3.2 (1.1)	2 (0)	3.2 (2.4)	2 (0)
Time First Half	2.53 (.9)	3.71 (.8)	2.32 (.8)	2.89 (.5)
Score First Half	71.43 (8.8)	65.53 (9.7)	80.48 (7.2)	75.67 (8.3)
Average Problem Type First Half	2.26 (.3)	2.01 (.3)	2.56 (.3)	2.14 (.2)

I used the Wilcoxon Signed Ranks test to identify whether there were significant improvements for the subgroups. Only LP in the control group improved significantly between pre- and post-test (W = 15, p = .043, Cohen's d = 1.38). There was no significant difference on the pre/post-test results and normalised gains between the LP in the control group and the LP in the experimental group. Similarly, no significant difference was found when pre/post-test results and normalised gains were compared between HP students in the control group and those in the experimental group.

Table 7.11 reports the numbers of problems solved by LP and HP students. Analysing the problems that the four subgroups received, the Friedman test revealed a significant difference on the number of regular Parsons problems solved (p = .046, FM = 8). Moreover, an overall significant difference was found for the initial seven problems on the average time spent per problem (p = .026, FM = 9.24), the average score (p = .033, FM = 8.76) and average problem level between the four subgroups (p = .037, FM = 8.489).

LP in the experimental group attempted significantly more Level 3 (Dbg) problems than LP in the

control group (U = 7.5, p = .028); they were also significantly faster on the initial seven problems (U = 46, p = .008). Furthermore, HP students from the experimental group received significantly harder problems in the first half of the session than HP learners from the control group (U = 5, p = .019). HP students in the experimental group received significantly fewer Level 1 (Reg_Pars) problems (easiest activity in PyKinetic) than HP in the control group (U = 40.5, p = .012). There was not enough evidence to support our hypothesis H2 for HP students.

7.7 Discussion

I presented a study with PyKinetic_Fixed and PyKinetic_Adaptive, which is aimed at supporting various programming skills for introductory programming students. My results show that students improved their performance after solving problems in PyKinetic for an hour. The adaptive version of PyKinetic provided a further benefit in comparison to the fixed strategy. Although the experimental group received harder problems than the control group, they achieved comparative scores on the post test, and were significantly faster on the first half of the problems.

One of the major contributions of my work is providing several activities in order to target multiple coding skills: code understanding, code tracing, code writing, and debugging. I acknowledge that programming requires a set of interrelated skills. Therefore, I implemented PyKinetic_Fixed on a component-skills perspective (McArthur et al, 1988), and regard the sequencing of tasks to be a distinguishing characteristic of the pedagogical design of PyKinetic. Furthermore, results from my previous studies encouraged us to develop an adaptive version (PyKinetic_Adaptive) which offers the same activities in PyKinetic_Fixed but provides adaptive problem selection. I sequenced the activities in order of complexity based on literature and results from my previous studies. I designed seven levels of difficulty which comprised with levels 1-4 with one activity, and levels 5-7 with 2-3 activities. Levels 1-4 contained one activity to gradually aim at improving one or two coding skills at a time. My pedagogical design intended that the first three levels comprised of activities that are different in nature to focus on developing different coding skills. Although it is the simplest problem, Level 1 (Reg_Pars) aids in enhancing code understanding and code writing skills. Level 2 output prediction (Out), assists in enhancing code tracing skills and Level 3 identifying erroneous LOCs (Dbg), aims at debugging skills. Level 4 Parsons problems with incomplete LOCs (Pars_Inc), also helps with code understanding and code writing skills like Reg_Pars. However, Pars_Inc further supports code writing by providing incomplete LOCs which act as micro-exercises. Pars_Inc reinforces procedural knowledge and to also assist in the heightening conceptual knowledge, I introduced menu-based SE prompts. Based on results from my previous study (Chapter 5), the combination of Parsons problems with incomplete LOCs and menu-based SE prompts were effective

for learning especially for learners with low prior knowledge. The last three levels (5-7) had a combination of 2-3 activities which aimed at enriching multiple coding skills at a time. Level 5 aimed in boosting debugging and code tracing skills by the combination of Dbg and Out activities. Then, Level 6 (Dbg and Fix) supported debugging and code writing skills. Finally, Level 7 (Dbg, Out, and Fix) reinforced debugging, code tracing, and code writing skills.

Apart from supporting multiple coding skills, I also provided support in enhancing both procedural and conceptual knowledge. This was achieved through hints, and with the additional SE prompts. For Reg_Pars, Dbg, and Pars_Inc, half of the hints provided was procedural while the other half was conceptual. For Out activities, only procedural hints were given. I have decided to not give any conceptual hints for output prediction activities because it is difficult to give them to aid the learner without divulging too much information. For example, for an Out activity with conditional statements, an example of a conceptual hint could be *"The code inside an if statement is executed when the condition is met."*. This hint may not be useful for an average learner because it might already be obvious. Another example of a possible conceptual hint is to define a support library function used in the code. For example, in a problem where the expected result is a string with all its characters in upper case, a conceptual hint could be "*upper() returns a copy of a string with all its characters in upper case."*. However, this reveals too much information to the learner. Nevertheless, I am just speculating possible scenarios here. It may have been beneficial for learners solving Out activities to receive half of the hints for conceptual and the other half for procedural knowledge.

For Fix activities and when completing incomplete LOCs, no hints were provided. I intended not to provide any hints when solving these due to feedback received from previous studies (Chapter 5-6). Solving these tasks were made easier by tapping through alternative choices instead of typing. However, some participants mentioned that the ease of control encouraged them to use trial and error strategies. Therefore, I have decided not to provide any hints when solving these tasks. It may be argued that providing hints might lessen trial and error behaviour. But I suggest that a better solution is probably to also provide limitations in solutions for these tasks. Perhaps provide a time out/pause each time an alternative choice is submitted. Another possible improvement is to implement game elements with a points system to incur penalties and gain rewards, based on the performance of the student to discourage using trial and error strategies.

PyKinetic_Fixed and PyKinetic_Adaptive were both implemented with a component-skills strategy where problems were ordered in an increasing complexity. The only difference is the adaptive problem selection offered in PyKinetic_Adaptive. Due to the fixed set of problems, PyKinetic_Fixed offered learners a full experience of all the problems in the tutor. However, for PyKinetic_Adaptive, students could receive the same type of problem multiple times. For example, a consistently high performing student should never receive a Parsons problem (either Reg_Pars or Pars_Inc). On the contrary, consistently low-performing

students would always receive a Parsons problem. I intended it to be this way since as I found from my previous study (Chapter 5), high prior knowledge students (assuming they are also high-performing students) had insignificant benefits when solving Parsons problems. Middle performing students would receive a mixture of Parsons problems, Out and Dbg problems. Despite my intentions, my results revealed that although the experimental group had lower pre-test scores (avg=67.22, sd=19.7) than the control group (avg=79.86, sd=17.5), more difficult problems were given to the experimental group. In fact, the experimental group received less Reg_Pars ($p = .029$, $U = 165$) which is the simplest activity in PyKinetic and more Dbg activities ($p = .004$, $U = 45$). However, despite receiving more problems with higher difficulty level ($U=165.5$, $p = .004$), the experimental group were still significantly faster in completing them compared to the control group ($U=173$, $p = .01$). Therefore, I seek to improve my adaptive problem selection strategy because it may be argued that although the experimental group performed well, they should have received significantly easier problems to support their learning better. Perhaps a longer session is needed to fully see the effect of the learning, especially in PyKinetic_Adaptive where some activities i.e. Parsons problems is not given to the learner due to the performance of the learner. In addition, the imbalance of the participants' abilities due to random allocation to each group gave limitations for analysis. I propose that I repeat my evaluation study and dynamically assign participants to either control or experimental group to ensure that both groups have comparable prior knowledge.

7.8 Conclusions

The main contribution of this evaluation study is that a programming tutor in a smartphone is effective even for students with low prior knowledge to learn multiple coding skills by solving activities of various nature.

The results of my study show that students improved their performance after solving problems in PyKinetic for an average of one hour. The results support my hypothesis H1, revealing evidence that the activities chosen were effective for learning, as well as the combination and sequence of the activities.

When I divided the participants in each group based on the median score on the pre-test, I found an imbalance on both groups: 67% of the control group were HP students, and on the contrary 67% of the experimental group were LP students. The adaptive version of PyKinetic proved effective regardless of the imbalance of the participants' abilities on both groups due to random allocation. Although the experimental group were mostly LP students, their average time, extra attempts and score on PyKinetic were statistically comparable to that of the control group, which on the contrary had more HP students. Moreover, the experimental group participants were notably faster on the first half of the session despite receiving significantly more output prediction and debugging problems (i.e. harder problems). Overall, the experimental group received significantly more difficult problems. These results provide enough evidence

to support my hypothesis H2, that the adaptive strategy is beneficial.

My results show enough evidence to accept hypothesis H3, that PyKinetic is more beneficial for low prior knowledge students. Despite having only interacted with PyKinetic for an hour, LP 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 LPs for code writing were still significantly lower than scores of HP students. This is consistent with results from literature, showing that code writing skills require higher order of knowledge than other coding skills. I also found some evidence that the LP learners were learning multiple coding skills. The scores of LPs on debugging (identifying errors and code fixing) and output prediction questions on the pre-test were significantly lower than the scores of HP students. However, their post-test scores for those question types were not significantly different; indicating that the LP have reduced their gap on debugging and code tracing skills.

Hypothesis H4 was that HP students in the experimental group would benefit more than those in the control group. Results revealed that HP students in the experimental group received more difficult problems than those in the control group. However, that was not enough evidence to support hypothesis H4. There were only five HP students in the experimental group but ten in the control group which most possibly affected my results.

The limitations of this study include the small set of participants, and imbalance of the abilities in each group. There was also limited feedback provided by PyKinetic. For each activity, there is only one pre-defined hint. This resulted in situations when feedback was not helpful to students, especially those who were using the trial-and-error strategy. However, it did not prove to be a major impediment for learning.

PyKinetic is designed to support practicing coding skills in Python, and my findings support this. I aim to further improve PyKinetic and my adaptive problem selection strategy. Another direction for future improvement of PyKinetic is to add more hints. A good balance of scaffolding and difficulty must be maintained to avoid students from floundering. This study has the potential of revealing which activities are most effective for specific coding skills. However, the small set of participants limits my analysis. My future work also includes repeating the study with more participants.

The latest versions of PyKinetic: PyKinetic_Fixed and PyKinetic_Adaptive were covered in this chapter. These versions of PyKinetic were developed based literature and on findings from my evaluation studies (Chapters 4-6). The next chapter is the last chapter, which summarises and concludes the entire PhD project.

8 CONCLUSIONS

This chapter starts by presenting the summary of the thesis and the research questions for my PhD. Afterwards, I present the contributions of my research project. Then, I present the limitations of the research, and lastly conclude the thesis with future directions.

8.1 Summary

The aim of my PhD research project is to develop effective programming activities for a smartphone tutor. I introduced PyKinetic, a smartphone tutor for Python programming aimed at introductory programming students. The intention of my project is not to develop a finished product; rather PyKinetic was developed to conduct evaluations to investigate effective activities for enhancing coding skills in a smartphone tutor. Moreover, PyKinetic is designed to be a supplementary resource for students and is not meant to replace other avenues of learning.

I presented my motivations for the research project in Chapter 1. In Chapter 2, I presented literature from computer science education, educational psychology, and mobile learning. Mobile learning has been proved effective for various domains including teaching computing. Moreover, I showed other examples of educational systems for Python programming. However, only a few of these are developed specifically for targeting coding skills on mobile devices (Karavirta, Helminen, and Ihantola, 2012; Vinay, Vaseekharan and Mohamedally, 2013; Grandl et al., 2018; Mbogo, Blake, and Suleman, 2016). There were other promising mobile tutors for Python programming developed by commercial companies (SoloLearn, 2019; DataCamp, 2019; Enki.com, 2019; Py, 2019; Mimohello GmbH, 2019; Learn Programming, Programming App, Coding App Education, 2019; ApkZube, 2019; Zenva Pty Ltd, 2019; Coding and Programming App, 2019). But because the applications are of commercial nature, there are no published evaluations that assesses the effectiveness of these tutors. The only basis for comparison would be the ratings and comments publicly displayed on Google Play for each application. Therefore, my research project seeks to address the gap in knowledge by investigating the effectiveness of supporting programming skills by learning via a mobile tutor.

To ensure that my learning activities are effective for learning, I considered the strengths and limitations of smartphones. Combined with the literature I presented in Chapter 2, I developed guidelines that I followed throughout various aspects of the research project. In Chapter 3, I presented guidelines in designing programming activities and creating content for PyKinetic, together with my plans for evaluation studies. The activities and content for all versions of PyKinetic were implemented based on the guidelines presented in Chapter 3. I presented my pilot study in Chapter 4, and my other evaluation studies in Chapters 5-7. The summary of the evaluation studies is presented in Table 8.1.

Table 8.1 Summary of Evaluation Studies

Study Summary	**Hypotheses or Research Goals**	**Is the hypothesis validated or is the research goal achieved?**
Pilot Study (Chapter 4) PyKinetic_Pilot Treatment: none Problems: 1) Regular Parsons problems (Reg_Pars) 2) Parsons problems with distractors	(Pilot_R1) to evaluate the UI of PyKinetic_Pilot, most importantly to investigate whether the orientation of the interface (portrait or landscape) when solving Parsons problems is important for learning;	Yes
	(Pilot_R2) to identify problem-solving strategies used by students and tutors when working on Parsons problems with and without distractors.	Yes
Study 1 (Chapter 5) PyKinetic_IncLOCs and PyKinetic_IncLOCs_SE Treatment: menu-based SE prompts Problems: 1) Parsons problems with incomplete LOCs and menu-based SE prompts (Pars_Inc)	(H1) Both PyKinetic_IncLOCs and PyKinetic_IncLOCs_SE would be successful in supporting learning	Yes
	(H2) Menu-based SE prompts would result in further learning benefits	Yes
	(H3) Students with low prior knowledge (LP) would learn more from my Parsons problems in comparison to those with high prior knowledge (HP).	Yes

Study 2 (Chapter 6) PyKinetic_DbgOut Treatment: none Problems: 1) Code Reading (Dbg_Read) 2) Identifying erroneous LOCs (Dbg_Ident) 3) Output prediction of actual output (Out_Act) 4) Dbg_Ident -> Out_Act 5) Dbg_Ident -> Output prediction of expected output (Out_Exp) 6) Dbg_Ident -> Fixing erroneous LOCs (Dbg_Fix) 7) Dbg_Ident -> Out_Act -> Dbg_Fix	(H1) The combination of coding activities is effective for learning programming. (H2) LP students would have higher learning gains in comparison to HP students.	Yes, but primarily for HP. No
Study 3 (Chapter 7) PyKinetic_Fixed and PyKinetic_Adaptive Treatment: adaptive problem selection Problems: 1) Reg_Pars 2) Out_Act 3) Dbg_Ident 4) Pars_Inc 5) Dbg_Ident -> Out_Act 6) Dbg_Ident -> Dbg_Fix 7) Dbg_Ident -> Out_Act -> Dbg_Fix	(H1) The problems in PyKinetic_Fixed and PyKinetic_Adaptive will be effective for learning (H2) The adaptive selection would be superior to a fixed sequence of problems (H3) The problems in PyKinetic_Fixed and PyKinetic_Adaptive will be more effective for LP students in comparison to HP students. (H4) HP students who used adaptive problem selection will benefit more than HP students in the control group.	Yes Yes Yes Inconclusive

In Chapter 1.2 I enumerated my research questions. To address R1: *"How does one create a framework for designing effective learning activities in a smartphone programming tutor?"*, I devised frameworks for designing programming activities for smartphones (Chapter 3), based on the literature review presented in Chapter 2. A framework for designing activities and for authoring the content was presented as both are vital for producing effective learning activities. I did not intend to address (R1) by conducting usability studies. Rather, the intension was to evaluate the activities for learning by investigating learning gains and other benefits. R1 was evaluated through my studies where participants interacted with PyKinetic for roughly one hour (Chapters 4-7). Apart from investigating learning gains, observations were made, and feedback were gathered from the participants on improving the UI of PyKinetic. Participants learned from the learning activities and generally found PyKinetic to be intuitive.

R2: *"What problem-solving strategies can be observed when solving Parsons problems?"* was addressed in Chapter 4. The results revealed that experts used optimal strategies such as moving each LOC to its correct position. On the contrary, novices used sub-optimal strategies like relying on superficial features without understanding the code by grouping the LOCs based on the level of the indentations.

R3: *"Which programming skills can be enhanced by a mobile tutor?"* was addressed in my evaluation studies. I found that procedural and conceptual coding skills can be supported by a mobile tutor. Self-explanation prompts were specifically effective for improving conceptual knowledge, as discussed in literature (Najar, Mitrovic, and McLaren, 2016). For procedural skills I found that several coding skills can be collectively improved by a mobile tutor such as debugging skills, output prediction skills, and code writing skills.

For R4: *"What combination of activities is effective for learning Python programming in a mobile tutor?"*, I did not particularly compare the versions of PyKinetic. Rather, based on results from a preceding study, I improved PyKinetic and extended it with more activities. Therefore, the latest version of PyKinetic (PyKinetic_Adaptive) is intended to be the most effective version of the tutor. I combined Parsons problems with incomplete LOCs and menu-based SE prompts, output prediction, and debugging activities. There could be more effective activities and combinations, but it is beyond the scope of my PhD research to explore other options.

For R5: *"What pedagogical strategies are effective for learning in PyKinetic?"*, several pedagogical strategies were implemented in PyKinetic, such as introducing self-explanation, combining activities of diverse nature, and adaptive problem selection. These pedagogical strategies were all effective for PyKinetic as evaluated in my studies (Chapters 5 and 6). I utilised the strategies mentioned above in the latest version of PyKinetic (PyKinetic_Adaptive), discussed in Chapter 7.

For my sixth research question R6: *"Which of the activities I developed are effective for improving code debugging and code writing skills?"*, PyKinetic_Fixed and PyKinetic_Adaptive both revealed to be effective specifically for enhancing code writing skills. Even due to the imbalance of participants' abilities in both groups, where there were more HP students who worked with PyKinetic_Fixed and more LP students who worked with PyKinetic_Adaptive, there were no significant difference with the normalised gains of both groups. On specific activities that support debugging skills, there were no activities or combination that was found to be effective. Participants improved on their coding skills collectively, but none were found to be proficient specifically for enhancing debugging skills. A possible explanation for this finding comes from the evidence in the literature that debugging requires a higher order of skill than code writing (Ahmadzadeh, Elliman, and Higgins, 2005; Fitzgerald et al., 2008). Furthermore, it is more difficult to debug another person's code and to perform it without tools. Another possibility is that it might be necessary to provide longer pre/post-test questions to get a better grasp of the abilities of a student, and

also have longer sessions in using PyKinetic.

For research question R7: *"What is the learning effectiveness of students with lower prior knowledge (LP) compared to those with higher prior knowledge (HP)?"* I investigated the differences between LP and HP students. PyKinetic_IncLOCs (Chapter 5) was more beneficial for LP students, while PyKinetic_DbgOut (Chapter 6) was more favourable for HP students. Furthermore, I found that PyKinetic_Adaptive (Chapter 7) was more beneficial for LP students. However, I propose that PyKinetic_Adaptive appeared to only be more beneficial for LP learners because of imbalanced abilities in each group due to random allocation of participants.

Finally, R8 which was "*What is the relationship between coding skills (code debugging, code tracing, and code writing) when solving problems in a mobile tutor for LP compared to HP students?"*, was addressed in my second evaluation study (Chapter 6), where I validated literature on coding skills. I compared the performance of participants in PyKinetic_DbgOut using their normalised gains and post-test scores by running several correlation tests. For LP students, I found evidence that they perform similarly when doing debugging, tracing, and fixing exercises. On the contrary, HP students only revealed a strong positive correlation on their debugging and tracing exercises. Code fixing exercises were found to be more related to code writing. However, this was only evident for HP students, because it is highly likely that LP students struggle with core programming concepts in general and therefore, performed similarly across exercises of various nature.

8.2 Contributions

One of the significant contributions of my PhD project is designing frameworks for programming activities specifically for a smartphone tutor and evaluating them. Programming tutors are not new, but most of them are designed for personal computers. Although mobile tutors for programming are also not new, these are limited and are mostly commercial applications. For the commercial mobile programming tutors, there are no available evaluations on their effectiveness.

My second contribution is investigating activities that are best suitable for certain programming skills such as debugging and code writing. The latest versions of PyKinetic revealed to be effective for code writing skills. In addition, I investigated the relationship of coding skills when solving problems on a mobile tutor. Results from the literature were verified when the relationship of coding skills differed between LP and HP students. Substantial evidence was also presented that Parsons problems were more effective for LP students while debugging exercises for HP students. Moreover, my other contribution is developing PyKinetic. The focus is not to deploy it as a standalone application but implementing PyKinetic was crucial for investigating my research questions about programming activities in a mobile tutor. Developing

PyKinetic allowed me to have the freedom to evaluate my research questions with complete flexibility.

Parsons problems are increasingly becoming popular, but most research on Parsons problems are either on paper or on personal computers. Therefore, I am contributing to knowledge by evaluating Parsons problems specifically on smartphones. I also developed my own variant of Parsons problems with incomplete LOCs, with the following features: 1) blocks of code are received and solved in the same area (instead of one area for solving, and another for receiving code); 2) indentations are provided as scaffolding; 3) all blocks of code contain single LOCs; 4) a menu-based self-explanation (SE) prompt (designed to facilitate deeper learning) is provided after solving an incomplete LOC. Furthermore, to the best of our knowledge I was the first one to combine Parsons problems with SE prompts. SE is widely known to be effective in various domains, as discussed in Chapter 2. My contribution is to combine SE with Parsons problems which are low cognitive load exercises, but still require a level of problem-solving skills unlike worked examples which are usually combined with SE. Furthermore, I evaluated SE prompts on a mobile tutor which has not been done to the best of our knowledge.

8.3 Limitations

As mentioned in Section 3.13, the difficulty of the problems in PyKinetic was determined on the bases of the following objective factors: the number of activities required in a problem, number of LOCs, number of topics covered, and levels of indentation. However, ordering the problems based on difficulty for my evaluation studies involved my own and other colleagues' perceptions of difficulty based on our Python programming experience. Therefore, this is a limitation of my research.

Another limitation is that the pre- and post-tests were not counterbalanced in Study 1 (Chapter 5). Therefore, there is a possibility that the pre-test might have been easier than the post-test. In Studies 2 and 3, half of the participants used Test A as their pre-test and Test B as post-test, while the other half used Test B as their pre-test and Test A as their post-test. Studies 2 and 3 were conducted in this way as an improvement to Study 1, and to eliminate the possibility that one test is easier than another. Furthermore, I have compared the pre-test scores of participants who took Test A to the pre-test scores of those who took Test B (separately for Study 2 and Study 3) and there were no significant differences between the scores, showing that the tests were of comparable difficulty.

In Study 1, some data was lost due to network connection issues. In this study, some participants used their own devices for the session. I did not anticipate that some phones were not consistently connected to the WiFi, so some of the logs were not sent through the server. I did not have this issue in my other studies because I have used our own development smartphones which sent logs through our server and at the same time stored the logs in the devices themselves as backup. I found that writing data on the device

itself was fault proof. Another limitation was during my third evaluation study (Chapter 7), there was an imbalance of abilities between the two groups due to random allocation. Furthermore, in all evaluation studies I had relatively small number of participants. It would have been ideal to have more participants for a richer dataset.

8.4 Further Directions

Like any other research, there are countless possibilities for future directions for my research project and for PyKinetic. One direction would be to deploy PyKinetic, so students can use it as an additional learning resource outside of traditional classrooms and labs. Another path would be to extend PyKinetic with more features and to improve its current specifications. One more possibility is to conduct further evaluation studies with any of the versions of PyKinetic presented in the thesis. Lastly, further analyses can be performed using data I have acquired from my evaluation studies. Possibilities for future work are discussed in more detail in the subsections to follow.

8.4.1 Improving PyKinetic

Menu-based SE revealed to be effective when combined with Parsons problems with incomplete LOCs. Ideally, this should also be implemented with my other activities such as when identifying erroneous LOCs, fixing erroneous LOCs, and predicting output. It will take time to author content for SE prompts for all my activities, but it would be interesting to evaluate a version of PyKinetic with SE for all activities. For instance, it might be interesting to extend PyKinetic_Adaptive to include SE for all activities.

The participants from the evaluation studies commented that they would have preferred more feedback. More specifically in my pilot and first evaluation studies (Chapters 4-5), participants suggested that more detailed feedback should be given when rearranging LOCs in Parsons problems. In my last study (Chapter 7), one pre-defined hint was given to participants which helped most. But, it could be improved by adding more specified hints for each problem. Other alternatives would be to add a help button which could be triggered by the learner to provide information about the topic/s being solved by the learner. I could also add personalised feedback where the detail in the feedback adjusts with the performance of the learner from the previous task like work of Ericson, Foley, and Rick (2018) with two-dimensional Parsons problems with distractors.

Some participants remarked that interacting with PyKinetic felt like playing a game or solving a quiz and have suggested that it might be good to add some gaming elements. Gamification was implemented in mobile programming commercial applications such as SoloLearn, (2019); DataCamp, (2019); Mimohello GmbH, (2019); Py, (2019); Zenva Pty Ltd, (2019). For example, I can add a points-based system that would

award learners a score depending on their performance in solving activities. For instance, the score can be calculated using the formula I used in my third evaluation study (Chapter 7). The learner is awarded an *AttemptScore* based on the number of attempts done to solve the problem compared to the minimum number of attempts (submissions) required to solve the problem. The learner is also given a *TimeScore* which is the quotient of ideal time to solve the activity (based on my previous evaluation studies) and the actual time that the student took to complete it. Both scores are then combined to calculate the final score for the activity with a penalty applied if the time per attempt is less than 10 seconds (0.17 min). To motivate the learners to gain high scores, a timer can be shown while solving an activity with their accumulated score displayed. Furthermore, scores can be translated into experience points (exp) which would continue to increase every time a learner uses PyKinetic. Learners can level up if a certain threshold of exp is met. In addition, badges can be awarded when learners reach a milestone or achieve something tremendous such as completing all Parsons problems. Other gaming elements that can supplement this is a shop available in the tutor where learners can use their exp as currency to redeem power-ups. Examples of power-ups could be one where learners gain perks while solving a problem for an allocated time such as earning double exp, and the ability to pause the tutor timer. Another option is to add a fictional character that the learner can customise with clothes and accessories from that can be redeemed similarly by using their exp as currency. I could also implement a leader board to motivate learners by competition. The leader board can show the status of the learner (i.e. level, number of badges, exp) and the characteristics of his/her fictional character. To implement the gaming elements discussed above, a significant amount of activities must be added to the database. Also, it is necessary to store the students' data in the database to allow them to save their progress and status in PyKinetic.

Another direction to improve PyKinetic is to implement it as an Intelligent Tutoring System (ITS). Intelligent Tutoring Systems (ITSs) are knowledge-based systems that simulate the behaviour of good human teachers (Woolf, 2010). ITSs have been proven effective in supporting student learning in different domains, e.g. (Heift and Nicholson, 2001; Melis et al., 2001; Mitrovic, 2003; Weber and Brusilovsky, 2001). Specifically, when learning a programming language, having access to an ITS is helpful for students to practice in their own time. Using an ITS also gives students the opportunity to receive personalised treatment through the feedback and activities they receive. Although it may not be as comprehensive as a human tutor, an intricately designed ITS may be able to target certain areas of the pedagogy especially when it is used effectively with other educational resources and tools. The activities included in the ITS can be designed to focus on increasing engagement, improving students' self-efficacy and motivation for learning. Some students may also find that using an ITS is less socially awkward since it does not involve directly communicating with a human tutor. This may then motivate them to use the ITS more in their own time and will therefore contribute to opportunities of gaining deeper understanding of the domain. I have

implemented a version of PyKinetic with adaptive problem selection. However, more components need to be added for PyKinetic to be considered as a full ITS.

8.4.2 Evaluation Studies

Other evaluation studies can be performed with all versions of PyKinetic presented in this thesis even without changing them. For example, my last study can be repeated with another set of participants. But participants should be allocated to either control or experimental group based on their pre-test scores to avoid the issue of having an imbalance of abilities. Furthermore, a study with a longer session time can be done. In my evaluation studies, the pre-/post-tests only contain questions that are answered in a short amount of time (15 minutes for Study 1, 18 minutes for Study 2-3). Perhaps the pre-/post-tests can be made longer by providing more questions and allow participants to complete it in a longer duration i.e. 30 minutes. Longer pre-/post-tests might enable us to acquire a broader view of the prior knowledge of the participants in comparison to their performance on the post-test. Also, a delayed post-test can be conducted a week after the evaluation study to investigate retention of knowledge.

Another possibility is to add more problems to PyKinetic to conduct a semester long study. The versions of PyKinetic presented in the thesis only contain enough problems for an hour of interaction with PyKinetic. I did not intend for PyKinetic to contain too much problems which would make it impossible for all of them to be completed within one experimental session. The intention was not for participants to have an impulse to finish all given problems. Furthermore, it might end up causing frustrations when they can only solve a proportion of the activities offered.

8.4.3 Analyses

I have mostly performed statistical analysis with the data from my studies. However, it is possible for other analyses to be conducted. For example, it might be interesting to investigate further on the patterns that learners use in solving Parsons problems. In my pilot study (Chapter 4), I recorded the students' actions which contains the line by line reordering of LOCs made by participants when solving Parsons problems with distractors. Although I have a small sample size in my pilot study, students solved multiple problems each which would be enough to build a model on the patterns used in solving Parsons problems with distractors. In my first and third evaluation studies (Chapters 5 and 7) I also have similar data in the logs recorded for students solving regular Parsons problems and Parsons problems with incomplete LOCs. It is possible to inspect the data and extract the patterns used in solving regular Parsons problems and Parsons problems with incomplete LOCs. Furthermore, it would be interesting to compare the patterns used by students when solving regular Parsons problems, Parsons problems with distractors, and Parsons problems with incomplete LOCs. There might also be a difference in the strategies that students used in solving

Parsons problems with incomplete LOCs depending on the version of PyKinetic (PyKinetic_IncLOCs, PyKinetic_Fixed, and PyKinetic_Adaptive). I speculate that there might be a difference because PyKinetic_Fixed and PyKinetic_Adaptive also contain other types of activities (output prediction and debugging), whereas PyKinetic_IncLOCs only contains Parsons problems with incomplete LOCs. In addition, I suspect that the strategies used by learners when solving Parsons problems with incomplete LOCs could also differ based on whether they received adaptive problem selection. Participants who used PyKinetic_Adaptive may improve their strategies when solving Parsons problems with incomplete LOCs as they learn on how to solve them better especially when they receive problems of that same type. On the contrary, learners using PyKinetic_Fixed only got two Parsons problems with incomplete LOCs and therefore it is probably not enough to exhibit an improved behaviour. Lastly, based on my observations from the recorded videos in my pilot study (Chapter 4), it might be interesting to also compare strategies used by LP students compared to HP students. I expect that HP students will demonstrate more optimal strategies when solving variants of Parsons problems compared to LP students. Furthermore, it might be to investigate the strategies they use in comparison to tutors (experts).

Lastly, I have other raw data which I have not analysed. In Studies 1 and 3, I performed similar evaluation studies with high school students. A future direction would be to process and conduct statistical analyses with this data. I also performed the Baker Rodrigo Ocumpaugh monitoring protocol BROMP 2.0 (Ocumpaugh, 2015) in Study 1 with participants from Ateneo de Manila university. BROMP specifies how human observer should note affective states of learners in a classroom. A future direction is to process and analyse the raw data from this study and investigate the emotions manifested when solving Parsons problems with incomplete LOCs in PyKinetic.

BIBLIOGRAPHY

Adams, D. M., McLaren, B. M., Durkin, K., Mayer, R. E., Rittle-Johnson, B., Isotani, S., & Van Velsen, M. (2014). Using erroneous examples to improve mathematics learning with a web-based tutoring system. Computers in Human Behavior, 36, 401-411.

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.

Ainsworth, S., & Loizou, A. T. (2003). The effects of self-explaining when learning with text or diagrams. Cognitive Science: A Multidisciplinary Journal, 27, 669–681.

Aleven, V. A., & Koedinger, K. R. (2002). An effective metacognitive strategy: Learning by doing and explaining with a computer-based Cognitive Tutor. Cognitive science, 26(2), 147-179.

Aleven, V., & Koedinger, K. R. (2000). The need for tutorial dialog to support self-explanation. In Building dialogue systems for tutorial applications, papers of the 2000 AAAI Fall Symposium (pp. 65-73).

Aleven, V., Koedinger, K. R., & Cross, K. (1999) Tutoring answer explanation fosters learning with understanding. In: Lajoie, S.P. and Vivet, M. (Eds.), Proc. 9th Int. Conf. Artificial Intelligence in Education, (pp. 199-206). IOS Press.

Aleven, V., Ogan, A., Popescu, O., Torrey, C., & Koedinger, K. (2004). Evaluating the effectiveness of a tutorial dialogue system for self-explanation. In International Conference on Intelligent Tutoring Systems (pp. 443-454). Springer, Berlin, Heidelberg.

Anderson, J. R. (1982). Acquisition of cognitive skill. Psychological review, 89(4), p. 369.

Anshari, M., Almunawar, M. N., Shahrill, M., Wicaksono, D. K., & Huda, M. (2017). Smartphones usage in the classrooms: Learning aid or interference? Education and Information Technologies, 22(6), pp.3063-3079.

ApkZube. Learn Python Programming. [cited 22 May 2019]; Available from: https://play.google.com/store/apps/details?id=com.apkzube.learnpython

Areias, C., & Mendes, A. (2007). A tool to help students to develop programming skills. In Proceedings of the 2007 international conference on Computer systems and technologies (p. 89-95). ACM.

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. Journal of Educational Psychology, 95(4), pp.774-783.

Au, M., Lam, J., & Chan, R. (2015). Social media education: Barriers and critical issues. In Technology in Education. Transforming Educational Practices with Technology (pp. 199–205). Springer Berlin Heidelberg.

Bari, A. G., Gaspar, A., Wiegand, R. P., Albert, J. L., Bucci, A., & Kumar, A. N. (2019). EvoParsons: design, implementation and preliminary evaluation of evolutionary Parsons puzzle. Genetic Programming and Evolvable Machines, 20: (pp. 213-244).

Barria-Pineda, J., Guerra, J., Huang, Y., & Brusilovsky, P. (2017, March). Concept-level knowledge visualization for supporting self-regulated learning. In Proceedings of the 22nd International Conference on intelligent user interfaces companion (pp. 141-144). ACM.

Bennett, S., Maton, K., & Kervin, L. (2008). The 'digital natives' debate: A critical review of the evidence. British journal of educational technology, 39(5), 775-786.

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.

Bielaczyc, K., Pirolli, P., & Brown, A.L. (1993) Training in Self-Explanation and Self-Regulation Strategies: Investigating the Effects of Knowledge Acquisition Activities on Problem-solving. Cognition and Instruction, 13(2), 221-252.

Blackwell, A. F. (2002). What is programming?. In PPIG (pp. 204-218).

Boticki, I., Barisic, A., Martin, S., & Drljevic, N. (2013). Teaching and learning computer science sorting algorithms with mobile devices: A case study. Computer Applications in Engineering Education, 21(S1), E41- E50.

Browne, K., & Anand, C. (2013, October). Gamification and serious game approaches for introductory computer science tablet software. In Proceedings of the First International Conference on Gameful Design, Research, and Applications (pp. 50-57). ACM.

Butcher, K. R. (2006). Learning from text with diagrams: Promoting mental model development and inference generation. Journal of Educational Psychology, 98(1), 182-197.

Cass, S., & Parthasaradhi, B. (2018). Interactive: The Top Programming Languages 2018. https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2018. Accessed 4 Sept 2018.

Chamberland, M., St-Onge, C., Setrakian, J., Lanthier, L., Bergeron, L., Bourget, A., ... & Rikers, R. (2011). The influence of medical students' self-explanations on diagnostic performance. Medical Education, 45(7), 688-695.

Chi, M. T. (2000). Self-explaining expository texts: The dual processes of generating inferences and repairing mental models. Advances in instructional psychology, 5, 161-238.

Chi, M. T., & Wylie, R. (2014). The ICAP framework: Linking cognitive engagement to active learning outcomes. Educational Psychologist, 49(4), 219-243.

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.

Chi, M. T., De Leeuw, N., Chiu, M. H., & LaVancher, C. (1994). Eliciting self-explanations improves understanding. Cognitive science, 18(3), 439-477.

Chi, M.T., R. Glaser, and E. Rees, Expertise in problem solving. (1981), DTIC Document.

Chmiel, R., & Loui, M. C. (2004, March). Debugging: from novice to expert. In ACM SIGCSE Bulletin (Vol. 36, No. 1, pp. 17-21). ACM.

Coding and Programming App. Programming Hub: Learn to Code. [cited 22 May 2019]; Available from: https://play.google.com/store/apps/details?id=com.freeit.java

Conati, C., & Vanlehn, K. (2000). Toward computer-based support of meta-cognitive skills: A computational framework to coach self-explanation. International Journal of Artificial Intelligence in Education (IJAIED), 11, 389-415.

DataCamp. DataCamp – Learn R, Python & SQL. [cited 22 May 2019]; Available from: https://play.google.com/store/apps/details?id=com.datacamp

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.

Ducasse, M., & Emde, A. M. (1988, April). A review of automated debugging systems: knowledge, strategies and techniques. In Proceedings of the 10th international conference on Software engineering (pp. 162-171). IEEE Computer Society Press.

Dukic, Z., Chiu, D. K., & Lo, P. (2015). How useful are smartphones for learning? Perceptions and practices of Library and Information Science students from Hong Kong and Japan. Library Hi Tech, 33(4), 545-561.

Enki.com. Enki: Learn data science, coding, tech skills. [cited 22 May 2019]; Available from: https://play.google.com/store/apps/details?id=com.enki.insights

Ericson, B. J., Foley, J. D., & Rick, J. (2018, August). Evaluating the Efficiency and Effectiveness of Adaptive Parsons Problems. In Proceedings of the 2018 ACM Conference on International Computing Education Research(pp. 60-68). ACM.

Ericson, B. J., Margulieux, L. E., & Rick, J. (2017). Solving Parsons problems versus fixing and writing code. In

Proc.17th Koli Calling Int. Conf. Computing Education Research (pp. 20-29). ACM.

Ericsson, K. A., & Simon, H. A. (1993). Protocol Analysis: Verbal Reports as Data (Revised Ed). Cambridge: MIT Press.

Fabic, G. V. F., Mitrovic, A., & Neshatian, K. (2017a). Investigating the effectiveness of menu-based self-explanation prompts in a mobile Python Tutor. In: E. Andre, R. Baker, X. Hu, M. Rodrigo, B. du Boulay (Eds.), Proc. 18th International Conference on Artificial Intelligence in Education (pp. 498-501). Springer, Cham.

Fabic, G. V. F., Mitrovic, A., & Neshatian, K. (2017b). Learning with Engaging Activities via a Mobile Python Tutor. In International Conference on Artificial Intelligence in Education (pp. 613-616). Springer, Cham.

Fabic, G. V. F., Mitrovic, A., & Neshatian, K. (2018a). Adaptive Problem Selection in a Mobile Python Tutor. In Adjunct Publication of the 26th Conference on User Modeling, Adaptation and Personalization (pp. 269-274). ACM.

Fabic, G. V. F., Mitrovic, A., & Neshatian, K. (2018b). Supporting Novices and Advanced Students in Acquiring Multiple Coding Skills. In Proc. 26th International Conference on Computers in Education, (pp. 65-70).

Fabic, G. V. F., Mitrovic, A., & Neshatian, K. (2018c). Investigating the effects of learning activities in a mobile Python tutor for targeting multiple coding skills. Research and practice in technology enhanced learning, 13(1), 23.

Fabic, G., Mitrovic A., & Neshatian, K. (2017c) A comparison of different types of learning activities in a mobile Python tutor. Proc. 25th International Conference on Computers in Education, (pp. 604-613).

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, (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 PQTEL 2016, pp. 434-444, APSCE.

Fitzgerald, S., Lewandowski, G., McCauley, R., Murphy, L., Simon, B., Thomas, L., & Zander, C. (2008). Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers. Computer Science Education, 18(2), 93-116.

Gadgil, S., Nokes-Malach, T. J., & Chi, M. T. (2012). Effectiveness of holistic mental model confrontation in driving conceptual change. Learning and Instruction, 22, 47–61.

Garner, S. (2007). An exploration of how a technology-facilitated part-complete solution method supports the

learning of computer programming. Issues in Informing Science & Information Technology, 4, 491-502.

Gavali, M. Y., Khismatrao, D. S., Gavali, Y. V., & Patil, K. B. (2017). Smartphone, the new learning aid amongst medical students. Journal of clinical and diagnostic research, 11(5), p.JC05.

GitHub. bauerca/drag-sort-listview [cited 8 June 2019a]; Available from: https://github.com/bauerca/drag-sort-listview.

GitHub. jgilfelt/android-sqlite-asset-helper [cited 8 June 2019b]; Available from: https://github.com/jgilfelt/android-sqlite-asset-helper.

Gould, J. D. (1975). Some psychological evidence on how people debug computer programs. International Journal of Man-Machine Studies, 7(2), 151-182.

Gould, J. D., & Drongowski, P. (1974). An exploratory study of computer program debugging. Human Factors, 16(3), 258-277.

Grandl, M., Ebner, M., Slany, W., & Janisch, S. (2018). It's in your pocket: A MOOC about programming for kids and the role of OER in teaching and learning contexts. Open Education Global Conference. Delft University of Technology

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.

Hailpern, B., & Santhanam, P. (2002). Software debugging, testing, and verification. IBM Systems Journal, 41(1), 4-12.

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.

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

Heift, T., & Nicholson, D. (2001). Web delivery of adaptive and interactive language tutoring. International Journal of Artificial Intelligence in Education, 12(4), 310-325.

Helminen, J., Ihantola, P., Karavirta, V., & Malmi, L. (2012). How do students solve parsons programming problems?: An analysis of interaction traces. In Proceedings of the ninth annual international conference on International computing education research (pp. 119-126). ACM.

Hosseini, R. (2018). Program Construction Examples in Computer Science Education: From Static Text to Adaptive

and Engaging Learning Technology (Doctoral dissertation, University of Pittsburgh).

Hsu, C. Y., Tsai, C. C., & Wang, H. Y. (2012). Facilitating third graders' acquisition of scientific concepts through digital game-based learning: The effects of self-explanation principles. The Asia-Pacific Education Researcher, 21(1),71-82.

Hürst, W., Lauer, T., & Nold, E. (2007, March). A study of algorithm animations on mobile devices. In ACM SIGCSE Bulletin (Vol. 39, No. 1, pp. 160-164). ACM.

Hwang, G. J., & Chang, S. C. (2016). Effects of a peer competition-based mobile learning approach on students' affective domain exhibition in social studies courses. British Journal of Educational Technology, 47(6), 1217-1231.

Ihantola, P., & Karavirta, V. (2011). Two-dimensional parson's puzzles: The concept, tools, and first observations. Journal of Information Technology Education, 10, 119-132.

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.

Jenkins, T. (2002, August). On the difficulty of learning to program. In Proceedings of the 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences (Vol. 4, No. 2002, pp. 53-58).

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.

Jones, A. C., Scanlon, E., & Clough, G. (2013). Mobile learning: Two case studies of supporting inquiry learning in informal and semiformal settings. Computers & Education, 61, 21-32.

Karavirta, V. Quiz & Learn Python - Learning Python on Mobile Devices. [cited 2019 25 May 2019]; Available from: http://www.villekaravirta.com/projects/quizlearn-python/.

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.

Kim, H., & Kwon, Y. (2012). Exploring smartphone applications for effective mobile-assisted language learning. Multimedia-Assisted Language Learning, 15(1), 31-57.

Klopfer, E., Osterweil, S., Groff, J., & Haas, J. (2009). Using the technology of today in the classroom today: The instructional power of digital games, social networking, simulations and how teachers can leverage them. The Education Arcade, 1, 20.

Klopfer, E., Sheldon, J., Perry, J., & Chen, V. H. (2012). Ubiquitous games for learning (UbiqGames): Weatherlings,

a worked example. Journal of Computer Assisted Learning, 28(5), 465-476.

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

Kumar, A. N. (2019). Representing and Evaluating Strategies for Solving Parsons Puzzles. In International Conference on Intelligent Tutoring Systems (pp. 193-203). Springer, Cham.

Kwon, K., Kumalasari, C. D., & Howland, J. L. (2011). Self-explanation prompts on problem-solving performance in an interactive learning environment. Journal of Interactive Online Learning, 10(2), 96-112.

Lahtinen, E., Ala-Mutka, K., & Järvinen, H. M. (2005). A study of the difficulties of novice programmers. Acm Sigcse Bulletin, 37(3), 14-18.

Learn Programming, Programming App, Coding App. Programming Hero: Coding App Just Got Fun (beta). [cited 22 May 2019]; Available from: https://play.google.com/store/apps/details?id=com.learnprogramming.codecamp

LeBlanc, T. J., & Mellor-Crummey, J. M. (1986). Debugging Parallel Programs with Instant Replay (No. TR- 194). ROCHESTER UNIV NY DEPT OF COMPUTER SCIENCE.

Lee, M. J., Ko, A. J., & Kwan, I. (2013, August). In-game assessments increase novice programmers' engagement and level completion speed. In Proceedings of the ninth annual international ACM conference on International computing education research (pp. 153-160). ACM.

Linn, M. C., & Dalbey, J. (1985). Cognitive consequences of programming instruction: Instruction, access, and ability. Educational Psychologist, 20(4), 191-206.

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.

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.

Lister, Raymond, Beth Simon, Errol Thompson, Jacqueline L. Whalley, and Christine Prasad. (2006). "Not seeing the forest for the trees: novice programmers and the SOLO taxonomy." ACM SIGCSE Bulletin 38, no. 3: 118-122.

Liu, T. C., Lin, Y. C., Tsai, M. J., & Paas, F. (2012). Split-attention and redundancy effects on mobile learning in physical environments. Computers & Education, 58(1), 172-180.

Lokar, M., & Pretnar, M. (2015, November). A low overhead automated service for teaching programming. In Proceedings of the 15th Koli Calling Conference on Computing Education Research (pp. 132-136). 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.

Lye, S. Y., & Koh, J. H. L. (2014). Review on teaching and learning of computational thinking through programming: What is next for K-12?. Computers in Human Behavior, 41, 51-61.

Marx, J. D., & Cummings, K. (2007). Normalized change. American Journal of Physics, 75(1), 87-91.

Mason, R., & Cooper, G. (2014). Introductory Programming Courses in Australia and New Zealand in 2013- trends and reasons. In Proc. 16th Australasian Computing Education Conference-Volume 148 (pp. 139-147). Australian Computer Society, Inc..

Matthews, P., & Rittle-Johnson, B. (2009). In pursuit of knowledge: Comparing self-explanations, concepts, and procedures as pedagogical tools. Journal of experimental child psychology, 104(1), 1-21.

Mayer, R. E., & Johnson, C. I. (2010). Adding instructional features that promote learning in a game-like environment. Journal of Educational Computing Research, 42(3), 241-265.

Mayer, R.E., Multimedia learning. 2009: Cambridge university press.

Mbogo, C., Blake, E., & Suleman, H. (2016). Design and use of static scaffolding techniques to support Java programming on a mobile phone. In Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education (pp. 314-319). ACM.

McArthur, D., Stasz, C., Hotta, J., Peter, O., & Burdorf, C. (1988). Skill-oriented task sequencing in an intelligent tutor for basic algebra. Instructional Science, 17(4), 281-307.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B. D., ... & Wilusz, T. (2001, December). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. In Working group reports from ITiCSE on Innovation and technology in computer science education (pp. 125-180). ACM.

McLaren, B. M., Adams, D. M., & Mayer, R. E. (2015). Delayed learning effects with erroneous examples: a study of learning decimals with a web-based tutor. International Journal of Artificial Intelligence in Education, 25(4), 520-542.

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.

Melis, E., Andres, E., Budenbender, J., Frischauf, A., Goduadze, G., Libbrecht, P., ... & Ullrich, C. (2001). ActiveMath: A generic and adaptive web-based learning environment. International Journal of Artificial

Intelligence in Education (IJAIED), 12, 385-407.

Mimohello GmbH. Mimo: Learn to Code. [cited 22 May 2019]; Available from: https://play.google.com/store/apps/details?id=com.getmimo

Mitrovic, A. (2003) Supporting Self-Explanation in a Data Normalization Tutor. In: V. Aleven, U. Hoppe, J. Kay, R. Mizoguchi, H. Pain, F. Verdejo, K. Yacef (eds) Supplementary proceedings, AIED 2003, pp. 565-577, 2003.

Mitrovic, A. (2003). An intelligent SQL tutor on the web. International Journal of Artificial Intelligence in Education, 13(2-4), 173-197.

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

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.

Murphy, L., Lewandowski, G., McCauley, R., Simon, B., Thomas, L., & Zander, C. (2008, March). Debugging: the good, the bad, and the quirky--a qualitative analysis of novices' strategies. In ACM SIGCSE Bulletin (Vol. 40, No. 1, pp. 163-167). ACM.

Myers, G. J. (1978). A controlled experiment in program testing and code walkthroughs/inspections. Communications of the ACM, 21(9), 760-768.

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.

Nakaya, K., & Murota, M. (2013). Development and evaluation of an interactive English conversation learning system with a mobile device using topics based on the life of the learner. Research & Practice in Technology Enhanced Learning, 8(1), 65-89.

Nathan, M. J., Mertz, K., & Ryan, R. (1994). Learning through self-explanation of mathematics examples: Effects of cognitive load (poster). In Presentation to the American Educational Research Association (AERA) annual meeting.

Nouri, J., Cerratto-Pargman, T., Rossitto, C., & Ramberg, R. (2014). Learning with or without mobile devices? A comparison of traditional school field trips and inquiry-based mobile learning activities. Research & Practice in Technology Enhanced Learning, 9(2), 241-262.

O'Neil, H. F., Chung, G. K., Kerr, D., Vendlinski, T. P., Buschang, R. E., & Mayer, R. E. (2014). Adding self-explanation prompts to an educational computer game. Computers in Human Behavior, 30, 23-28.

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, Colourado.: EDUCAUSE., pp. 12-31.

Ocumpaugh, J. (2015). Baker Rodrigo Ocumpaugh monitoring protocol (BROMP) 2.0 technical and training manual. New York, NY and Manila, Philippines: Teachers College, Columbia University and Ateneo Laboratory for the Learning Sciences, 60.

O'Malley, C., Vavoula, G., Glew, J. P., Taylor, J., Sharples, M., Lefrere, P., & Waycott, J. (2005). Guidelines for learning/teaching/tutoring in a mobile environment. Public deliverable from the MOBILearn project.

Oyelere, S. S., Suhonen, J., Wajiga, G. M., & Sutinen, E. (2018). Design, development, and evaluation of a mobile learning application for computing education. Education and Information Technologies, 23(1), 467-495.

Park, Y. (2011). A pedagogical framework for mobile learning: Categorizing educational applications of mobile technologies into four types. International Review of Research in Open and Distributed Learning, 12(2), 78-102.

Parsons, D., & Haden, P. (2006). Parson's programming puzzles: a fun and effective learning tool for first programming courses. In Proc. 8th Australasian Conf. Computing Education,52 (pp. 157-163).

Parsons, D., Wood, K., & Haden, P. (2015). What Are We Doing When We Assess Programming?. In Proceedings of the 17th Australasian Computing Education Conference (ACE 2015) (Vol. 27, p. 119-127).

Pashler, H., Bain, P. M., Bottge, B. A., Graesser, A., Koedinger, K., McDaniel, M., & Metcalfe, J. (2007). Organizing Instruction and Study to Improve Student Learning. IES Practice Guide. NCER 2007-2004. National Center for Education Research.

Pea, R. D. (1986). Language-independent conceptual “bugs” in novice programming. Educational Computing Research, 2(1), 25-36.

Pea, R. D., & Maldonado, H. (2006). WILD for learning: Interacting through new computing devices anytime, anywhere. The Cambridge Handbook of the Learning Sciences. New York: Cambridge University Press, 852-886.

Perry, J., & Klopfer, E. (2014). UbiqBio: Adoptions and outcomes of mobile biology games in the ecology of school. Computers in the Schools, 31(1-2), 43-64.

Piteira, M., & Costa, C. (2013). Learning computer programming: study of difficulties in learning programming. In Proceedings of the 2013 International Conference on Information Systems and Design of Communication (pp. 75-80). ACM.

Py. Py. [cited 22 May 2019]; Available from: https://play.google.com/store/apps/details?id=com.py.learn

Quinson, M., & Oster, G. (2015). A Teaching System to Learn Programming: the Programmer's Learning Machine. In Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education (pp. 260-265). ACM.

Quiz & Learn Python - Learn Python on Your Mobile. [cited 2019 25 May 2019]; Available from: http://mobileicecube.com/quiz-learn-python/.

Rau, M. A., Aleven, V., & Rummel, N. (2009). Intelligent Tutoring Systems with Multiple Representations and Self-Explanation Prompts Support Learning of Fractions. In AIED (pp. 441-448).

Rau, M. A., Aleven, V., & Rummel, N. (2015). Successful learning with multiple graphical representations and self-explanation prompts. Journal of Educational Psychology, 107(1), 30-46.

Research New Zealand (2015). A Report on a Survey of New Zealanders' Use of Smartphones and other Mobile Communication Devices. Available from: http://www.researchnz.com/pdf/Special%20Reports/Research%20New%20Zealand%20Special%20Report%20-%20Use%20of%20Smartphones.pdf

Rittle-Johnson, B. (2006). Promoting transfer: Effects of self-explanation and direct instruction. Child development, 77(1), 1-15.

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A review and discussion. Computer science education, 13(2), 137-172.

Roschelle, J., Rafanan, K., Bhanot, R., Estrella, G., Penuel, B., Nussbaum, M., & Claro, S. (2010). Scaffolding group explanation and feedback with handheld technology: impact on students' mathematics learning. Educational Technology Research and Development, 58(4), 399-419.

Runestone Interactive Overview. [cited 2019 25 May 2019]; Available from: http://interactivepython.org/runestone/static/overview/overview.html.

Shaffer, S. C. (2005). Ludwig: an online programming tutoring and assessment system. ACM SIGCSE Bulletin, 37(2), 56-60.

Shih, J. L., Chuang, C. W., & Hwang, G. J. (2010). An inquiry-based mobile learning approach to enhancing social science learning effectiveness. Journal of Educational Technology & Society, 13(4), 50-62.

SoloLearn. Learn Python. [cited 22 May 2019]; Available from: https://play.google.com/store/apps/details?id=com.sololearn.python

Someren, M. V., Barnard, Y. F., & Sandberg, J. A. (1994). The think aloud method: a practical approach to modelling cognitive processes. Academic Press.

Sorva, J. (2013). Notional machines and introductory programming education. ACM Transactions on Computing Education (TOCE), 13(2), 8.

Su, C. H., & Cheng, C. H. (2015). A mobile gamification learning system for improving the learning motivation and achievements. Journal of Computer Assisted Learning, 31(3), 268-286.

Sullivan, F.a., *NZ will have 90% smartphone and 78% tablet ownership by 2018*. 2013, Scoop Media.

Sun, D., & Looi, C. K. (2017). Focusing a mobile science learning process: difference in activity participation. Research and Practice in Technology Enhanced Learning, 12(1), 3.

Sun, J. C. Y., Chang, K. Y., & Chen, Y. H. (2015). GPS sensor-based mobile learning for English: An exploratory study on self-efficacy, self-regulation and student achievement. Research and Practice in Technology Enhanced Learning, 10(1), 23.

Sweller, J., P. Ayres, and S. Kalyuga, Cognitive Load Theory. 2011: Springer New York.

Sweller, J., Van Merrienboer, J. J., & Paas, F. G. (1998). Cognitive architecture and instructional design. Educational psychology review, 10(3), 251-296.

Tan, P. H., Ting, C. Y., & Ling, S. W. (2009). Learning difficulties in programming courses: undergraduates' perspective and perception. In 2009 International Conference on Computer Technology and Development (Vol. 1, pp. 42-46). IEEE.

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.

van der Meij, J., & de Jong, T. (2011). The effects of directive self-explanation prompts to support active processing of multiple representations in a simulation-based learning environment. Journal of Computer Assisted Learning, 27, 411– 423

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

Vessey, I. (1985). Expertise in debugging computer programs: A process analysis. International Journal of Man-Machine Studies, 23(5), 459-494.

Vinay, K. V., & Vishal, K. (2013). Smartphone applications for medical students and professionals. Nitte University Journal of Health Science, 3(1), 59-62.

Vinay, S., Vaseekharan, M., & Mohamedally, D. (2013). RoboRun: A gamification approach to control flow learning for young students with TouchDevelop. arXiv preprint arXiv:1310.0810.

Wang, M., Shen, R., Novak, D., & Pan, X. (2009). The impact of mobile learning on students' learning behaviours and performance: Report from a large blended classroom. British Journal of Educational Technology, 40(4), 673-695.

Wang, Y. H. (2016). Could a mobile-assisted learning system support flipped classrooms for classical Chinese learning?. Journal of Computer Assisted Learning, 32(5), 391-415.

Weber, G., & Brusilovsky, P. (2001). ELM-ART: An adaptive versatile system for Web-based instruction. International Journal of Artificial Intelligence in Education (IJAIED), 12, 351-384.

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

Weerasinghe, A., Mitrovic, A. (2006b). Individualizing Self-Explanation Support for Ill-Defined Tasks in Constraint-based Tutors. V. Aleven, K. Ashley, Lynch, C., Pinkwart, N. (eds) ITS 2006 workshop on ITS for Ill-defined domains, pp. 56-64, Jhongli, Taiwan, 26-30.6.2006.

Weerasinghe, A., Mitrovic, A., Martin, B. (2007). Towards a general model of self-explanation to enhance learning. R. Luckin, K. Koedinger, J. Greer (eds) Proc. 13th Int. Conf. Artificial Intelligence in Education AIED 2007, Los Angeles, 2007: 665-667.

Wen, C., & Zhang, J. (2015). Design of a microlecture mobile learning system based on smartphone and web platforms. IEEE Transactions on Education, 58(3), 203-207.

Winslow, L. E. (1996). Programming pedagogy—a psychological overview. ACM SIGCSE Bulletin, 28(3), 17- 22.

Wong, L. H., Looi, C. K., & Boticki, I. (2017). Improving the design of a mCSCL Chinese character forming game with a distributed scaffolding design framework. Research and Practice in Technology Enhanced Learning, 12(1), 27.

Woolf, B. P. (2010). Building intelligent interactive tutors: Student-centered strategies for revolutionizing e- learning. Morgan Kaufmann.

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.

Yoon, B. D., & Garcia, O. N. (1998, March). Cognitive activities and support in debugging. In Proceedings Fourth Annual Symposium on Human Interaction with Complex Systems (pp. 160-169). IEEE.

Zenva Pty Ltd. Codemurai – Learn Programming. [cited 22 May 2019]; Available from: https://play.google.com/store/apps/details?id=com.zenva.codemurai

Zingaro, D., Cherenkova, Y., Karpova, O., & Petersen, A. (2013, March). Facilitating code-writing in PI classes. In Proceeding of the 44th ACM technical symposium on Computer science education (pp. 585-590). ACM.

Appendix A. Pre-tests and Post-tests

First Evaluation Study Pre-test

1. Circle all **syntactically valid** Python strings from the following:
    a. `'Herbert the Heffalump'`
    b. `"He said "Hi!" to me"`
    c. `"""IAma$tr!nG, d0c$tr!nG…^_^"""`
    d. `"'Oh no!', He exclaimed.'`
    e. `'It\'s raining and pouring, the sun\'s never returning.'`
    f. `"One line/nTwo lines/nThree lines/nFour lines/n"`

2. The following code snippet produces an error. Why this is the case? (Circle the answer)

```
a = "b"
a = 49
print("qr" + a + "st")
```

    a. `"qr"` and `"st"` are not variables. Only variables can use '+' operator.
    b. You cannot use the '+' operator on string values.
    c. The '+' operator only works on values with the same type. i.e. variable a in this case is an int while the others are strings.
    d. Pylint does not like variable `a` since its name is too short.

3. Circle all that apply about `if` conditional statements (compared to `elif` conditional statements).
    a. `if` statements are normally used when conditions are ***mutually inclusive*** i.e. if -> if -> else (the two if statements are mutually inclusive)
    b. `if` statements are normally used when conditions are ***mutually exclusive*** i.e. if -> if -> else (the two if statements are mutually exclusive)
    c. `if` statements are used at the start of a set of conditional clauses
    d. `if` statements are used at the end of a set of conditional clauses

4. What is the output of the following code snippet:

```
number = 42
message = "hello"
if number > 16:
    print("I am greater than 16.")
if number > 35 and message == "HELLO":
    print("I am greater than 35.")
elif number >= 40 or message == "hello":
    print("Comparing to 40.")
else:
    print("Today is a good day.")
```

5. `data` is a variable with a list value and `n` has a value of 1. Circle all code statements that will give variable `my_var` a non-empty list value. Assume that data contains at least two elements.
    a. `my_var = data[n]`
    b. `my_var = data[0:n]`
    c. `my_var = data[-1]`
    d. `my_var = data[:-1]`
    e. `my_var = data[-1:]`

6. Rearrange the following lines and fill in the missing elements of the function `three_odds`() which takes a list `numbers` as an argument and returns the first three odd elements of the `numbers`. The test cases must also be rearranged to match the expected output. Note: Assume that the input contains at least three odd numbers. Write the line numbers to answer.

    Expected Output:
    ```
    [5, 9, 11]
    [11, 39, 1]
    [11, 9, 3]
    ```

    Code:
    ```
    print(three_odds([2,4,10,11,9,3,29]))
    def three_odds(numbers):
        return result
            count += 1
            if numbers[count] % 2 != 0:
        result = []
    print(three_odds([11,39,10,12,84,1]))
                result.append(numbers[count])
        while len(______) < 3:
        count = 0
    print(three_odds([5,9,10,11,39,10]))
        """Return the first three odd numbers in a list"""
    ```

7. You need to always specify a counter when using a for loop. i.e. `i += 1`. True or False?

    True False

8. A tuple can contain a mix of data types. Therefore, the following is valid:
    `(('hello', 32, 93.0), [30, 39, 2984, 39])`

    True False

First Evaluation Study Post-test

1. Circle all **syntactically valid** Python variables from the following:
    a. `Variable`
    b. `MyTuple`
    c. `iAmAvar$^&`
    d. `x_y`
    e. `a1`
    f. `1a`

2. The following code snippet produces an error. Why this is the case? (Circle the answer)

```
x = 19
x = [17, 39, 40, 289]
print(20 + x + 60)
```

    a. You cannot use the `'+'` operator on variables.
    b. Pylint does not like variable `x` since its name is too short.
    c. 20 and 60 are ints, the `'+'` operator can only be used on strings.
    d. The value that `x` is referencing was changed to a list. `x` has to also be an int to use the `'+'` operator with 20 and 60.

3. Circle all that apply about `elif` compared to `if` conditional statements.
    a. `elif` statements do not need a condition when used i.e. `elif:` is valid
    b. `elif` statements requires a condition when used i.e. `elif:` is **not** valid
    c. `elif` statements are normally used when conditions are ***mutually inclusive*** i.e. if -> elif -> else (the if and elif statements are mutually inclusive)
    d. `elif` statements are normally used when conditions are ***mutually exclusive*** i.e. if -> elif -> else (the if and elif statements are mutually exclusive)

4. What is the output of the following code snippet:

```
message = "hi"
my_list = [8, 45, 90, 23]

if my_list[-1] > 16:
    print("I am greater than 16.")
if my_list[2] < 20 or message == "hiho":
    print("I am less than 20.")
elif my_list[3] >= 40 and message == "hi":
    print("Comparing to 40.")
else:
    print("Tomorrow was a good day.")
```

5. Suppose `data = ["hello", "hi", "greetings", "kia ora"]` and `message` has a string value. Circle all code statements that will give variable `my_var` a string value.
    a. `my_var = data[0:1]`
    b. `my_var = data["kia ora"]`
    c. `my_var = data[-1]`
    d. `my_var = data[1][-1]`
    e. `my_var = data[message]`

6. Rearrange the following lines of the function `abs_reversed()` which takes a list `numbers` as an argument and returns the absolute values of the elements in the list `numbers` in a reverse order. The test cases must also be rearranged to match the expected output.

    Expected Output:

```
[10, 37, 12, 20, 0]
[12, 11, 10, 29, 40]
[10, 39, 8, 3]
```

    Code:

```
1       result = []
2       for i in range(len(numbers)-1, -1, -1):
3   abs_reversed([3,-8,39,10])
4   def abs_reversed(numbers):
5           if numbers[i] < 0:
6   abs_reversed([40,-29,-10,-11,-12])
7               result.append(numbers[i])
8   abs_reversed([0,20,-12,37,-10])
9       """takes a list of numbers and returns its' absolute values in reverse"""
10              result.append(numbers[i] * -1)
11      return result
12          else:
```

7. For loops can be converted to while loops. True or False?

    True False

8. A tuple can contain another tuple. However, lists cannot contain another list. True or False?

    True False

Second Evaluation Study Test A

1. Code Reading

The following code `print_nice_weather(weather)` accepts a tuple and should **PRINT** `"Nice weather"` when `rain_mm` is less than 2.0 and `temperature` is at least 12. Otherwise it should print `"Weather is not so nice"`.

```
def print_nice_weather(weather):
    '''Checks if today's weather is nice'''
    rain_mm, temperature = weather
    if rain_mm < 2.0 and temperature >= 12:
        print("Nice weather")
    else:
        print("Weather is not so nice")
```

QUESTION: Is the code above correct? Circle YES or NO

Test cases:

Test case	Expected Result/Output
`print_nice_weather((5.6, -5))`	`Weather is not so nice`
`print_nice_weather((2.6, 15))`	`Weather is not so nice`
`print_nice_weather((0.5, 12))`	`Nice weather`
`print_nice_weather((0, 14))`	`Nice weather`

2. Identify 1 wrong line

The following code `print_triangle_hash(size)` is supposed to **PRINT** a "triangle of hashes". For example: if `size` is 3, it would print:

```
###
##
#
```

It should not print anything if size given is less than 1.

Circle the wrong line (that needs to be fixed) in the following code below:

```
def print_triangle_hash(size):
    '''Prints a triangle of hashtags'''
    while size >= 1:
        print('#' * size)
        size += 1
```

Test case	Expected Result/Output
`print_triangle_hash(-5)`	`No output displayed`
`print_triangle_hash(2)`	`##` `#`

3. Fix 1 wrong line

The following code supposed to **RETURN** the longest word from the given string. However, the code is not returning the longest word. The highlighted line number 4 is causing the trouble. What should replace this line to fix the code?

```
1  def longest_word(string_of_words):
2      '''Returns the longest word from the given string'''
3      len_longest = 0
4      for word in string_of_words:
5          length = len(word)
6          if length > len_longest:
7              len_longest = length
8              longest = word
9      return longest
```

Test case	Expected Result/Output
`string_of_words = "red blueberry poncho leprechaun lolly ant"` `print('Longest word: {}'.format (longest_word(string_of_words)))`	`Longest word: leprechaun`

What is the line of code that should replace `"for word in string_of_words: "`:

4. Actual code output

```
def to_upper(strings):
    '''Returns a new list with all strings from my_strings
    changed into uppercase'''
    strings_upper = []
    for word in strings:
        word_upper = word.upper()
        strings_upper.append(word_upper)
    return strings_upper

strings = ["word1", "majESty", "confiDent"]
print(to_upper(strings))
```

What is the output of the code above?

5. Expected output
The following code supposed to take a list of words and remove all the vowels in each word. It should **PRINT** a new list of words without the vowels. The code is correct except for the highlighted line 8.

```
1  def print_words_no_vowels_and_word_length(words):
2      '''Prints a new_list of words without vowels'''
3      vowels = ['a', 'e', 'i', 'o', 'u']
4      new_list = []
5      for word in words:
6          new_word = ""
7          for letter in word:
8              if letter != vowels:
9                  new_word += letter
10         new_list.append(new_word)
11     print(new_list)
```

Assuming the above code is fixed, what output are you expecting from the test case below? You don't have to fix the code. Just write the expected output of the test case below.

Test case	Expected Result/Output
`words = ["onomatopoeia", "precipitation", "gummy"]` `print_words_no_vowels_and_word_length(words)`	`#YOUR ANSWER HERE`

6. Code writing
Write `print_triangle_hash_reversed(size)` which **PRINTS** a triangle of hashtags but in a reversed manner unlike code in Q2. It should not print anything if size given is less than 1.

```
def print_triangle_hash_reversed(size):
    '''Prints a triangle of hashtags reversed'''
    #YOUR CODE GOES HERE
```

Test cases:

Test case	Expected Result/Output

`print_triangle_hash_reversed(3)`	`#`
	`##`
	`###`

Second Evaluation Study Test B

1. Code Reading:
The following code `is_good_day(mood_and_temperature)` accepts a tuple and should **RETURN** a string `"Good day"` when `mood` is `"happy"` or `temperature` is at least 12. Otherwise, it should return `"Not so good day".`

```
def is_good_day(mood_and_temperature): '''Checks if
    today is a good day''' mood, temperature =
    mood_and_temperature if temperature > 12 or mood
    = happy:
        result = "Good day" else:
        result = "Not so good day" return
    result
```

QUESTION: Is the code above correct? Circle YES or NO

Test cases:

Test case	Expected Result/Output
`print(is_good_day(("sad", 11)))`	`Not so good day`
`print(is_good_day(("meh", 20)))`	`Good day`
`print(is_good_day(("happy", 0)))`	`Good day`
`print(is_good_day(("hmm", -5)))`	`Not so good day`

2. Identify 1 wrong line
The following code `insert_stars(string)` is supposed to take words from a string and insert stars (asterisks) in between each word. For example: " insert stars in middle" becomes "insert*stars*in*middle". The code is expected to **RETURN** this new string with stars.

Circle the wrong line (that needs to be fixed) in the following code below:

```
  def insert_stars(string):
     '''Inserts *s in between words and returns it''' words =
     string.split()
     result = '*'.join(string) return
     result
```

Test case	Expected Result/Output
`my_string = " i need some stars "` `print(insert_stars(my_string))`	`i*need*some*stars`

3. Fix 1 wrong line

The following code supposed to **PRINT** alternating `'*'` and `'#'` and a trunk `'|_|'` to print a tree as seen below:

```
*
##
***
####
*****
######
*******
|__|
```

However, the tree is not being printed correctly. The highlighted line number 6 is causing the trouble. What should replace this line to fix the code and print the tree as seen above?

```
1  def print_tree():
2      '''Prints a beautiful tree made of stars
3      and hash'''
4      count = 1
5      while count < 8:
6          if count // 2:
7              print('#' * count)
8          else:
9              print('*' * count)
10         count += 1
11             print("|__|")
12
13           print_tree()
```

What is the line of code that should replace `"if count // 2:"`:

4. Actual Code Output

```
  def secret_message(secret): '''Prints the
     secret message''' message = ""
     for letter in secret:
         if letter.isalpha(): message
             += letter
     print(message)

  secret   =   "       R#4e3v!&$e!$a#l!$5M09e   "
  secret_message(secret)
```

What is the output of the code above?

5. Expected output

The following code supposed to take a list of names and **PRINT** and replace all the letter `'d'` with the letter `'a'`. However, for names without `'d'` the name is printed with the word "good" i.e. ella is printed as goodella. The code is correct except for the highlighted line 4.

```
1  def print_names_d_to_a(names):
2     '''takes a list of names and replaces all d's into a's'''
3     for name in names:
4         if name[i] != "d": #if “d” not in name:
5             print("good" + name)
6         else:
7             print(name.replace('d','a'))
```

Assuming the above code is fixed, what output are you expecting from the test case below? You don't have to fix the code. Just write the expected output of the test case below.

Test case	Expected Result/Output

`names = ["edward", "michael", "deandre", "diana"]`
 	
`print_names_d_to_a(names)`
	`#YOUR ANSWER HERE`

6. Code writing

Write `print_square_stars(size)` which **PRINTS** a square of stars (asterisks) from the size given. It should not print anything if `size` given is less than 2.

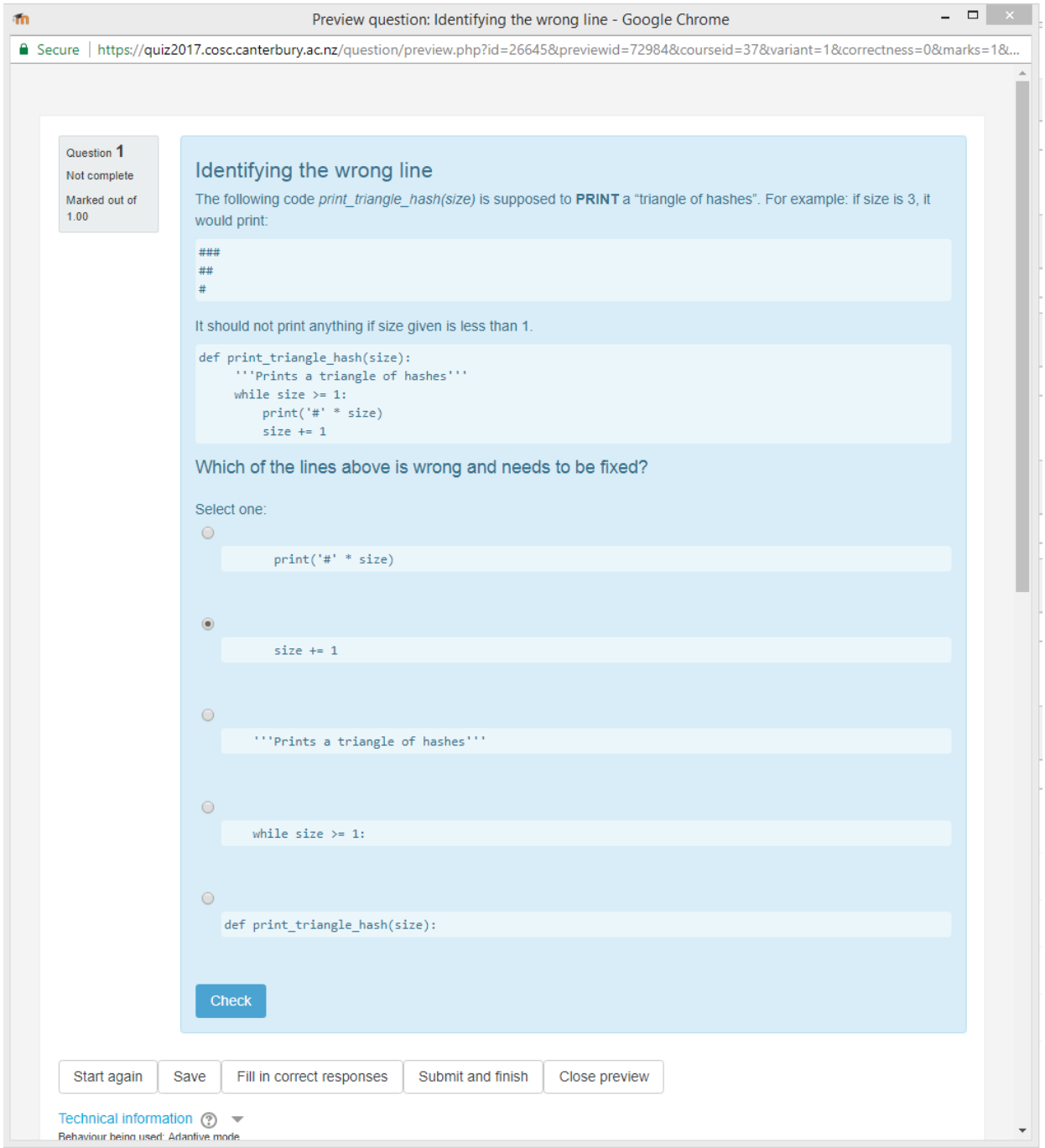
```
def print_square_stars(size):
    '''Prints a square of stars'''
    #YOUR CODE GOES HERE
```

Test cases:

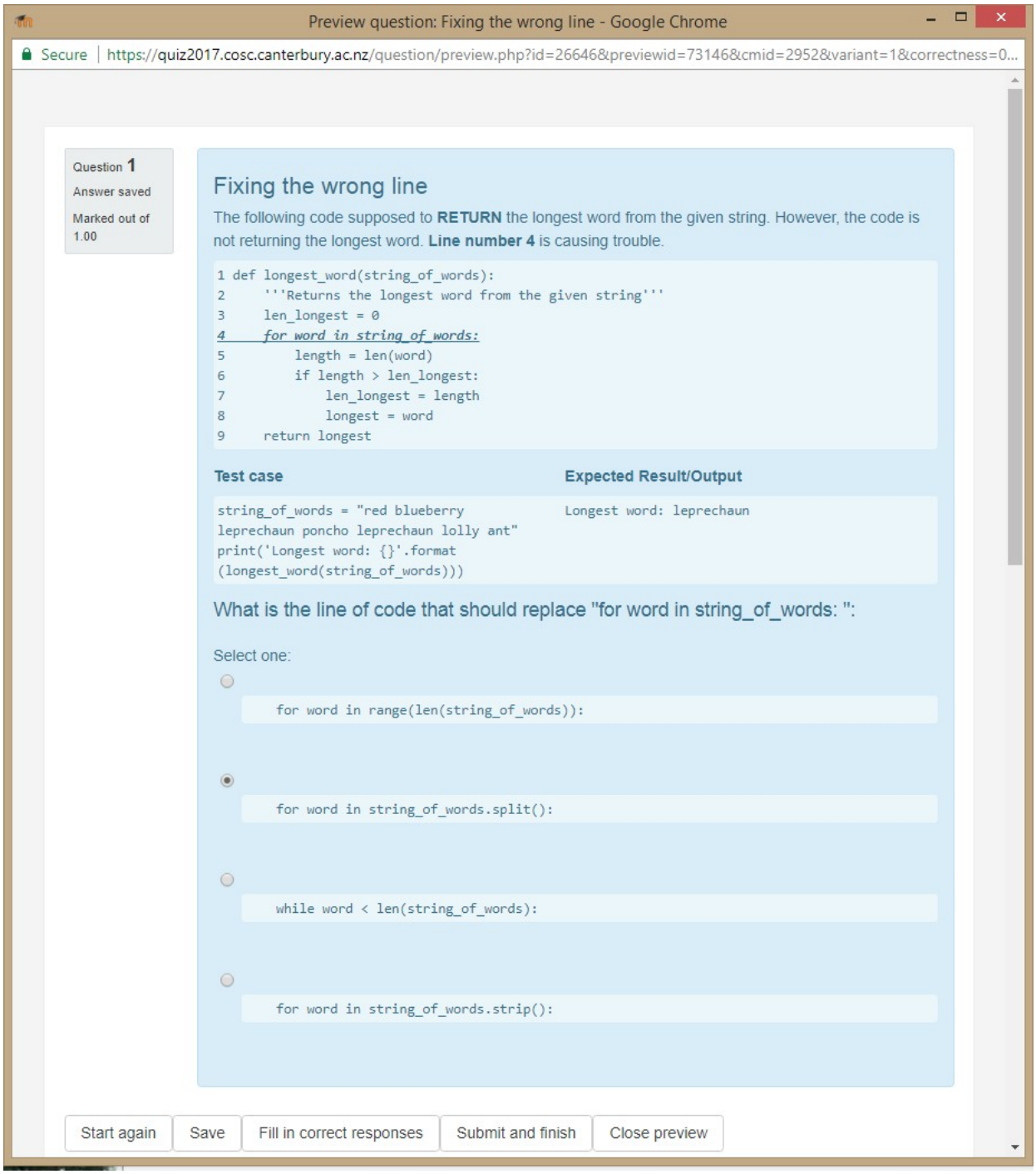
Test case	Expected Result/Output
`print_square_stars(2)`	`**` `**`
`print_square_stars(3)`	`***` `***` `***`

Third Evaluation Study Test A

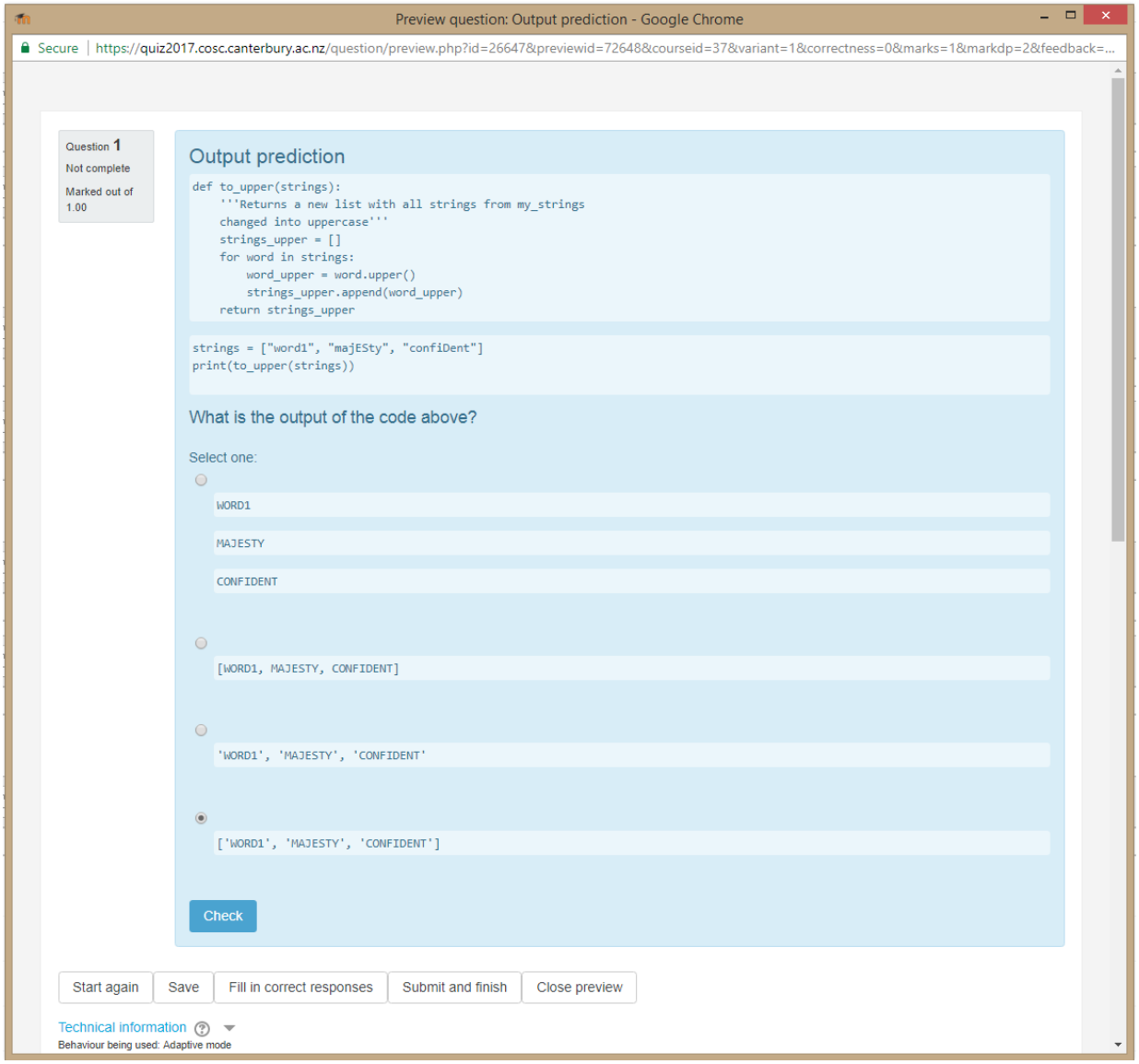
1. Identifying Errors



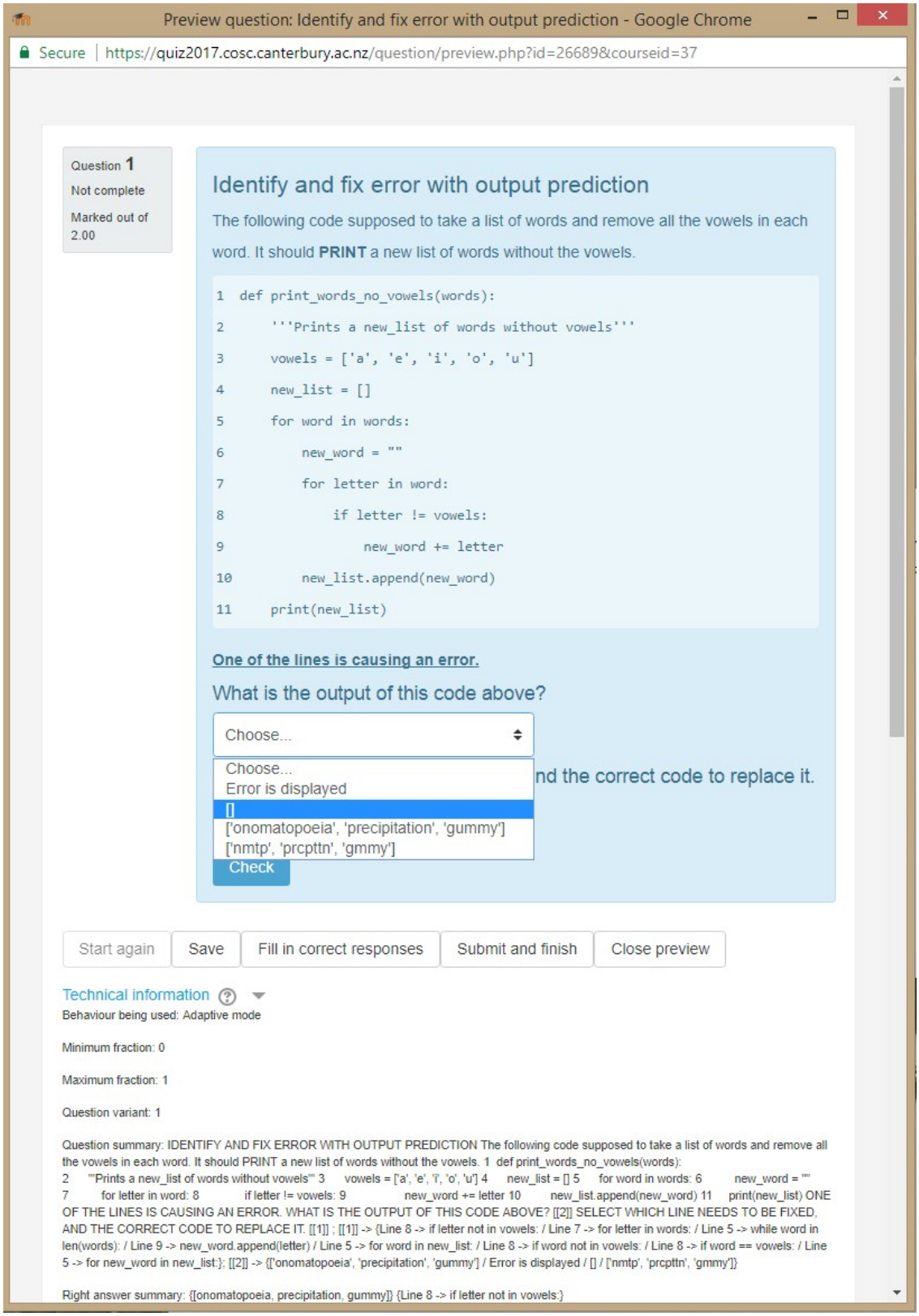
2. Fixing Erroneous LOC

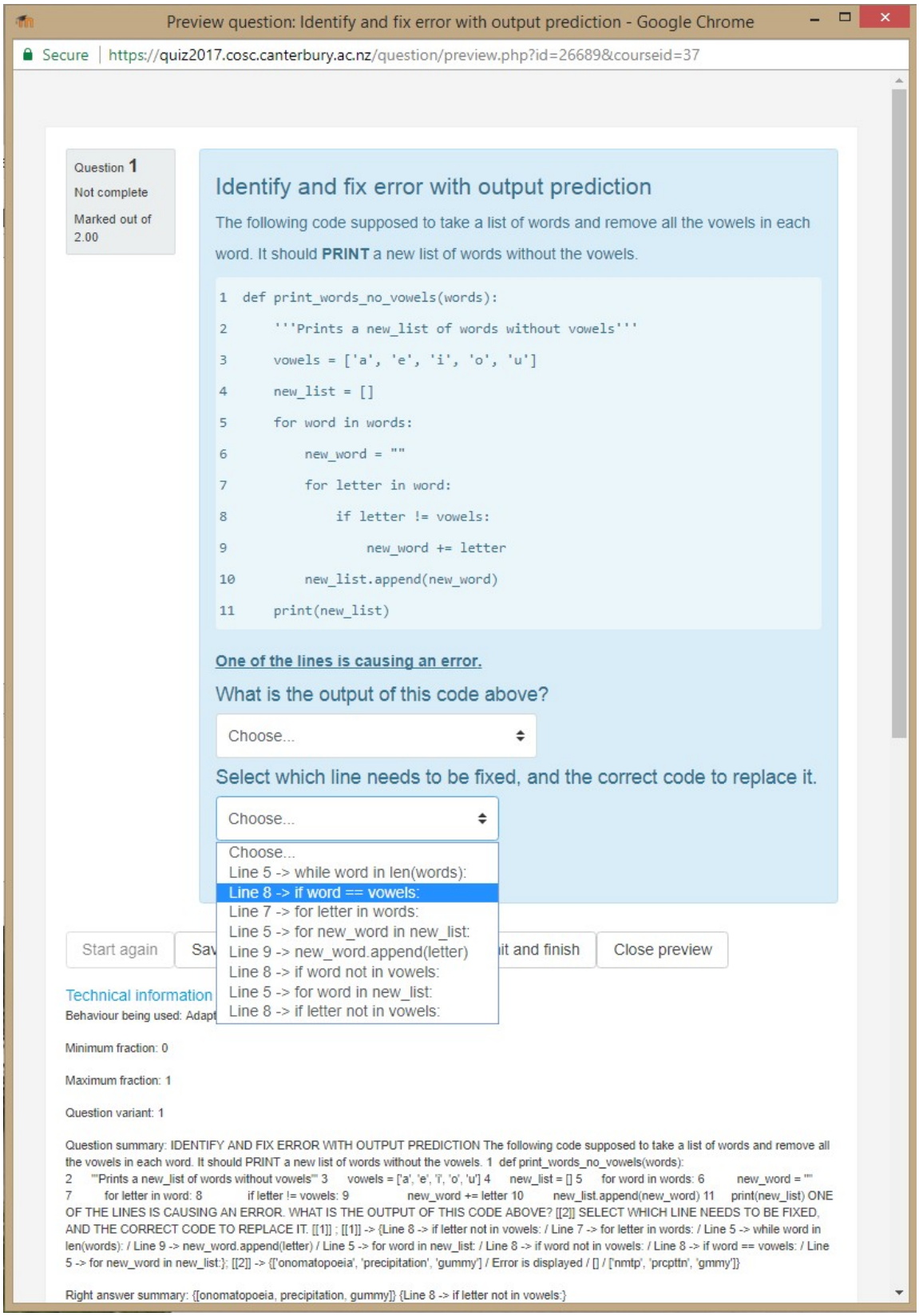


3. Output Prediction

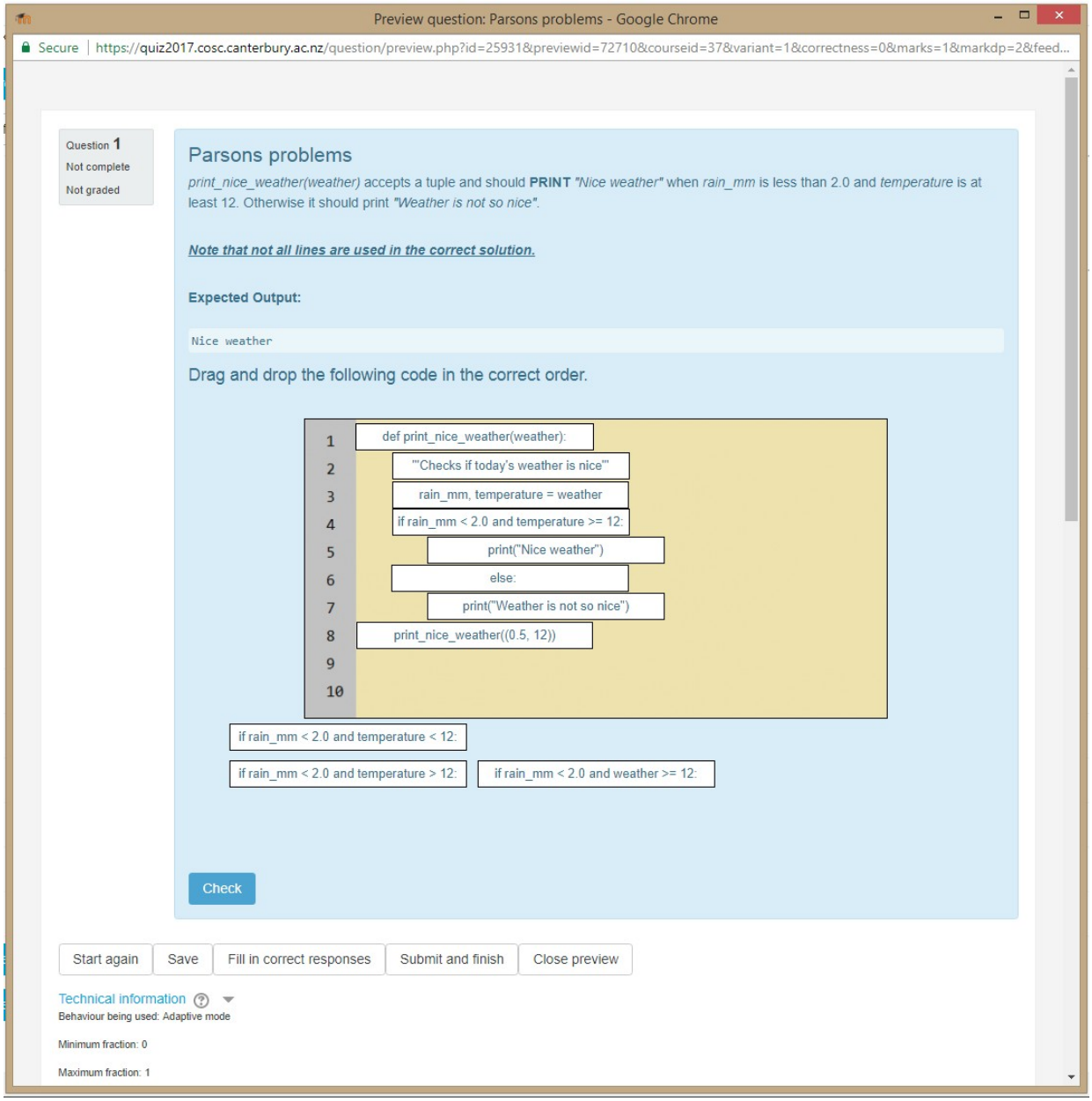


4. Identifying and Fixing Erroneous LOC with Output Prediction

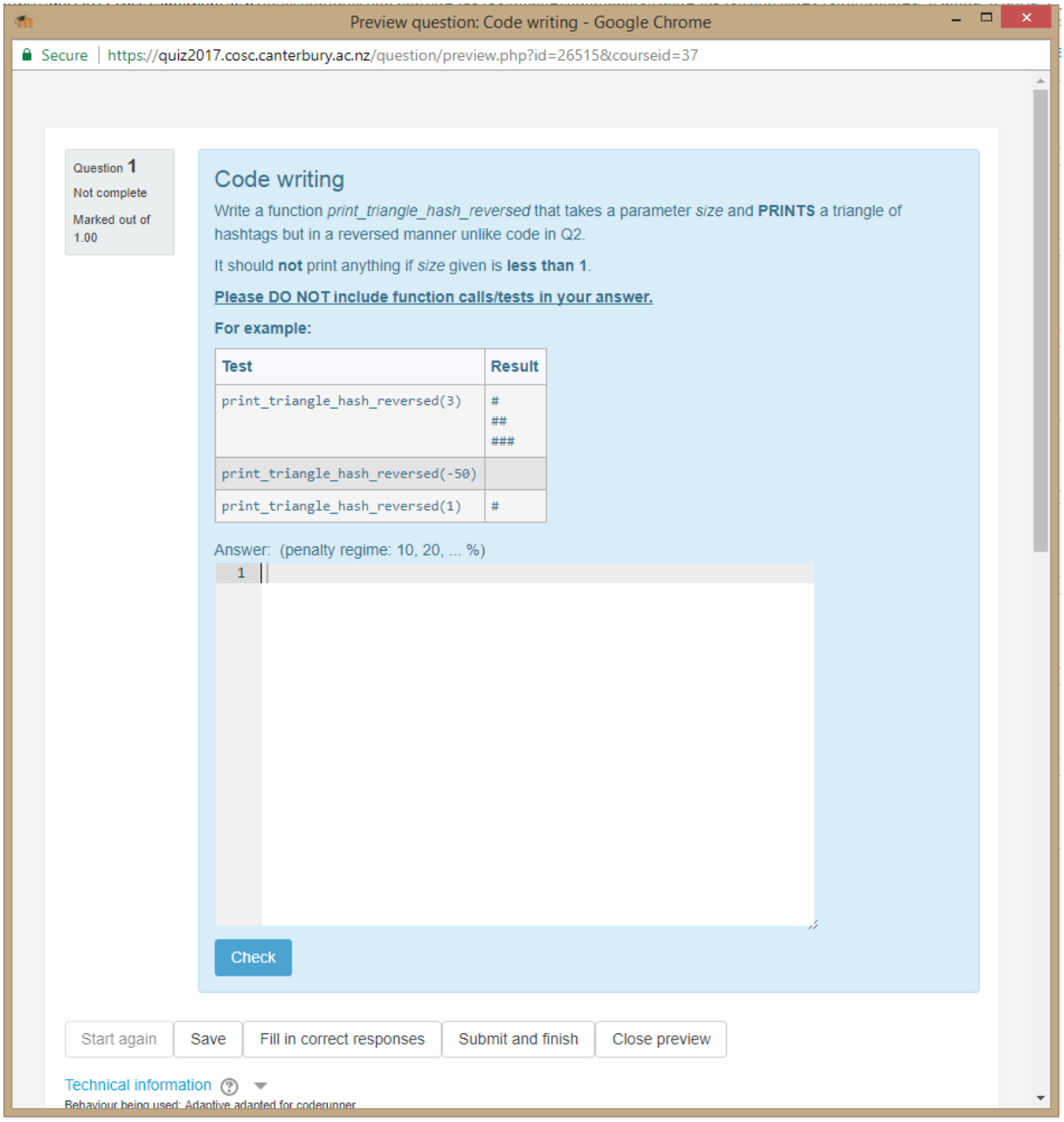



Preview question: Identify and fix error with output prediction - Google Chrome
Secure | https://quiz2017.cosc.canterbury.ac.nz/question/preview.php?id=26689&courseid=37
Question 1
Not complete
Marked out of 2.00
Identify and fix error with output prediction
The following code supposed to take a list of words and remove all the vowels in each word. It should PRINT a new list of words without the vowels.
1 def print_words_no_vowels(words):
2 '''Prints a new_list of words without vowels'''
3 vowels = ['a', 'e', 'i', 'o', 'u']
4 new_list = []
5 for word in words:
6 new_word = ""
7 for letter in word:
8 if letter != vowels:
9 new_word += letter
10 new_list.append(new_word)
11 print(new_list)
One of the lines is causing an error.
What is the output of this code above?
Choose...
Select which line needs to be fixed, and the correct code to replace it.
Choose...
Choose...
Line 5 -> while word in len(words):
Line 8 -> if word == vowels:
Line 7 -> for letter in words:
Line 5 -> for new_word in new_list:
Line 9 -> new_word.append(letter)
Line 8 -> if word not in vowels:
Line 5 -> for word in new_list:
Line 8 -> if letter not in vowels:
Start again
Close preview
Technical information
Behaviour being used: Adapt
Minimum fraction: 0
Maximum fraction: 1
Question variant: 1
Question summary: IDENTIFY AND FIX ERROR WITH OUTPUT PREDICTION The following code supposed to take a list of words and remove all the vowels in each word. It should PRINT a new list of words without the vowels. 1 def print_words_no_vowels(words): 2 '''Prints a new_list of words without vowels''' 3 vowels = ['a', 'e', 'i', 'o', 'u'] 4 new_list = [] 5 for word in words: 6 new_word = "" 7 for letter in word: 8 if letter != vowels: 9 new_word += letter 10 new_list.append(new_word) 11 print(new_list) ONE OF THE LINES IS CAUSING AN ERROR. WHAT IS THE OUTPUT OF THIS CODE ABOVE? [[2]] SELECT WHICH LINE NEEDS TO BE FIXED, AND THE CORRECT CODE TO REPLACE IT. [[1]] ; [[1]] -> {Line 8 -> if letter not in vowels: / Line 7 -> for letter in words: / Line 5 -> while word in len(words): / Line 9 -> new_word.append(letter) / Line 5 -> for word in new_list: / Line 8 -> if word not in vowels: / Line 8 -> if word == vowels: / Line 5 -> for new_word in new_list:}; [[2]] -> {{'onomatopoeia', 'precipitation', 'gummy'] / Error is displayed / [] / ['nmtp', 'prcpttn', 'gmmy']}
Right answer summary: {[onomatopoeia, precipitation, gummy]} {Line 8 -> if letter not in vowels:}

5. Parsons Problem with distractors (correct answer shown)

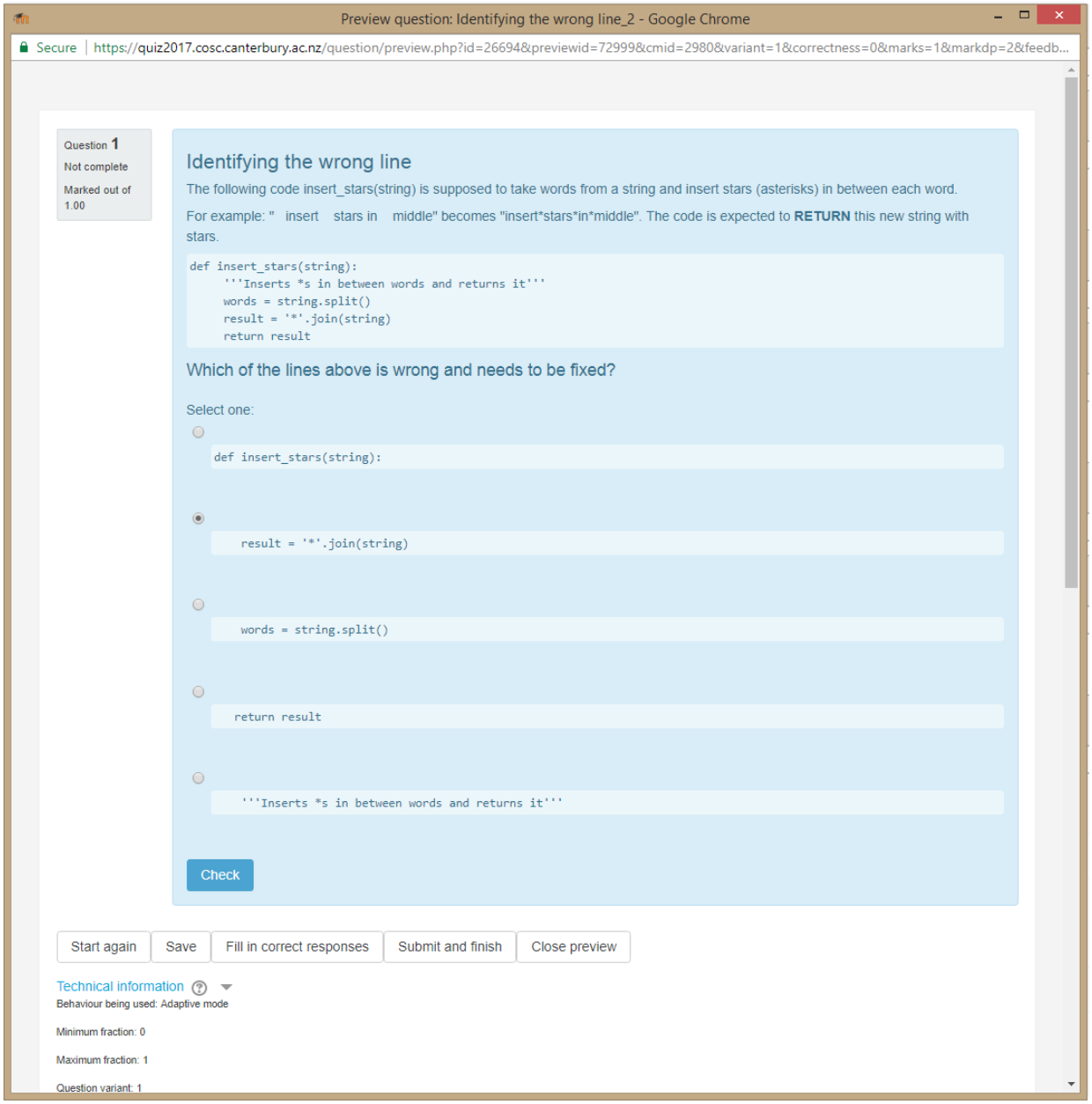


6. Code Writing

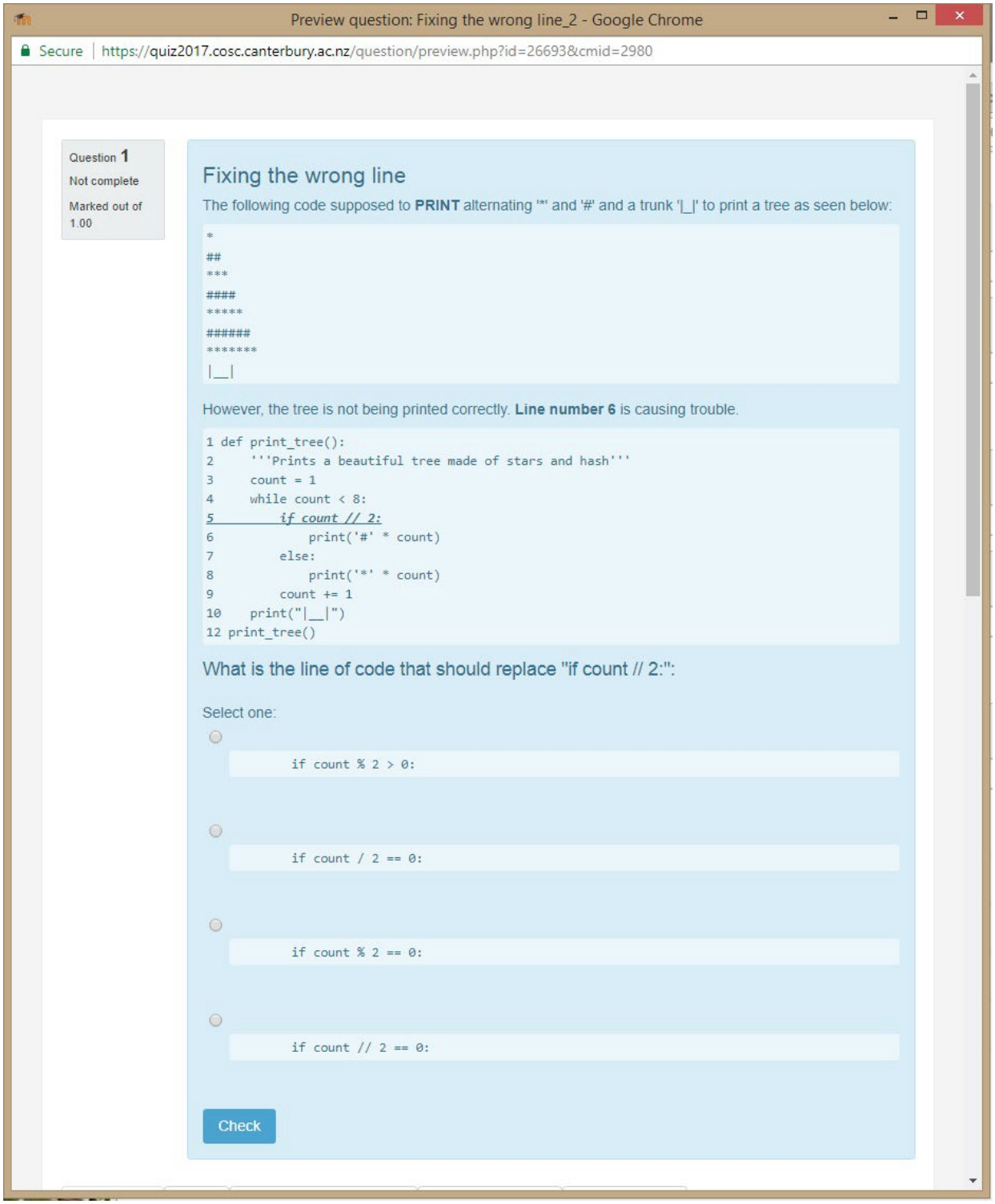


Third Evaluation Study Test B

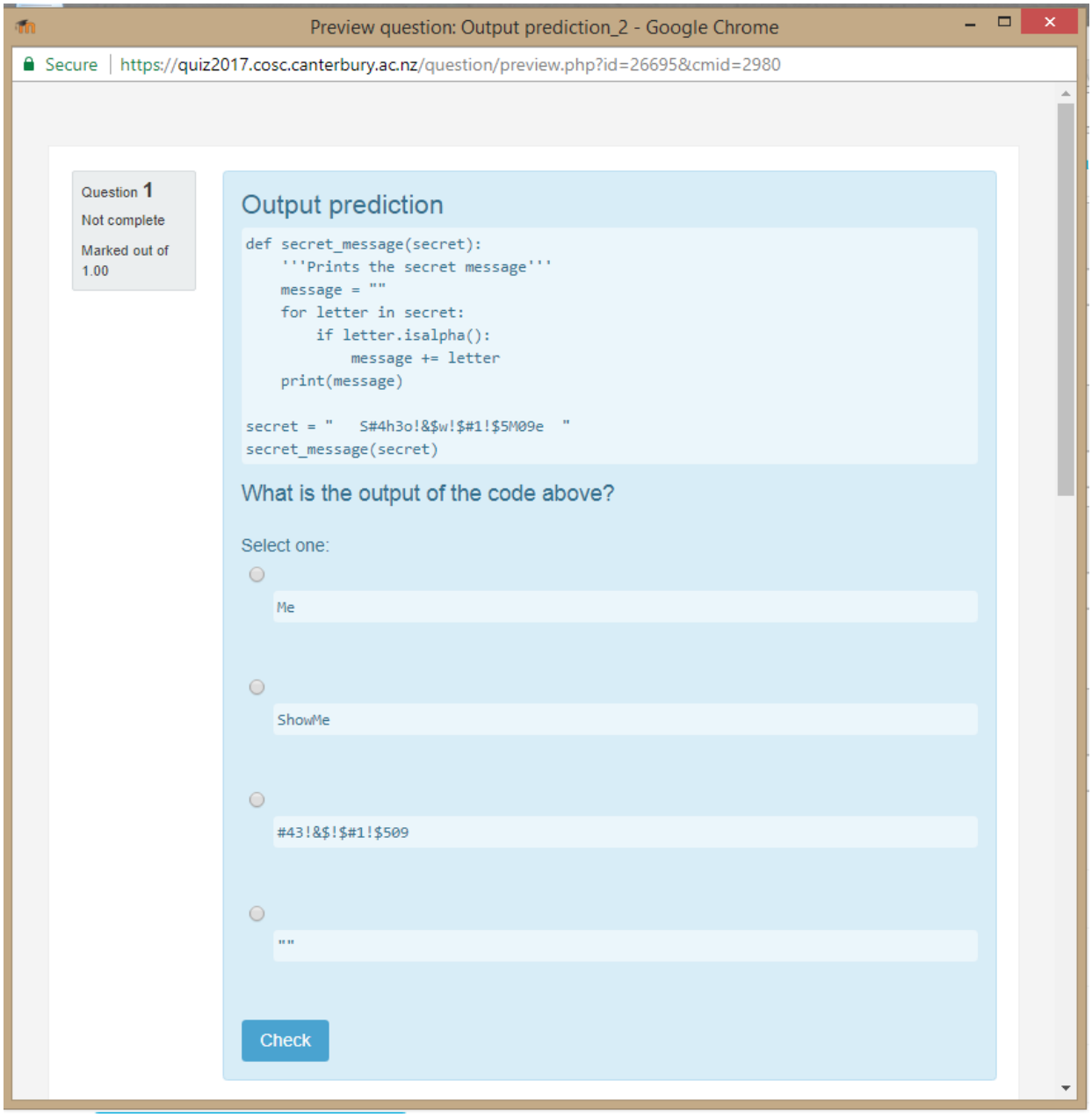
1. Identifying Errors



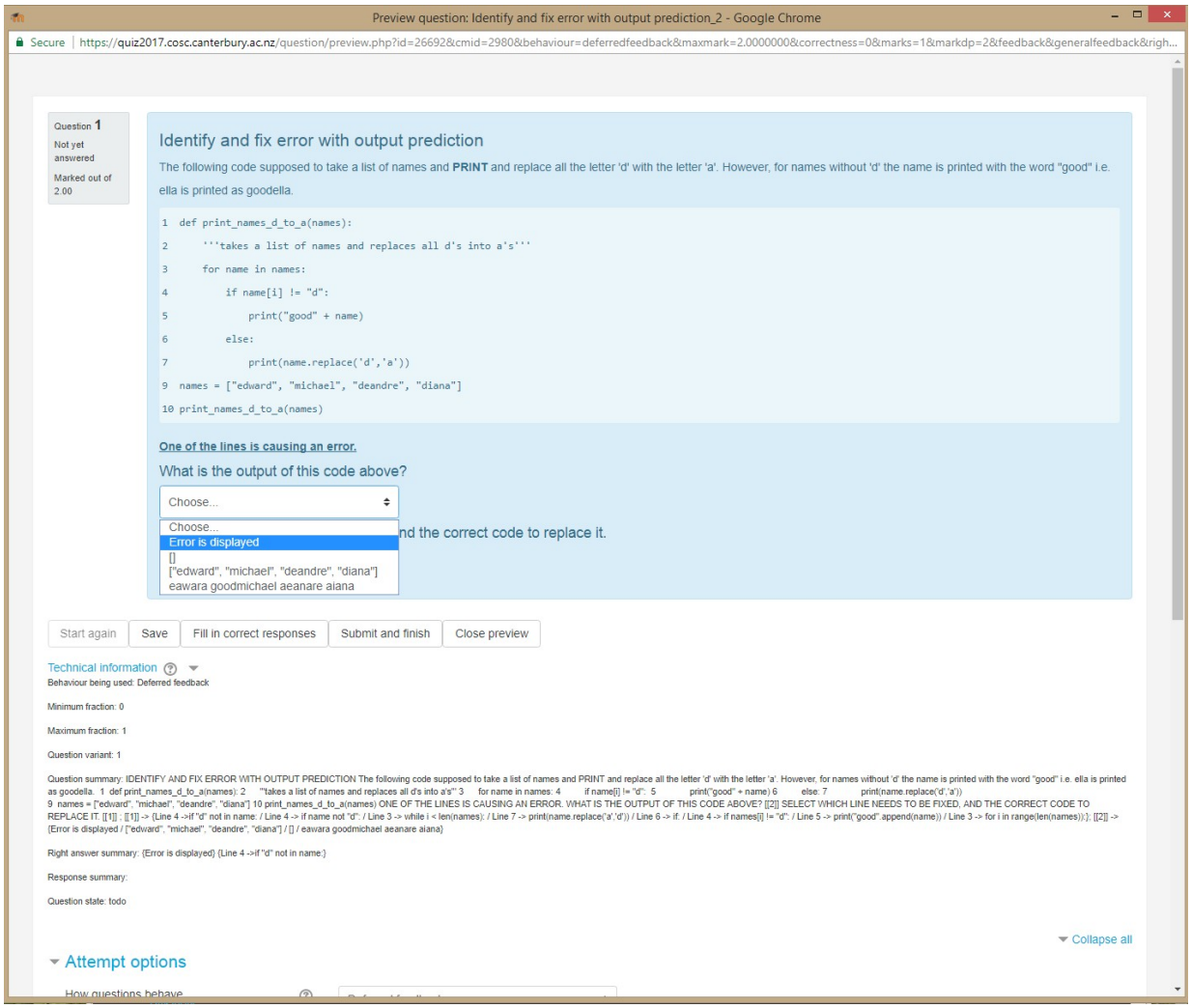
2. Fixing Erroneous LOC

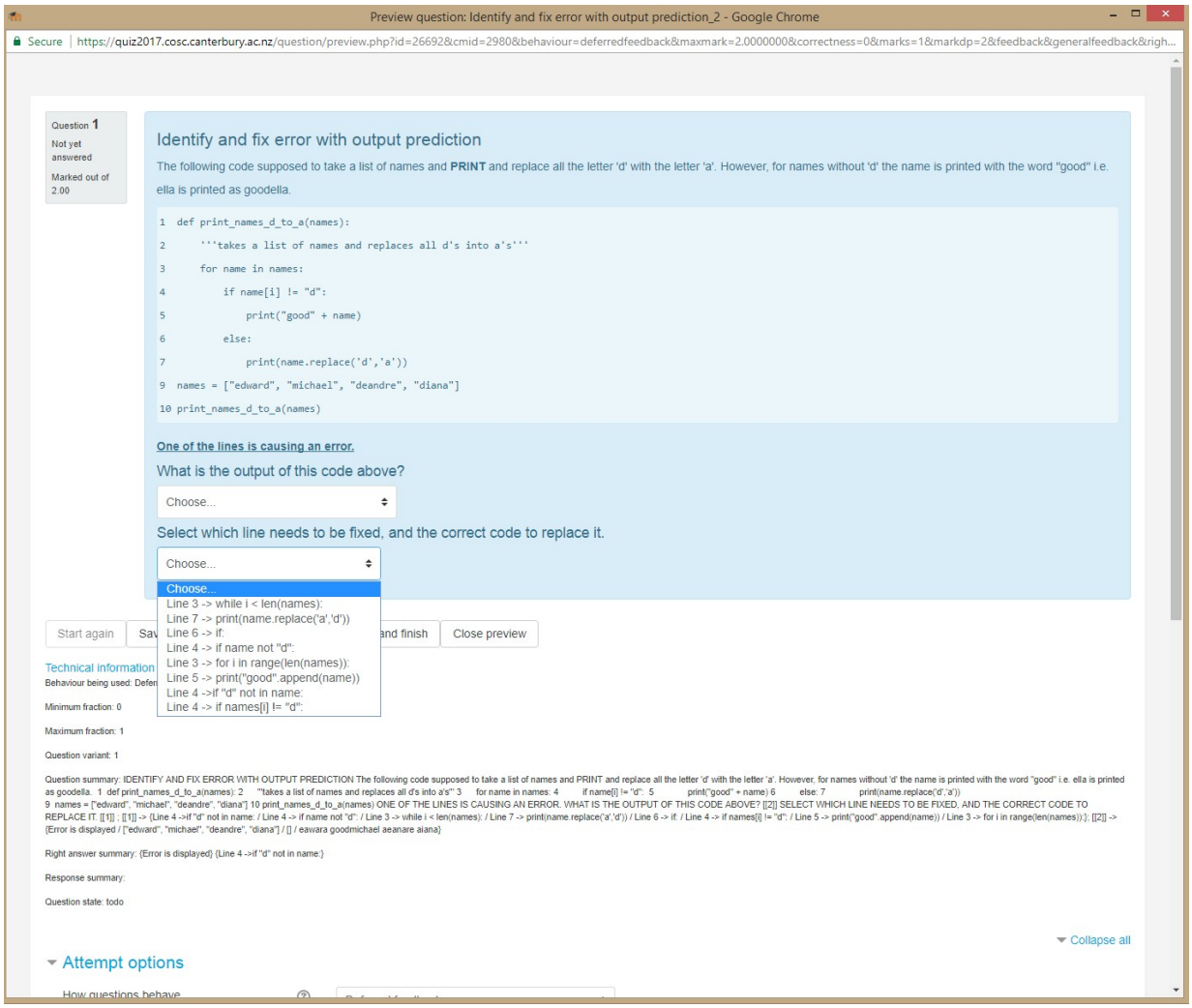


3. Output Prediction

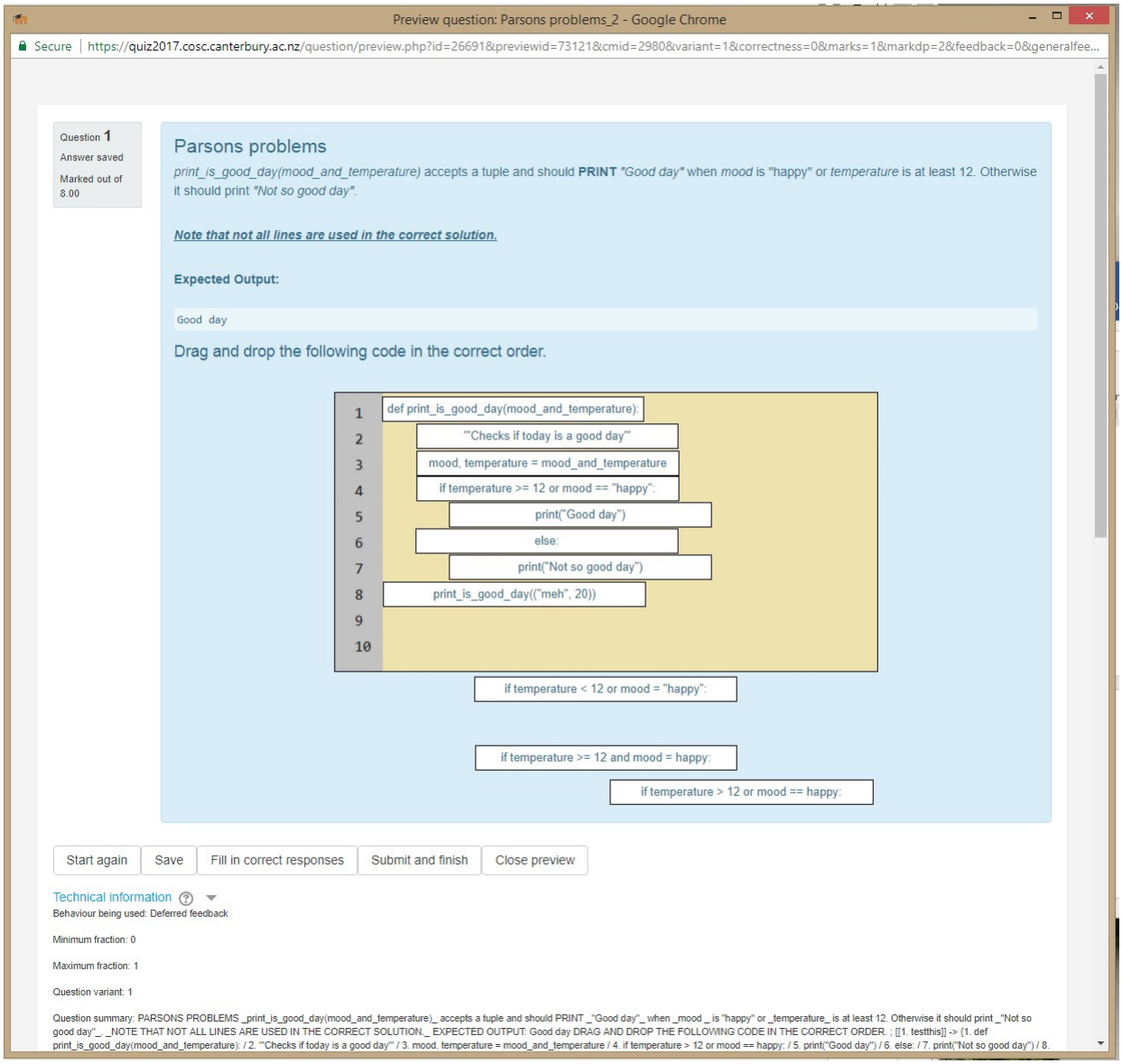


4. Identifying and Fixing Erroneous LOC with Output Prediction

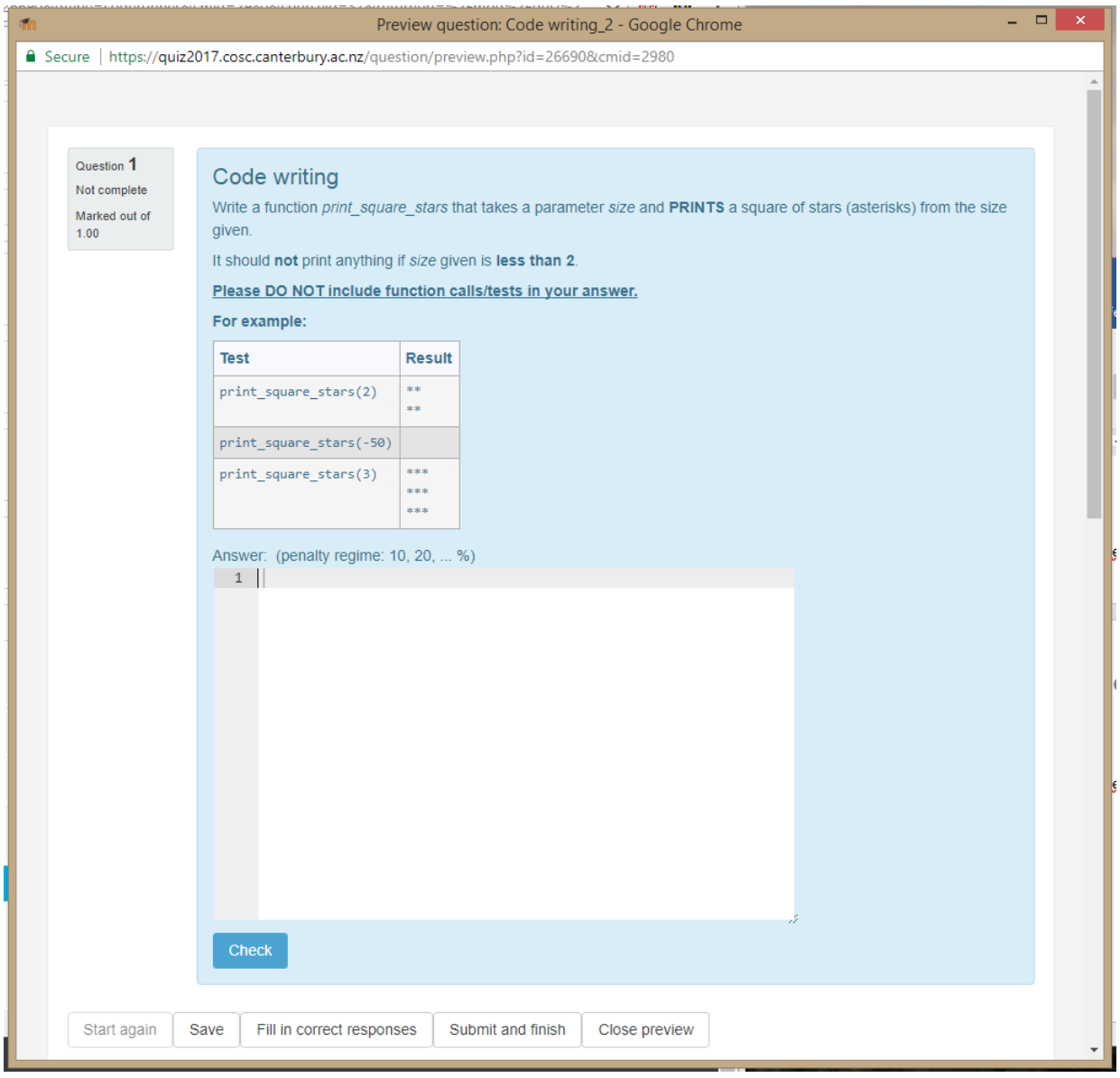



Preview question: Identify and fix error with output prediction_2 - Google Chrome
Secure | https://quiz2017.cosc.canterbury.ac.nz/question/preview.php?id=26692&cmid=2980&behaviour=deferredfeedback&maxmark=2.0000000&correctness=0&marks=1&markdp=2&feedback&generalfeedback&righ...
Question 1
Not yet answered
Marked out of 2.00
Identify and fix error with output prediction
The following code supposed to take a list of names and PRINT and replace all the letter 'd' with the letter 'a'. However, for names without 'd' the name is printed with the word "good" i.e. ella is printed as goodella.
1 def print_names_d_to_a(names):
2 '''takes a list of names and replaces all d's into a's'''
3 for name in names:
4 if name[i] != "d":
5 print("good" + name)
6 else:
7 print(name.replace('d','a'))
9 names = ["edward", "michael", "deandre", "diana"]
10 print_names_d_to_a(names)
One of the lines is causing an error.
What is the output of this code above?
Choose...
Select which line needs to be fixed, and the correct code to replace it.
Choose...
Choose...
Line 3 -> while i < len(names):
Line 7 -> print(name.replace('a','d'))
Line 6 -> if:
Line 4 -> if name not "d":
Line 3 -> for i in range(len(names)):
Line 5 -> print("good".append(name))
Line 4 ->if "d" not in name:
Line 4 -> if names[i] != "d":
Start again
Close preview
Technical information
Minimum fraction: 0
Maximum fraction: 1
Question variant: 1
Question summary: IDENTIFY AND FIX ERROR WITH OUTPUT PREDICTION The following code supposed to take a list of names and PRINT and replace all the letter 'd' with the letter 'a'. However, for names without 'd' the name is printed with the word "good" i.e. ella is printed as goodella. 1 def print_names_d_to_a(names): 2 '''takes a list of names and replaces all d's into a's''' 3 for name in names: 4 if name[i] != "d": 5 print("good" + name) 6 else: 7 print(name.replace('d','a')) 9 names = ["edward", "michael", "deandre", "diana"] 10 print_names_d_to_a(names) ONE OF THE LINES IS CAUSING AN ERROR. WHAT IS THE OUTPUT OF THIS CODE ABOVE? [[2]] SELECT WHICH LINE NEEDS TO BE FIXED, AND THE CORRECT CODE TO REPLACE IT. [[1]] : [[1]] -> {Line 4 ->if "d" not in name: / Line 4 -> if name not "d": / Line 3 -> while i < len(names): / Line 7 -> print(name.replace('a','d')) / Line 6 -> if: / Line 4 -> if names[i] != "d": / Line 5 -> print("good".append(name)) / Line 3 -> for i in range(len(names)):}; [[2]] -> {Error is displayed / ["edward", "michael", "deandre", "diana"] / [] / eawara goodmichael aeanare aiana}
Right answer summary: {Error is displayed} {Line 4 ->if "d" not in name:}
Response summary:
Question state: todo
Collapse all
Attempt options

5. Parsons Problem with distractors (correct answer shown)



6. Code Writing



Appendix B. Questionnaires for the Pilot Study and Second Evaluation Study

Pilot Study

Questionnaire

Background and Experience:

1. Please state the degree you are studying towards and your major: i.e. BSc Computer Science

_____

2. Please indicate your experience in programming with Python

Not so experienced						Highly Experienced
1	2	3	4	5	6	7

3. Have you used any educational systems before? If so, please write down some of them.

_____

4. Have you used any Python tutors before? If so, please write down some of them.

_____

5. Which types of mobile devices do you own?
   - ☐ Smartphone
   - ☐ Tablet
   - ☐ Wearables (i.e. fitbit, apple watch etc.)
   - ☐ Other: _____
   - ☐ Do not own any (If yes, skip to Question 7)

6. What does your mobile devices run on?
   - ☐ Android
   - ☐ iOS/Apple
   - ☐ Blackberry
   - ☐ Windows
   - ☐ Other: _____

7. How much time on average do you spend every day in using a smartphone device?
   - ☐ < 1 hour
   - ☐ 1 – 3 hours
   - ☐ More than 3 – 6 hours
   - ☐ > 8 hours
   - ☐ Do not own/do not have access to one

Questions about Python Tutor:

1. Was the tutor's interface intuitive and easy to use?

Strongly Disagree					**Strongly Agree**	
1	**2**	**3**	**4**	**5**	**6**	**7**

2. Was the tutor fun to interact with?

Strongly Disagree					**Strongly Agree**	
1	**2**	**3**	**4**	**5**	**6**	**7**

3. Would you say you have learned some new things and/or enhanced your skills by interacting with the tutor?

Strongly Disagree					**Strongly Agree**	
1	**2**	**3**	**4**	**5**	**6**	**7**

4. Do you think it is beneficial that this tutor is developed on a mobile platform?

Strongly Disagree					**Strongly Agree**	
1	**2**	**3**	**4**	**5**	**6**	**7**

5. Were the problem statements in the tutor clear enough to understand what needed to be achieved?

Strongly Disagree					**Strongly Agree**	
1	**2**	**3**	**4**	**5**	**6**	**7**

6. Please rate the average difficulty of the problems loaded in the tutor:

Too Easy					**Too Difficult**	
1	**2**	**3**	**4**	**5**	**6**	**7**

7. Do you think there is enough feedback given when attempting a problem?

Too Little					**Too Much**	
1	**2**	**3**	**4**	**5**	**6**	**7**

8. Which of these skills have you used to attempt the problems in this tutor? (Please select all that apply.)
   - ☐ Code reading skills
   - ☐ Code syntax and structure skills
   - ☐ Logical and semantic reasoning
   - ☐ Other: _____
   - ☐ I don't know

9. If given the opportunity, will you be using this tutor again to practice learning Python? Why or why not?

_____

_____

10. Do you have any comments/suggestions about the tutor?

_____

_____

11. Do you have any comments/suggestions about the contents of the tutor i.e. topics and problems included?

_____

_____

Second Evaluation Study: Given Before Using PyKinetic

Age:
Gender:
Is English your first language?

1. Have you studied any other programming languages before? If yes, please state them.

_____

2. Please indicate your experience in programming with Python

Not so Experienced									**Experienced**
1	2	3	4	5	6	7	8	9	10

3. Have you used any computer-based Python tutors before? If so, please list them.

_____

4. What does your mobile device run on?
   - ☐ Android
   - ☐ iOS/Apple
   - ☐ Blackberry
   - ☐ Windows
   - ☐ Other: _____

5. How much time on average do you spend every day in using a smartphone device?
   - ☐ < 1 hour
   - ☐ 1 – 3 hours
   - ☐ More than 3 – 6 hours
   - ☐ > 8 hours
   - ☐ Do not own/do not have access to one

Second Evaluation Study: Given After Using PyKinetic

Comments and Suggestions

1. If given the opportunity, will you be using this tutor again to practice learning Python? Why or why not?

   _____

   _____

2. Do you have any comments/suggestions about the tutor?

   _____

   _____

Appendix C. Information Sheets

Pilot Study

Information Sheet for Pilot Study

Department: Computer Science and Software Engineering
Email: geela.fabic@pg.canterbury.ac.nz

Investigating the Effectiveness of a Python Tutor on a Mobile Platform for Novice Programmers

Information Sheet for participants

I am Geela Fabic, a Masters student at the University of Canterbury from the Computer Science and Software Engineering department. I am developing a Python tutor for smartphone devices and investigating its effectiveness for helping novice programmers in learning Python.

I am the main researcher of this project. Planning, designing and conducting the evaluation of the study will be done by myself. I will also be overseeing all the sessions in the study. Data gathered in the study will also be managed, analysed and processed by myself.

If you agree to take part in the study, you will be asked to do the following tasks:

- Listen to a brief introduction to the study.
- Follow the "think-aloud" protocol: while solving problems in the tutor, you will be asked to say what you are thinking about.
- Attempt at least one exercise from each topic. You are free to select the order on which you attempt topics and problems. You are free to abandon any problem as you wish and are not required to finish all problems attempted. You are also free to attempt more problems as required just as long as it is within the time frame.
- Fill in questionnaire. The questionnaire will have general questions about your background and experience with Python and smartphone devices, as well as specific questions about the system and your experience in using it.

If you are confused about how to use the system, you may ask the instructor for further clarifications. The goal of the study is not to assess your knowledge, but to evaluate the system itself. Your feedback is important for the future development of the system.

You may receive a copy of the project results by contacting the researcher at the conclusion of the project. Participation is voluntary and you have the right to withdraw at any stage without penalty. If you choose withdraw, I will remove all information relating to you.

The results of the project may be published, but you may be assured of the complete confidentiality of data gathered in this investigation: your identity will not be made public. To ensure confidentiality, data gathered in this session will be stored in password protected files and a password protected computer and also in locked storage facilities in the University of Canterbury. Only the researchers working on the project will have access to the data. Utmost care will be taken to maintain confidentiality of the participants and the results of this study. After 5 years following the Masters research, all data will be destroyed.

A thesis is a public document and will be available through the UC Library. The project is being carried out as a part of a Masters research project at the University of Canterbury in the Department of Computer Science and Software Engineering by Geela Fabic under the supervision of Dr. Antonija Mitrovic, who can be contacted at tanja.mitrovic@canterbury.ac.nz and Dr. Kourosh Neshatian who can be contacted at kourosh.neshatian@canterbury.ac.nz. They will be pleased to discuss any concerns you may have about participation in the project.

This project has been reviewed and approved by the University of Canterbury Human Ethics Committee, and participants should address any complaints to The Chair, Human Ethics Committee, University of Canterbury, Private Bag 4800, Christchurch (human-ethics@canterbury.ac.nz).
If you agree to participate in the study, you are asked to complete the consent form and return it to myself, Geela Fabic before taking part in the study.

First Evaluation Study

1. University of Canterbury

Information Sheet

Department: Computer Science and Software Engineering
Email: geela.fabic@pg.canterbury.ac.nz

Developing and Evaluating Activities for Increasing Engagement and Maximising Learning in a Mobile Programming Tutor

Information Sheet for participants

I am Geela Fabic, a PhD student at the University of Canterbury from the Computer Science and Software Engineering department. I am developing a Python tutor - PyKinetic for smartphone devices and evaluating activities that increase engagement and maximise learning helping novice programmers in learning Python.

I am the main researcher of this project. Planning, designing and conducting the evaluation of the study will be done by myself. I will also be overseeing all the sessions in the study. Data gathered in the study will also be managed, analysed and processed by myself.

If you agree to take part in the study, you will be asked to do the following tasks:

- Listen to a brief introduction to the study.
- Attempt a pre-test on paper. We will collect pre-tests when the allocated time for the pre-test is over.
- You will be presented with an instruction sheet on how to download and start using the application. Use your own Android smartphone to download and install the application from a given link. If you do not have an Android smartphone with you, you can borrow one of ours which would already have the application in it. Note: If you are using your own smartphone, you are kindly asked to turn off notifications from other applications, especially from social media applications that may disrupt your interactions with PyKinetic. Moreover, if you are familiar with installing Android applications you will know that some applications ask for permissions. PyKinetic only has **two permissions**: have full network access for connecting to WiFi and view network connections for checking if you are connected.
- Important: Before starting to use the application, ensure that you have internet quota or money left in your account and connect to **UCwireless Wifi**.
- Once you have started using the application, just follow the instructions and answer the problems within the application. You are not allowed to go back or skip a problem. Just try to attempt all problems until the time is up.

- Once allocated time for using the application is over, we will hand you the post-test on paper for you to attempt. Post-tests will be collected until the allocated time is over.
- For participants who used their own devices, we will also ask you to kindly delete and uninstall the application at this point.

If you are confused about how to use the system, you may ask the instructor for further clarifications. Your feedback is important for the future development of the system. It will be ideal for you to finish all questions in the pre-test and post-test, but do not feel bad if you did not manage to do so.

You may receive a copy of the project results by contacting the researcher at the conclusion of the project. Participation is voluntary and you have the right to withdraw at any stage without penalty. If you choose withdraw, I will remove all information relating to you.

The results of the project may be published, but you may be assured of the complete confidentiality of data gathered in this investigation: your identity will not be made public. To ensure confidentiality, data gathered in this session will be stored in password protected files and a password protected computer and also in locked storage facilities in the University of Canterbury. Only the researchers working on the project will have access to the data. Utmost care will be taken to maintain confidentiality of the participants and the results of this study. After 10 years following the PhD research, all data will be destroyed.

A thesis is a public document and will be available through the UC Library. The project is being carried out as a part of a PhD research project at the University of Canterbury in the Department of Computer Science and Software Engineering by Geela Fabic under the supervision of Dr. Antonija Mitrovic, who can be contacted at tanja.mitrovic@canterbury.ac.nz and Dr. Kourosh Neshatian who can be contacted at kourosh.neshatian@canterbury.ac.nz. They will be pleased to discuss any concerns you may have about participation in the project.

This project has been reviewed and approved by the University of Canterbury Human Ethics Committee, and participants should address any complaints to The Chair, Human Ethics Committee, University of Canterbury, Private Bag 4800, Christchurch (human-ethics@canterbury.ac.nz).
If you agree to participate in the study, you are asked to complete the consent form and return it to myself, Geela Fabic before taking part in the study.

2. Ateneo de Manila University

Information Sheet

Department: Computer Science and Software Engineering
Email: geela.fabic@pg.canterbury.ac.nz

Developing and Evaluating Activities for Increasing Engagement and Maximising Learning in a Mobile Programming Tutor

Information Sheet for participants

I am Geela Fabic, a PhD student at the University of Canterbury from the Computer Science and Software Engineering department. I am developing a Python tutor - PyKinetic for smartphone devices and evaluating activities that increase engagement and maximise learning helping novice programmers in learning Python. Based on our initial results, we have found evidence which shows that using PyKinetic is beneficial for novice programmers learning Python. Therefore, your participation in this study may help you enhance your Python programming skills. There is no compensation for this study.

I am the main researcher of this project. Conducting the evaluation of the study will be done by your instructors. Data gathered in the study will be managed, analysed and processed by myself.

The study will be for approximately two hours. Your participation is voluntary. There are no foreseeable risks to this study. However, if you wish, you can withdraw at any time by contacting your instructor.

If you agree to take part in the study, you will be asked to do the following tasks:

- Listen to a brief introduction to the study.
- Attempt a pre-test on paper. We will collect pre-tests when the allocated time for the pre-test is over.
- You will be presented with an instruction sheet on how to download and start using the application. Use your own Android smartphone to download and install the application from a given link. You are kindly asked to turn off notifications from other applications, especially from social media applications that may disrupt your interactions with PyKinetic. Moreover, if you are familiar with installing Android applications you will know that some applications ask for permissions. PyKinetic only has **two permissions**: have full network access for connecting to WiFi and view network connections for checking if you are connected.
- **Important: Before starting to use the application, ensure that you are connected to the Wifi.**
- Once you have started using the application, just follow the instructions and answer the problems within the application. You are not allowed to go back or skip a problem. Just try to attempt all problems until the time is up.

- Once allocated time for using the application is over, we will hand you the post-test on paper for you to attempt. Post-tests will be collected until the allocated time is over. We will also ask you to kindly delete and uninstall the application at this point.

If you are confused on how to use the system, you may ask the instructor for further clarifications. Your feedback is important for the future development of the system. It will be ideal for you to finish all questions in the pre-test and post-test, but do not feel bad if you did not manage to do so.

You may receive a copy of the project results by contacting the researcher at the conclusion of the project. Participation is voluntary and you have the right to withdraw at any stage without penalty. If you choose withdraw, I will remove all information relating to you.

The results of the project may be published, but you may be assured of the complete confidentiality of data gathered in this investigation: your identity will not be made public. To ensure confidentiality, data gathered in this session will be stored in password protected files and a password protected computer and also in locked storage facilities in the University of Canterbury. Only the researchers working on the project will have access to the data. Utmost care will be taken to maintain confidentiality of the participants and the results of this study. After 10 years following the PhD research, all data will be destroyed.

A thesis is a public document and will be available through the University of Canterbury Library. The project is being carried out as a part of a PhD research project at the University of Canterbury in the Department of Computer Science and Software Engineering by Geela Fabic under the supervision of Dr. Antonija Mitrovic, who can be contacted at tanja.mitrovic@canterbury.ac.nz and Dr. Kourosh Neshatian who can be contacted at kourosh.neshatian@canterbury.ac.nz. They will be pleased to discuss any concerns you may have about participation in the project.

This project has been reviewed and approved by the University of Canterbury Human Ethics Committee, and participants should address any complaints to The Chair, Human Ethics Committee, University of Canterbury, Private Bag 4800, Christchurch (human-ethics@canterbury.ac.nz).

If you agree to participate in the study, you are asked to complete the consent form and return it to your instructor before taking part in the study.

Second Evaluation Study

Information Sheet

Department: Computer Science and Software Engineering
Email: geela.fabic@pg.canterbury.ac.nz

Developing and Evaluating Activities for Increasing Engagement and Maximising Learning in a Mobile Programming Tutor

Information Sheet for participants

I am Geela Fabic, a PhD student at the University of Canterbury from the Computer Science and Software Engineering department. I am developing a Python tutor - PyKinetic for smartphone devices and evaluating activities that increase engagement and maximise learning helping novice programmers in learning Python.

I am the main researcher of this project. Planning, designing and conducting the evaluation of the study will be done by myself. I will also be overseeing all the sessions in the study. Data gathered in the study will also be managed, analysed and processed by myself.

If you agree to take part in the study, you will be asked to do the following tasks:

- Listen to a brief introduction to the study.
- Attempt a pre-test on paper. I will collect pre-tests when the allocated time for the pre-test is over.
- You will be presented with an instruction sheet on how to start and use the application. You will use one of our Android phones in this study.
- Important: Before starting to use the application, ensure that you are connected to the WiFi.
- Once you have started using the application, just follow the instructions and answer the problems within the application. You are not allowed to go back or skip a problem. Just attempt all problems until the time is up.
- Once allocated time for using the application is over, I will hand you the post-test on paper for you to attempt. Post-tests will be collected when the allocated time is over.

If you are confused about how to use the app, you may ask the instructor for further clarifications. Your feedback is important for the future development of PyKinetic. It will be ideal for you to finish all questions in the pre-test and post-test, but do not feel bad if you do not manage to do so.

You may receive a copy of the project results by contacting me at the conclusion of the project. Participation is voluntary and you have the right to withdraw at any stage without penalty. If you choose to withdraw, I will remove all information relating to you.

The results of the project may be published, but you may be assured of the complete confidentiality of data gathered in this investigation: your identity will not be made public. To ensure confidentiality, data gathered in this session will be stored in password-protected files and a password-protected computer and in locked storage facilities in the University of Canterbury. Only the researchers working on the project will have access to the data. Utmost care will be taken to maintain confidentiality of the participants and the results of this study. After 10 years following the PhD research, all data will be destroyed.

A thesis is a public document and will be available through the UC Library. The project is being carried out as a part of a PhD research project at the University of Canterbury in the Department of Computer Science and Software Engineering by Geela Fabic under the supervision of Dr. Antonija Mitrovic, who can be contacted at tanja.mitrovic@canterbury.ac.nz and Dr. Kourosh Neshatian who can be contacted at kourosh.neshatian@canterbury.ac.nz. They will be pleased to discuss any concerns you may have about participation in the project.

This project has been reviewed and approved by the University of Canterbury Human Ethics Committee, and participants should address any complaints to The Chair, Human Ethics Committee, University of Canterbury, Private Bag 4800, Christchurch (human-ethics@canterbury.ac.nz).
If you agree to participate in the study, you are asked to complete the consent form and return it to myself, Geela Fabic before taking part in the study.

Third Evaluation Study

Information Sheet

Department: Computer Science and Software Engineering Email: geela.fabic@pg.canterbury.ac.nz

Developing and Evaluating Activities for Increasing Engagement and Maximising Learning in a Mobile Programming Tutor

Information Sheet for participants

I am Geela Fabic, a PhD student at the University of Canterbury from the Computer Science and Software Engineering department. I am developing a Python tutor - PyKinetic for smartphone devices and evaluating activities that increase engagement and maximise learning helping novice programmers in learning Python.

I am the main researcher of this project. Planning, designing and conducting the evaluation of the study will be done by myself. I will also be overseeing all the sessions in the study. Data gathered in the study will also be managed, analysed and processed by myself.

If you agree to take part in the study, you will be asked to do the following tasks:

- Listen to a brief introduction to the study.
- Attempt a pre-test on the quiz server.
- You will be presented with an instruction sheet on how to start and use the application. You will use one of our Android phones in this study.
- Important: Before starting to use the application, ensure that you are connected to the WiFi.
- Once you have started using the application, just follow the instructions and answer the problems within the application. You are not allowed to go back or skip a problem. Just attempt all problems until the time is up.
- Once allocated time for using the application is over, or you have finished all the problems. You will do a post-test on the quiz server.

If you are confused about how to use the app, you may ask the instructor for further clarifications. Your feedback is important for the future development of PyKinetic. It will be ideal for you to finish all questions in the pre-test and post-test, but do not feel bad if you do not manage to do so.

You may receive a copy of the project results by contacting me at the conclusion of the project. Participation is voluntary and you have the right to withdraw at any stage without penalty. If you choose to withdraw, I will remove all information relating to you.

The results of the project may be published, but you may be assured of the complete confidentiality of data gathered in this investigation: your identity will not be made public. To ensure confidentiality, data gathered in this session will be stored in password-protected files and a password-protected computer and in locked storage facilities in the University of Canterbury. Only the researchers working on the project will have access to the data. Utmost care will be taken to maintain confidentiality of the participants and the results of this study. After 10 years following the PhD research, all data will be destroyed.

A thesis is a public document and will be available through the UC Library. The project is being carried out as a part of a PhD research project at the University of Canterbury in the Department of Computer Science and Software Engineering by Geela Fabic under the supervision of Dr. Antonija Mitrovic, who can be contacted at tanja.mitrovic@canterbury.ac.nz and Dr. Kourosh Neshatian who can be contacted at kourosh.neshatian@canterbury.ac.nz. They will be pleased to discuss any concerns you may have about participation in the project.

This project has been reviewed and approved by the University of Canterbury Human Ethics Committee, and participants should address any complaints to The Chair, Human Ethics Committee, University of Canterbury, Private Bag 4800, Christchurch (human-ethics@canterbury.ac.nz).
If you agree to participate in the study, you are asked to complete the consent form and return it to myself, Geela Fabic before taking part in the study.

Appendix D. Instruction Sheets

First Evaluation Study

Installation and Instruction Sheet

Department: Computer Science and Software Engineering
Email: geela.fabic@pg.canterbury.ac.nz

If you are using your own device, download and install the application from link shown in slide.

How to use PyKinetic:

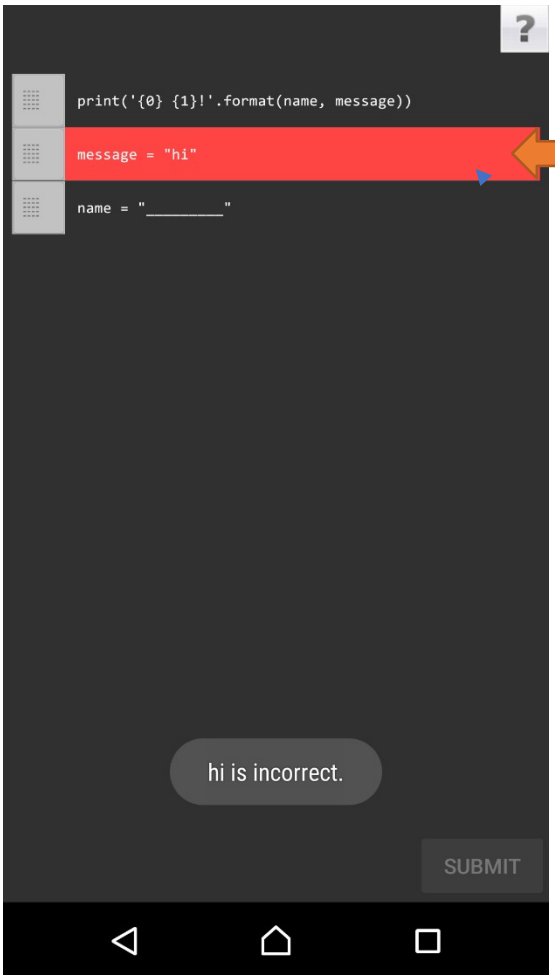
LOC – Line of Code

Rearrange LOCs by **dragging** the drag icon

Tap to view problem statement again

Long-click on incomplete LOCs to select (blue highlight represents **selected**.

Tap on **selected** LOCs to choose the correct missing keyword.



Once an option is selected, check your answer by **long-clicking** on the LOC again.

Red highlight indicates that your answer is **incorrect**. A **green** highlight that your answer is **correct**.

Once all incomplete LOCs are completely filled, Submit button will be enabled.

(This page of the instructions was only given to experimental group)

Self Explanation Question shown below: can only be attempted **ONCE**.

Dark Pink text indicates **incorrect** answers.

Green text indicates **correct** answers.

If you answered an Incomplete LOC correctly, you will be presented with a question regarding your correct answer. Choose one or more choices to answer the question.

If you answered an Incomplete LOC

Tap OK to submit your answer/s. You must attempt this question to go back to the main problem (seen underneath this question).

Tap RETURN to close the dialog box and return to the question you are working on.

Second Evaluation Study

Instruction Sheet

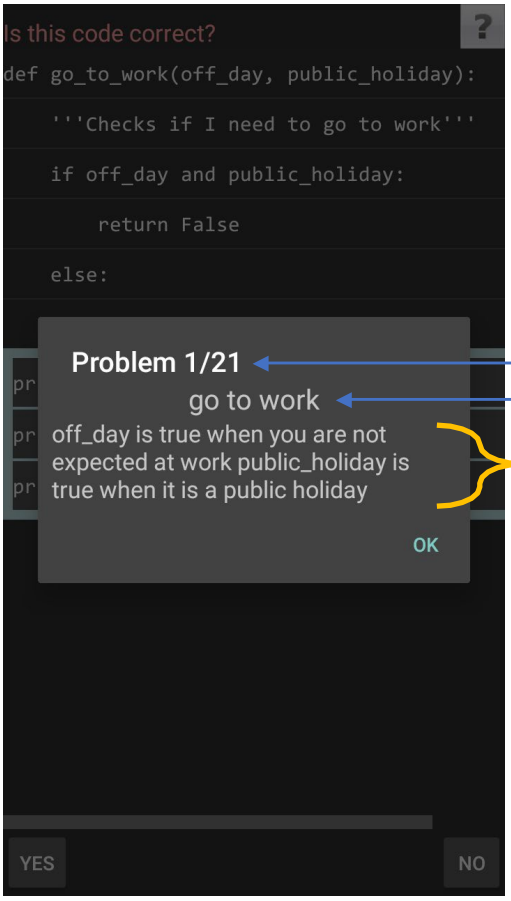
UNIVERSITY OF CANTERBURY
Te Whare Wānanga o Waitaha
CHRISTCHURCH NEW ZEALAND

Department: Computer Science and Software Engineering
Email: geela.fabic@pg.canterbury.ac.nz

How to use PyKinetic:

LOC – Line of Code

General tip: Interact with PyKinetic with clicks and long-clicks! ☺

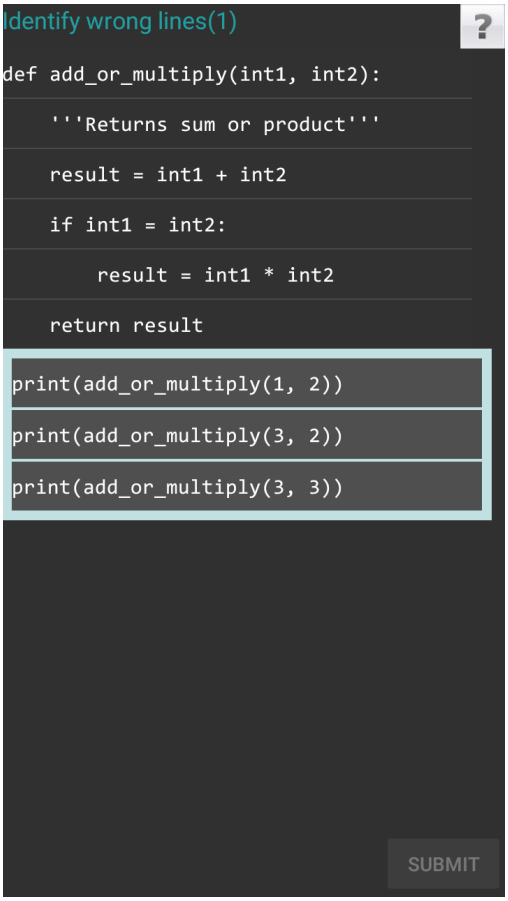


Tap to view problem question again

Problem number/Total problems

Problem name

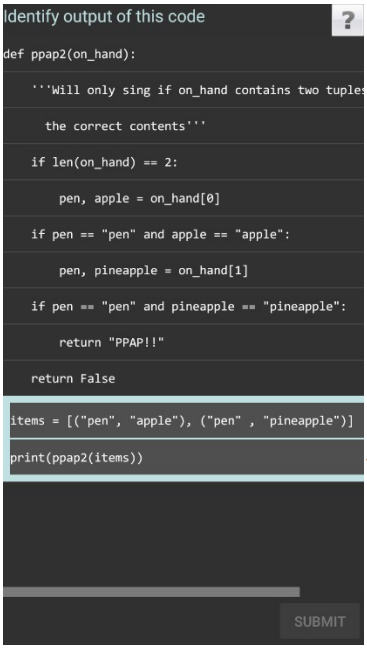
Problem description



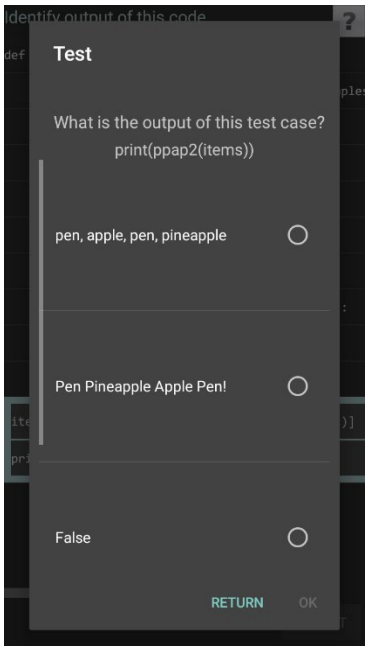
Long-click code to select/identify wrong lines. A blue highlight represents **selected**.

Long-click on test cases to view its output.

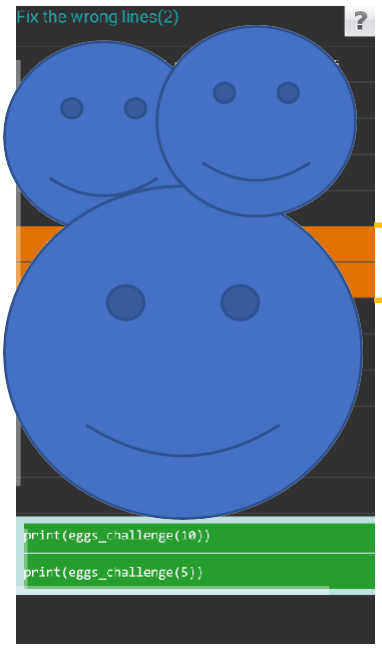
Once you selected at least one line, **click** on SUBMIT to check your answer



Long-click on test case to open dialog box to identify output

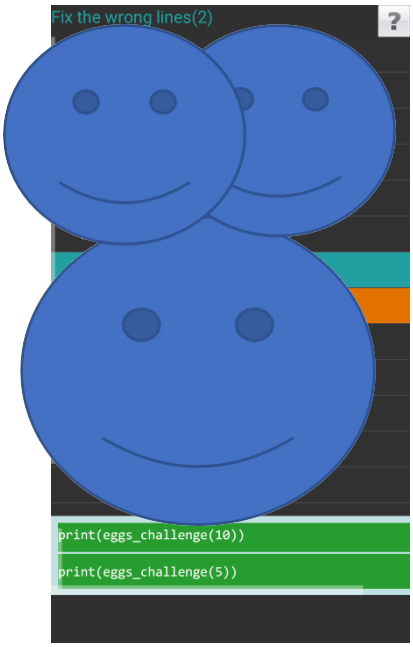


NOTE: Smiley faces not in app but just to hide answers.



Orange highlight indicates code that need to be fixed.

Red highlight indicates that your answer is **incorrect**. A **green** highlight that your answer is **correct**.



Long-click code to select a line to fix. A blue highlight represents **selected**.

Tap on **selected** code to view choices.

Once an option is selected, check your answer by **long-clicking** on the code again.

Third Evaluation Study

Instruction Sheet

UNIVERSITY OF CANTERBURY
Te Whare Wānanga o Waitaha
CHRISTCHURCH NEW ZEALAND

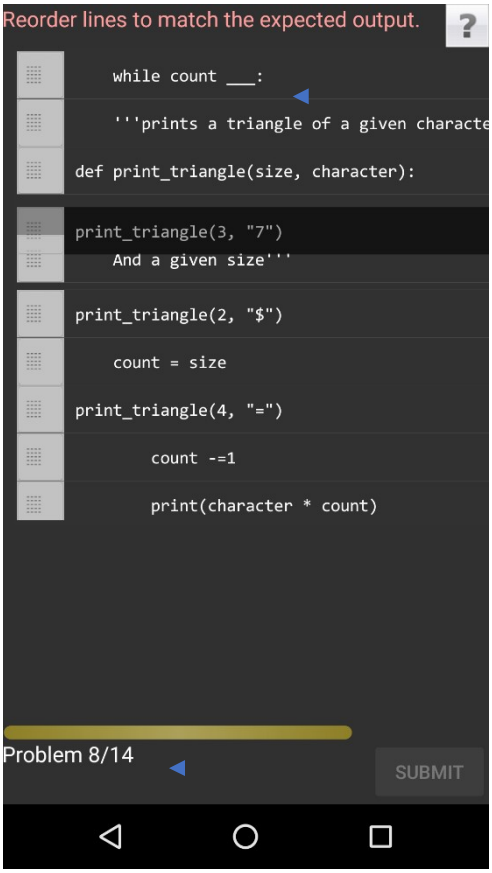
Department: Computer Science and Software Engineering
Email: geela.fabic@pg.canterbury.ac.nz

How to use PyKinetic:

LOC – Line of Code

General tip: Interact with PyKinetic with clicks and long-clicks! ☺

Parsons problems

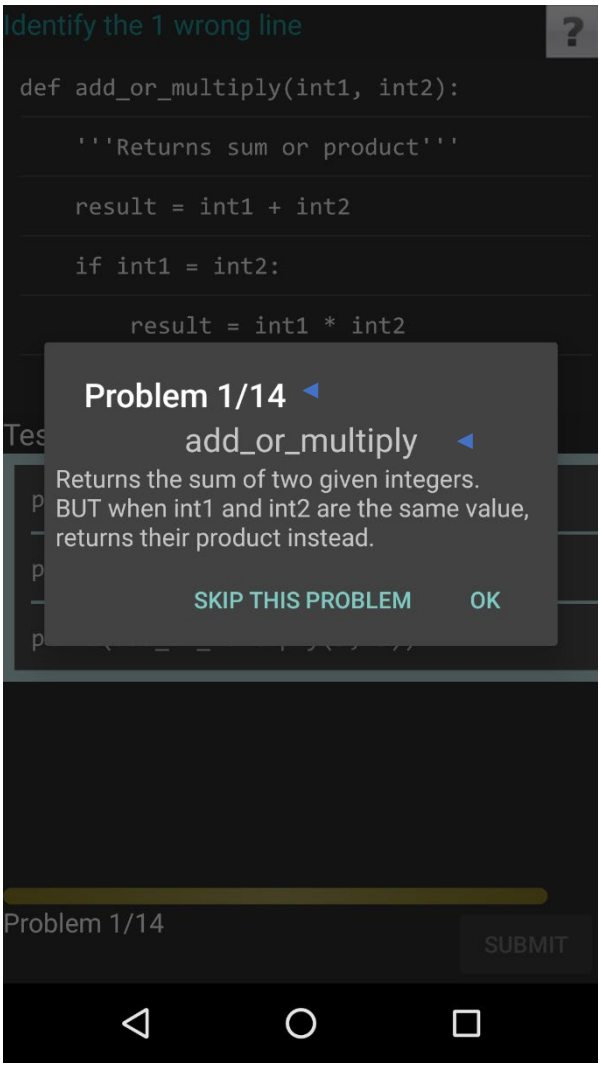


Drag and **drop** to reorder lines of code

Long-click on code with blank line to choose the correct keyword. A blue highlight represents **selected**.

Problem number/Total problems

Problem Details



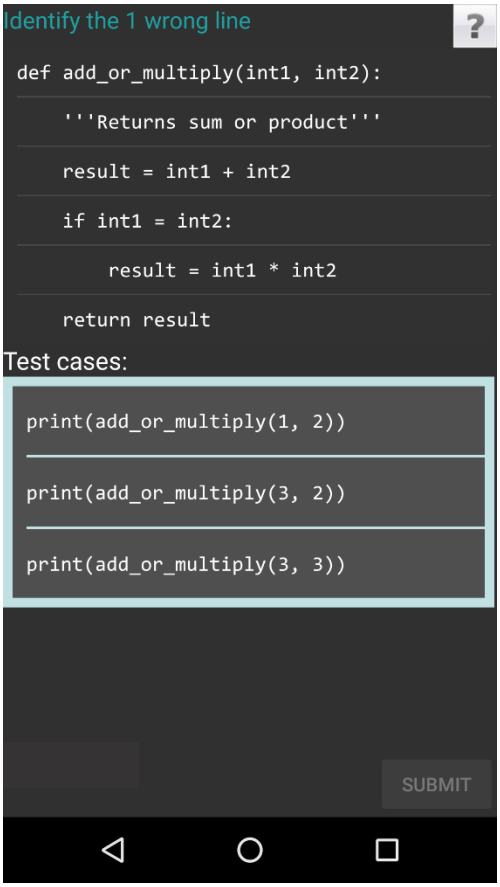
Tap to view problem question again

Problem number/Total problems

Problem name

Problem description

Identify wrong line

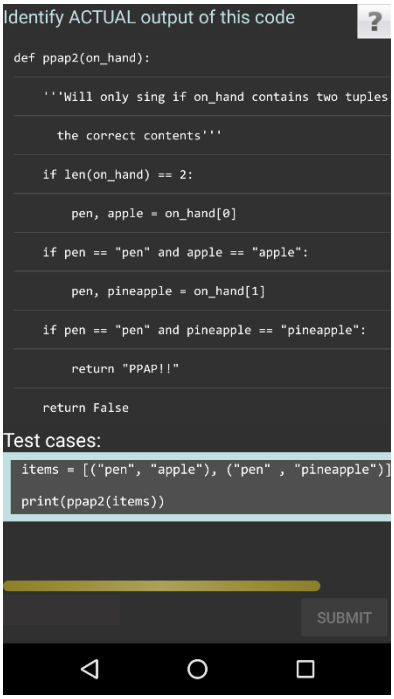


Long-click code to select/identify wrong lines. A blue highlight represents **selected**.

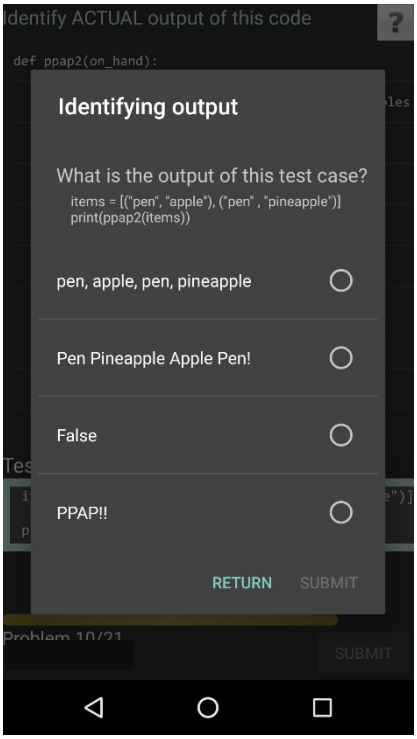
Long-click on a test case to view its output.

Once you selected at least one line, **click** on SUBMIT to check your answer

Identify output

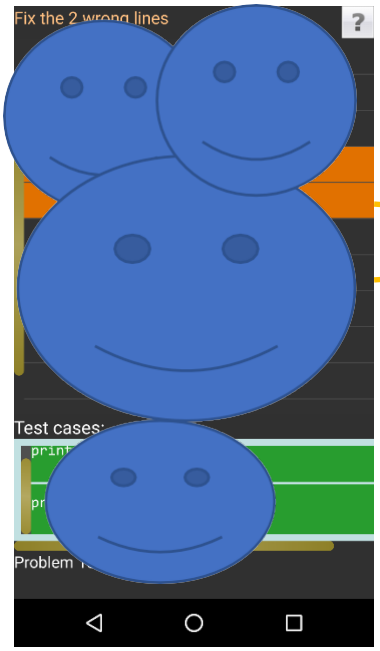


Long-click on a test case to open dialog box to identify its output



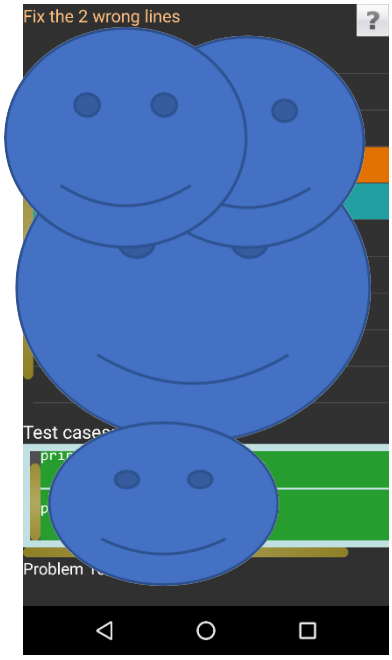
Fixing wrong lines

NOTE: Smiley faces not in app but just to hide answers.



Orange highlight indicates code that need to be fixed.

Red highlight indicates that your answer is **incorrect**. A **green** highlight that your answer is **correct**.



Long-click code to select a line to fix. A blue highlight represents **selected**.

Tap on **selected** code to view choices.

Once an option is selected, check your answer by **long-clicking** on the code again.

Appendix E. Consent Forms

Pilot Study

Consent Form for Pilot Study

Department: Computer Science and Software Engineering
Email: geela.fabic@pg.canterbury.ac.nz

Investigating the Effectiveness of a Python Tutor on a Mobile Platform for Novice Programmers

Consent Form for participants

I have been given a full explanation of this project and have had the opportunity to ask questions.
I understand what is required of me if I agree to take part in the research.
I understand that participation is voluntary and I may withdraw at any time without penalty. Withdrawal of participation will also include the withdrawal of any information I have provided should this remain practically achievable.

I understand that any information or opinions I provide will be kept confidential to the researcher and the supervisors for this research and that any published or reported results will not identify the participants. I understand that a thesis is a public document and will be available through the UC Library.

I understand that all data collected for the study will be kept in locked and secure facilities and/or in password protected electronic form and will be destroyed after five years after the Masters research.

I understand the risks associated with taking part and how they will be managed.
I understand that I am able to receive a report on the findings of the study by contacting the researcher at the conclusion of the project.

I understand that I can contact the researcher Geela Fabic at geela.fabic@pg.canterbury.ac.nz and/or supervisors Dr. Antonija Mitrovic at tanja.mitrovic@canterbury.ac.nz and Dr. Kourosh Neshatian at kourosh.neshatian@canterbury.ac.nz. If I have any complaints, I can contact the Chair of the University of Canterbury Human Ethics Committee, Private Bag 4800, Christchurch (human-ethics@canterbury.ac.nz)

By signing below, I agree to participate in this research project.

Name: _____

Date: _____

Signature: _____

Geela Fabic

First Evaluation Study (Ateneo de Manila)

Consent Form

Department: Computer Science and Software Engineering
Email: geela.fabic@pg.canterbury.ac.nz

Developing and Evaluating Activities for Increasing Engagement and Maximising Learning in a Mobile Programming Tutor

Consent Form for participants

I have been given a full explanation of this project and have had the opportunity to ask questions.
I understand what is required of me if I agree to take part in the research.
I understand that participation is voluntary and I may withdraw at any time without penalty. Withdrawal of participation will also include the withdrawal of any information I have provided should this remain practically achievable.

I understand that any information or opinions I provide will be kept confidential to the researcher and the supervisors for this research and that any published or reported results will not identify the participants. I understand that a thesis is a public document and will be available through the UC Library.

I understand that all data collected for the study will be kept in locked and secure facilities and/or in password protected electronic form and will be destroyed after ten years after the PhD research.

I understand the risks associated with taking part and how they will be managed.
I understand that I am able to receive a report on the findings of the study by contacting the researcher at the conclusion of the project.

I understand that I can contact the researcher Geela Fabic at geela.fabic@pg.canterbury.ac.nz and/or supervisors Dr. Antonija Mitrovic at tanja.mitrovic@canterbury.ac.nz and Dr. Kourosh Neshatian at kourosh.neshatian@canterbury.ac.nz. If I have any complaints, I can contact the Chair of the University of Canterbury Human Ethics Committee, Private Bag 4800, Christchurch (human-ethics@canterbury.ac.nz)

By signing below, I agree to participate in this research project.

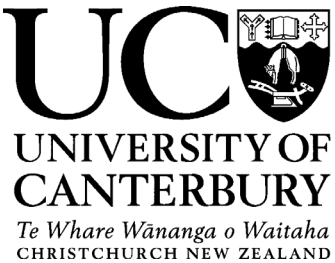
Name: _____

Date: _____

Signature: _____

First to Third Evaluation Study (University of Canterbury)

Consent Form



Department: Computer Science and Software Engineering
Email: geela.fabic@pg.canterbury.ac.nz

Developing and Evaluating Activities for Increasing Engagement and Maximising Learning in a Mobile Programming Tutor

Consent Form for participants

I have been given a full explanation of this project and have had the opportunity to ask questions.
I understand what is required of me if I agree to take part in the research.
I understand that participation is voluntary and I may withdraw at any time without penalty. Withdrawal of participation will also include the withdrawal of any information I have provided should this remain practically achievable.

I understand that any information or opinions I provide will be kept confidential to the researcher and the supervisors for this research and that any published or reported results will not identify the participants. I understand that a thesis is a public document and will be available through the UC Library.

I understand that all data collected for the study will be kept in locked and secure facilities and/or in password protected electronic form and will be destroyed after ten years after the PhD research.

I understand the risks associated with taking part and how they will be managed.

I understand that I am able to receive a report on the findings of the study by contacting the researcher at the conclusion of the project.

I understand that I can contact the researcher Geela Fabic at geela.fabic@pg.canterbury.ac.nz and/or supervisors Dr. Antonija Mitrovic at tanja.mitrovic@canterbury.ac.nz and Dr. Kourosh Neshatian at kourosh.neshatian@canterbury.ac.nz. If I have any complaints, I can contact the Chair of the University of Canterbury Human Ethics Committee, Private Bag 4800, Christchurch (human-ethics@canterbury.ac.nz)

By signing below, I agree to participate in this research project.

Name: _____

Date: _____

Signature: _____

Geela Fabic

Appendix F. Human Ethics Committees Approval Letters

UNIVERSITY OF CANTERBURY
Te Whare Wānanga o Waitaha
CHRISTCHURCH NEW ZEALAND

HUMAN ETHICS COMMITTEE

Secretary, Lynda Griffioen
Email: human-ethics@canterbury.ac.nz

Ref: HEC 2015/69/LR

27 August 2015

Geela Fabic
Department of Computer Science & Software Engineering
UNIVERSITY OF CANTERBURY

Dear Geela

Thank you for forwarding your Human Ethics Committee Low Risk application for your research proposal "Investigating the effectiveness of a python tutor on a mobile platform for novice programmers".

I am pleased to advise that the application has been reviewed and approved.

With best wishes for your project.

Yours sincerely

Lindsey MacDonald
Chair, Human Ethics Committee

HUMAN ETHICS COMMITTEE

Secretary, Rebecca Robinson
Telephone: +64 03 364 2987, Extn 45588
Email: human-ethics@canterbury.ac.nz

Ref: HEC 2016/24/LR-PS

6 May 2016

Geela Venise Fabic
Computer Science and Software Engineering
UNIVERSITY OF CANTERBURY

Dear Geela Venise

Thank you for submitting your low risk application to the Human Ethics Committee for the research proposal titled "Developing and Evaluating Activities for Increasing Engagement and Maximising Learning in a Mobile Programming Tutor".

I am pleased to advise that this application has been reviewed and approved.

Please note that this approval is subject to the incorporation of the amendments you have provided in your email of 2nd May 2016.

With best wishes for your project.

Yours sincerely

R. Robinson

pp.

Jane Maidment
Chair, Human Ethics Committee

HUMAN ETHICS COMMITTEE
Secretary, Rebecca Robinson
Telephone: +64 03 364 2987, Extn 45588
Email: human-ethics@canterbury.ac.nz

Ref: HEC 2016/24/LR-PS Amendment 1

2 August 2016

Geela Venise Fabic
Computer Science and Software Engineering
UNIVERSITY OF CANTERBURY

Dear Geela

Thank you for your request for an amendment to your research proposal "Developing and Evaluating Activities for Increasing Engagement and Maximising Learning in a Mobile Programming Tutor" as outlined in your email dated 22nd July 2016.

I am pleased to advise that this request has been considered and approved by the Human Ethics Committee; **subject to the following:**

We approve the amendment for research in other University settings, but new application to ERHEC will be required for conducting the research in high schools.

Yours sincerely

R. Robinson

pp.

Jane Maidment
Chair, Human Ethics Committee

UNIVERSITY OF CANTERBURY
Te Whare Wānanga o Waitaha
CHRISTCHURCH NEW ZEALAND

HUMAN ETHICS COMMITTEE

Secretary, Rebecca Robinson
Telephone: +64 03 364 2987, Extn 45588
Email: human-ethics@canterbury.ac.nz

Ref: 2016/39/ERHEC

7 September 2016

Geela Venise Fabic
Computer Science and Software Engineering
UNIVERSITY OF CANTERBURY

Dear Geela

Thank you for providing the revised documents in support of your application to the Educational Research Human Ethics Committee. I am very pleased to inform you that your research proposal "Developing and Evaluating Activities for Increasing Engagement and Maximising Learning in a Mobile Programming Tutor" has been granted ethical approval.

Please note that this approval is subject to the incorporation of the amendments you have provided in your email of 7th September 2016.

Should circumstances relevant to this current application change you are required to reapply for ethical approval.

If you have any questions regarding this approval, please let me know.

We wish you well for your research.

Yours sincerely

pp R. Robinson

Patrick Shepherd
Chair
Educational Research Human Ethics Committee

Please note that ethical approval relates only to the ethical elements of the relationship between the researcher, research participants and other stakeholders. The granting of approval by the Educational Research Human Ethics Committee should not be interpreted as comment on the methodology, legality, value or any other matters relating to this research.

F E S

ATENEO DE MANILA UNIVERSITY
UNIVERSITY RESEARCH ETHICS OFFICE

20 September 2016

Dr. Ma. Mercedes T. Rodrigo Professor
Department of Information Systems
and Computer Science
School of Science and Engineering

Re: Exemption from Ethics Review

Dear Dr. Rodrigo,

Peace!

I have reviewed your research protocol submission entitled "Developing and Evaluating Activities for Increasing Engagement and Maximizing Learning in a Mobile Programming Tutor" and validate that it is exempt from review by the University Research Ethics Committee. You can therefore proceed to conduct your study.

To maintain the status of exemption, you are to ensure the confidentiality of the respondents' data, that respondents provide voluntary consent to participate in the study, and that they may withdraw their participation and their data at any point without negative consequence to them.

You are expected to comply with the policies of the Ateneo de Manila University's Code of Research Ethics and the National Ethical Guidelines for Health Research. Kindly inform the UREO immediately if any of the conditions and procedures of your research design and methodology change. Such changes may entail a change in your study's exempt status. Likewise, any unanticipated problems that involve risks to participants must be reported to the UREO. Such reports must be made in writing and sent to the University Research Ethics Office (UREO).

We wish you success in your research undertaking.

Respectfully yours,

Liane Peña Alampay
Director

Rm 204, 2/F Xavier Hall, Katipunan Avenue, Loyola Heights, 1108 Quezon City, Philippines
Tel +63 2 426 6001 ext 4030, Email univresearchethics@ateneo.edu

Appendix G. List of Publications

Journal Publications:

1. Fabic, G. V. F., Mitrovic, A., & Neshatian, K. (2018). Investigating the effects of learning activities in a mobile Python tutor for targeting multiple coding skills. *Research and practice in technology enhanced learning*, *13*(1), 23.

2. Fabic, G. V. F., Mitrovic, A., & Neshatian, K. (2019). Evaluation of Parsons Problems with Menu-Based Self-Explanation Prompts in a Mobile Python Tutor. *International Journal of Artificial Intelligence in Education*, 1-29. ***(pre-print not included in this thesis)***

Conference Papers:

1. Fabic, G., Mitrovic, A., & Neshatian, K. (2016). Towards a mobile python tutor: understanding differences in strategies used by novices and experts. In *Proc. 13th Int. Conf. Intelligent Tutoring Systems* (Vol. 9684, pp. 447-448).

2. 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. In *Proc. 9th Workshop on Technology Enhanced Learning by Posing/Solving Problems/Questions* (pp. 434-444).

3. Fabic, G. V. F., Mitrovic, A., & Neshatian, K. (2017). Investigating the effectiveness of menu-based self-explanation prompts in a mobile Python tutor. In *International Conference on Artificial Intelligence in Education* (pp. 498-501). Springer, Cham.

4. Fabic, G. V. F., Mitrovic, A., & Neshatian, K. (2017). Learning with Engaging Activities via a Mobile Python Tutor. In *International Conference on Artificial Intelligence in Education* (pp. 613-616). Springer, Cham.

5. Fabic, G., Mitrovic, A., & Neshatian, K. (2017). A comparison of different types of learning activities in a mobile Python tutor. In *Proc. 25th International Conference on Computers in Education*, (pp. 604-613).

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

7. Fabic, G. V., Mitrovic, A., & Neshatian, K. (2018). Supporting Novices and Advanced Students in Acquiring Multiple Coding Skills. In *Proc. 26th International Conference on Computers in Education*, (pp. 65-70).

Appendix H. RPTEL Journal Paper

Fabic *et al. Research and Practice in Technology Enhanced Learning* (2018) 13:23
https://doi.org/10.1186/s41039-018-0092-x

Research and Practice in Technology Enhanced Learning

RESEARCH Open Access

Investigating the effects of learning activities in a mobile Python tutor for targeting multiple coding skills

Geela Venise Firmalo Fabic, Antonija Mitrovic* and Kourosh Neshatian

* Correspondence: tanja.mitrovic@canterbury.ac.nz
Department of Computer Science and Software Engineering, University of Canterbury, Private Bag 4800, Christchurch 8041, New Zealand

Abstract

Mobile devices are increasingly being utilized for learning due to their unique features including portability for providing ubiquitous experiences. In this paper, we present PyKinetic, a mobile tutor we developed for Python programming, aimed to serve as a supplement to traditional courses. The overarching goal of our work is to design coding activities that maximize learning. As we work towards our goal, we first focus on the learning effectiveness of the activities within PyKinetic, rather than evaluating the effectiveness of PyKinetic as a supplement resource for an introductory programming course. The version of PyKinetic (PyKinetic_DbgOut) used in the study contains five types of learning activities aimed at supporting debugging, code-tracing, and code writing skills. We evaluated PyKinetic in a controlled lab study with quantitative and qualitative results to address the following research questions: (R1) Is the combination of coding activities effective for learning programming? (R2) How do the activities affect the skills of students with lower prior knowledge (novices) compared to those who had higher prior knowledge (advanced)? (R3) How can we improve the usability of PyKinetic? Results revealed that PyKinetic_DbgOut was more beneficial for advanced students. Furthermore, we found how coding skills are interrelated differently for novices compared to advanced learners. Lastly, we acquired sufficient feedback from the participants to improve the tutor.

Keywords: Python tutor, Mobile learning, Programming skills, Code writing, Code tracing, Novice students, Advanced students

Introduction

Mobile handheld devices provide distinctive features which are increasingly being utilized for learning. Pea and Maldonado (2006) summarized the unique attributes of mo- bile devices for learning into seven features: size and portability, small screen size, computing power and modular platform, communication ability, wide range of applica- tions, synchronization and back-up abilities, and stylus-driven interface. These attri- butes are largely still relevant at this day and age, but nowadays most handheld devices, like smartphones, provide touchscreen surfaces where users can interact with directly using their bare hands without using a stylus or a thumb-pad keyboard. Smart- phones are used to access course material, listen to podcasts, watch instructional vid- eos, and communicate with peers (Dukic et al. 2015). Smartphones are also being used

in classrooms, to increase interaction between the teacher and students or between students (Au et al. 2015; Anshari et al. 2017).

Python is widely used as the first programming language in introductory program- ming courses (Guo 2013). It is also the number one programming language in 2018 based on IEEE Spectrum Ranking (Cass and Parthasaradhi 2018). As the popularity of Python and smartphones are increasing, we aim to provide opportunities for learners to continue enhancing their Python programming skills even when they are away from personal computers. Being a mobile tutor, we hope that it would appeal to the new gen- eration of students; allowing them to continue learning outside of a classroom setting. We present PyKinetic (Fabic et al. 2016a; 2016b), a mobile Python tutor, developed using Android SDK to teach Python 3.x programming. PyKinetic is not meant to be a stand-alone learning resource; instead, the tutor is a complement to traditional lecture and lab-based introductory programming courses. Traditional code writing exercises may be difficult on a small-screened device such as a smartphone, as the keyboard usu- ally obstructs half of the smartphone screen. For that reason, learning activities in PyKi- netic require only tap and long-click interactions.

The overall aim of our project is to design activities that will maximize learning (Fabic et al. 2017a) on a mobile device, to provide an avenue for students to continue learning even when outside of a classroom setting and/or away from a personal computer. In this paper, we present a set of learning activities that focus on debugging, code-tracing, and code-writing skills. We present a version of PyKinetic—PyKinetic_DbgOut, which contains debugging and output prediction exercises, designed to support acquisition of debugging and code tracing skills (Fabic et al. 2017b). We conducted a study with PyKinetic_DbgOut to answer the following research questions:

(R1) Is the combination of coding activities effective for learning programming?
(R2) How do the activities affect the skills of students with lower prior knowledge (novices) compared to those with higher prior knowledge (advanced)?
(R3) How can we improve the usability of PyKinetic?

We will answer our research questions by providing both quantitative and qualitative evidence. The paper is structured as follows. In the next section, we present related work on mobile learning, followed by a section on coding skills. Afterwards, we de- scribe learning activities in PyKinetic_DbgOut, followed by our experimental setup; then the findings. Following the section on findings, we present our discussion, and fi- nally our conclusions.

Mobile learning

Mobile learning is defined as learning that transpires in an undetermined setting or learning through mobile technologies (O'Malley et al. 2005). Park (2011) devised a pedagogical framework for mobile learning activities and classified them into four types: (1) high transactional distance and socialized (HS), (2) high transactional distance and individualized (HI), (3) low transactional distance and socialized (LS), and 4) low transactional distance and individualized (LI).

The first type (HS) is when the pedagogical strategies are mostly driven by mobile learning applications and when students are heavily participating with peers by communicating to learn together. The second type (HI) is similar to HS; the learning is directed by the mobile application, but students in this type learn individually instead of collaborating with peers. The third type (LS) is where a mobile application provides a lenient structure for learning, but students work together with their peers and teachers. Lastly, LI also provides a lenient structure like LS but students in this type learn independently. Our future goal for PyKinetic is aimed at type LI: mobile learning activities where students can independently learn Python anytime and anywhere. However, the version of PyKinetic presented in this paper can be classified as HI, as it poses a higher transactional distance (i.e., low level of learner autonomy), with a fixed order of learn- ing activities for evaluation purposes.

Apart from one-on-one teaching like in PyKinetic, mobile learning systems are also being utilized to support diverse learning situations (Oyelere et al. 2018). Mobile appli- cations are implemented to support an assortment of pedagogical strategies such as inquiry learning (Shih et al. 2010; Jones et al. 2013; Nouri et al. 2014; Sun and Looi 2017), flipped classrooms (Wang 2016; Grandl et al. 2018), game-based learning (Klopfer et al. 2012; Perry and Klopfer 2014; Su and Cheng 2015; Vinay et al. 2013), co- operative learning (Roschelle et al. 2010), collaborative learning (Wong et al. 2017), competition-based learning (Hwang and Chang 2016), blended classroom learning (Wang et al. 2009), exploratory learning (Liu et al. 2012), and context-aware learning (Sun e al. 2015). There are also mobile applications developed as a learning manage- ment system (LMS) (Wen and Zhang 2015; Oyelere et al. 2018), and as a massive open online course (MOOC) (Grandl et al. 2018). Moreover, the number of applications that support learning is growing rapidly in various instructional domains, such as in medi- cine (Vinay and Vishal 2013; Gavali et al. 2017), science (Liu et al. 2012; Klopfer et al. 2012; Jones et al. 2013; Nouri et al. 2014; Perry and Klopfer 2014; Su and Cheng 2015; Sun and Looi 2017), social science (Shih et al. 2010; Hwang and Chang 2016), language learning (Wang et al. 2009; Kim and Kwon 2012; Nakaya and Murota 2013; Sun et al. 2015; Wang 2016; Wong et al. 2017), mathematics (Roschelle et al. 2010; Wen and Zhang 2015), and computing education (Hürst et al. 2007; Karavirta et al. 2012; Boticki et al. 2013; Vinay et al. 2013; Wen and Zhang 2015; Mbogo et al. 2016; Grandl et al. 2018; Oyelere et al. 2018). Furthermore, mobile learning is proved to be effective for learning and motivating learners from various age groups, such as children and pre-teens aged between 7 and 14 (Roschelle et al. 2010; Shih et al. 2010; Liu et al. 2012; Klopfer et al. 2012; Vinay et al. 2013; Nouri et al. 2014; Su and Cheng 2015; Hwang and Chang 2016; Wong et al. 2017; Sun and Looi 2017; Grandl et al. 2018), teenage students ages 14–16 (Perry and Klopfer 2014; Wang 2016), undergraduate and graduate students aged 16–35 (Roschelle et al. 2010; Boticki et al. 2013; Nakaya and Murota 2013; Mbogo et al. 2016; Sun et al. 2015; Oyelere et al. 2018), university students and teachers (Wen and Zhang 2015), and working professionals (Wang et al. 2009).

There are many mobile applications that support various aspects of computing edu- cation such as supporting entire courses through an LMS (Oyelere et al. 2018; Wen and Zhang 2015), learning algorithm executions through visualizations (Boticki et al. 2013; Hürst et al. 2007), control-flow learning (Karavirta et al. 2012; Vinay et al. 2013; Fabic et al. 2017c; Grandl et al. 2018), and code writing (Mbogo et al. 2016). Oyelere et

al. (2018) implemented MobileEdu, an LMS for third year students in a systems ana- lysis and design course. The goal of MobileEdu was to alleviate poor engagement of students and to provide support for lecturers in managing the students. Oyelere and colleagues conducted a study where they compared students' learning with traditional lectures (control) to students learning almost entirely through MobileEdu (experimen- tal). MobileEdu had communication features which allowed students in the experimen- tal group to interact with their instructor despite not having face to face lectures. Results revealed that students who learned via MobileEdu learned significantly more than the control group. They also conducted a survey which revealed that the experi- mental group had significantly improved perception towards the course compared to the control group. Wen and Zhang (2015) also presented an LMS referred to as Micro-Lecture Mobile Learning System (MMLS). The system contained 12 courses. The results showed that participants who used the MMLS in Data Mining, Digital Signal Processing, and MATLAB courses showed significantly higher final exam marks than previous course intakes before introducing the MMLS.

Boticki et al. (2013) also conducted a semester-long study. The authors introduced SortKo, an Android application for teaching sorting algorithms with the help of visuali- zations. The authors found that SortKo learners learned 30% more than those who did not use SortKo, and their survey results verified that it helped motivate the students. Hürst et al. (2007) also introduced animations to help with understanding executions of algorithms. However, the study by Hürst and colleagues was more concerned on HCI implications. They compared differences in learning between devices (laptop vs. iPod with video capability) and modalities (audio vs. no audio).

There are also several implementations in control-flow learning with different ap- proaches. For example, Vinay et al. (2013) and Grandl et al. (2018) both had gamification approaches. However, Grandl and colleagues had a flipped classroom design where learners create various games via code like Scratch. Moreover, other implementations used Parsons problems (Parsons and Haden 2006) where syntactically correct code is given but needs to be reordered in the right order to match the given expected output (Karavirta et al. 2012; Fabic et al. 2017c). The study by Karavirta et al. (2012) was focused on supporting automated feedback for Parsons problems within their system MobilePar- sons, whereas our previous work (2017c) explored the effectiveness of *menu-based self-ex- planation* (*SE*) *prompts*. Menu-based SE prompts are learning activities which provide choices from a menu, designed to promote deeper learning by helping induce mental jus- tifications which are not directly presented by the material (Wylie and Chi 2014).

Lastly, Mbogo et al. (2016) conducted a 2-hr-long qualitative study where their mobile learning system provided code writing exercises in Java by designing static scaffolding. The learners were able to collapse parts of the code and toggle between collapsed and full versions of their programs to support Java code writing on a smartphone. Participants remarked that the scaffolding for code segmentations made the application easier to use.

There are very few mobile learning tutors presented in the literature specifically for enhancing programming skills. Our work presented in this paper is similar to work by Karavirta et al. (2012) and Mbogo et al. (2016), with the notion of providing program- ming exercises to support learning. However, instead of only providing one type of ac- tivity as in those two projects, we provide various tasks to target several coding skills. Our pedagogical strategy is an implementation with a component-skills perspective

(McArthur et al. 1988). We aim to fill the gap and provide an opportunity for learners to continue practicing their programming skills even when not in a lecture setting. Furthermore, we aim to provide learners an avenue to enhance several coding skills while they are away from a personal computer by supplying a combination of various coding activities.

Coding skills

Novice programmers are slow in solving problems due to the lack of declarative and/or procedural knowledge (Anderson, 1982). In programming, declarative knowledge comprises of the syntax of the programming language and familiarity with code constructs. According to Winslow (1996), it takes 10 years for a learner to become an expert programmer. Learners often lack mental models and are unable to translate a problem into manageable tasks (Winslow 1996). The difficulty in structuring code might be evidence of a deficiency of procedural knowledge. Some students perceive code as series of instructions that are expected to execute in the specified order (Ahmadzadeh et al. 2005). Students have difficulties in comprehending the execution order, predicting the output, debugging and code writing (Pea 1986).

A variety of skills necessary for programming has 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 and Cheng 2018). Further evidence proves re- lationships between code tracing, code writing, and code explaining (Lister et al. 2009; Venables et al. 2009). A strong positive correlation was found between code tracing and code writing (Lister et al. 2010). Harrington and Cheng (2018) conducted a study with 384 students, to investigate whether a gap exists between the ability of students to trace and write code. The study was conducted in an examination setting, where stu- dents were given two questions. The participants were randomly assigned to perform code tracing on one question and code writing on other. The results show that 56% of the students had almost no gap between their code tracing and code writing skills. For the remaining students who had of at least two out of eight marks, a strong negative correlation was found between the learners' performance in the course and the size of the gap. Regardless of whether the student was better in either code tracing or code writing, authors suggest that a large gap was more likely due to the student struggling in the course. The authors found that underachieving students were struggling with un- derstanding the core programming concepts. The conclusion of the authors is also sup- ported by literature, as learners need both declarative knowledge and procedural knowledge (Anderson, 1982). Students often struggle to translate problems into man- ageable tasks due to the absence of mental models (Winslow 1996). Ahmadzadeh et al. (2005) conducted a study on the debugging patterns of novices and found that most learners competent in debugging were also advanced programmers (66%). However, only 39% of advanced programmers were also competent in debugging. These findings provide evidence that debugging someone else's program requires a higher order of skill than code writing.

Learning activities in PyKinetic

The problems in PyKinetic_DbgOut consist of the problem description, code (containing 0–3 incorrect Lines of Code LOCs), 1–3 activities, and 1–3 questions for each activity. There are five types of activites (Table 1): three types of debugging

activities, and two types of output prediction activities. We first define some terminology used in the rest of the paper.

- An *activity* is an exercise based on a description, a code snippet, and possibly additional information.
- A *problem* is a task which may contain one or more coding activities, where activities are presented one at a time.
- A *test case* is an executable code which may contain more than one line of code; consists of parameters and calls to one or more functions.
- *Actual output* is the true displayed result of a program when executed, regardless of whether the code contains any errors.
- *Expected output* is the anticipated displayed result when a program which contains errors is "fixed" based on the problem requirements.

In *Dbg_Read* activities, the learner is given test cases with the actual output; the learner's task is to specify whether the given code is correct or not. Figure 1 shows an example of Dbg_Read where a learner is first given the problem description (Fig. 1, left). Function *go_to_work* should display True or False if a person is required to go to work, based on the value of two boolean parameters: *off_day* and *public_holiday*. Variable off_day is True if today is the person's day off work, and False if it is a normal working day. Variable public_holiday is True if today is a public holiday, and False if it is regular working day. The learner then needs to respond on whether the code is error-free or not, based on the given code and the problem description. If the code contains error/s, it is not mandatory to determine the part of the code which causes the error. The learner is also given test cases where PyKinetic_DbgOut displays the ac- tual output of a test case when executed when the learner long-presses on the test case. In Fig. 1 (middle), it shows that for the first test case, the output is True when the par- ameter off_day is True and public_holiday is False. The same output of True is also returned when off_day is False and public_holiday is True. The test cases show that the function contains an error since the function should only return True if the person needs to go to work. In Fig. 1 (right), the learner answered incorrectly, so PyKine- tic_DbgOut highlighted the LOC causing the error in red.

The second type of debugging activities (*Dbg_Ident*) provides similar information to the learner but instead of asking whether the code is error-free or not, the learner is made aware that the code contains errors by specifying the number of incorrect LOCs to be identified. The learner is required to identify one to three incorrect LOCs in one problem. An example is shown in Fig. 2 (left screenshot), where the student needs to identify two incorrect lines (the lines the student selected are highlighted in blue). The

Table 1 Five types of debugging and output prediction activities in PyKinetic_DbgOut

Type of question	Task	Additional information given
Dbg_Read	Is the code correct? (*Yes* or *No)*	Test cases with actual output
Dbg_Ident	Identify *n* erroneous LOCs (*n* is given)	Test cases with actual output
Dbg_Fix	Fix erroneous LOCs (by tapping through given choices)	Test cases with expected output
Out_Act	Select actual output of the code	Test cases
Out_Exp	Select expected output of the code	Test cases

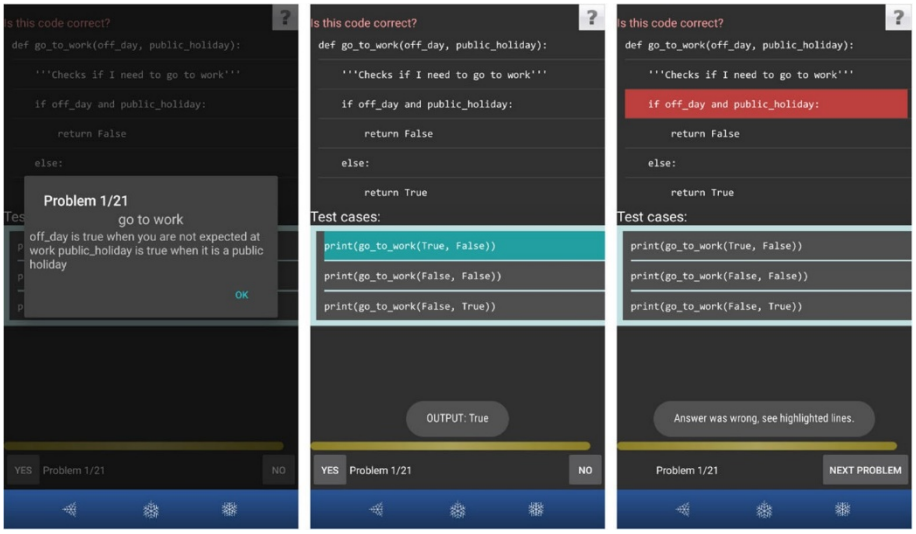


Fig. 1 Example of a Dbg_Read activity in PyKinetic_DbgOut (left: problem description; middle: code with test cases, output shown is the actual output of the highlighted line; right: feedback when learner answered incorrectly)

third type of debugging activities is *Dbg_Fix*, which starts with requiring the student to identify incorrect lines (Dbg_Ident), and then to fix them (Fig. 2, right). To fix incorrect LOCs, the student needs to select the correct option from given choices. In the screenshot shown in Fig. 2 (right), the student has completed the line highlighted in green, and needs to work on the other line (highlighted in red) as it was answered incorrectly. Each output prediction activity 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 (Fig. 2, middle). For example, if the code is erroneous, the actual output may be none with an error displayed (Fig. 2, middle, last option). On the contrary, in *Out_Exp*

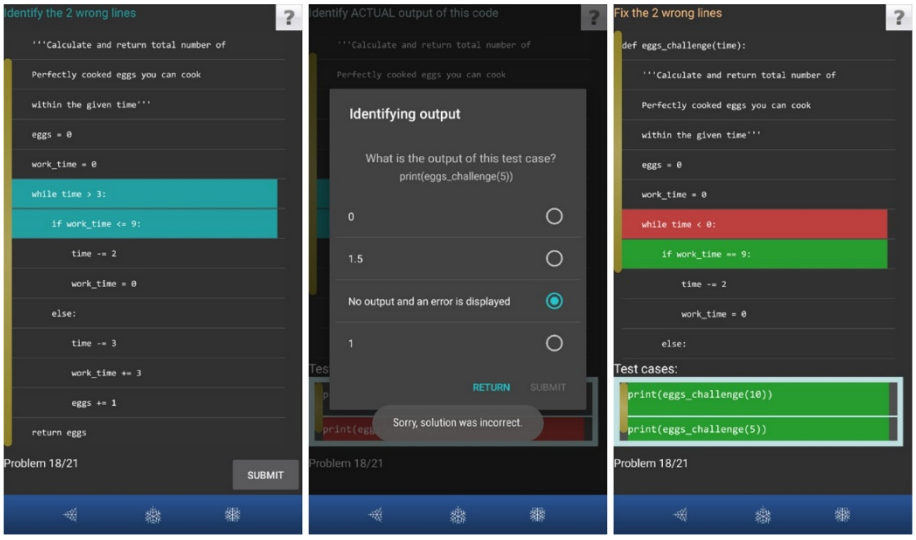


Fig. 2 Example of Dbg_Ident –> Out_Act –> Dbg_Fix activities in PyKinetic_DbgOut (left: Dbg_Ident; middle: Out_Act; right: Dbg_Fix)

activities, the student specifies the expected code output matching the problem description. All given Out_Exp problems, first starts with a Dbg_Ident activity where the student first needs to identify the error in code.

Figure 3 shows an example of a problem with Out_Exp where PyKinetic_DbgOut first shows the problem description (left). The problem contains a function with two parameters: a string parameter *pay_method*, and a list parameter *items_prices*. The problem requires to sum all the values in the given list items_prices and to put the sum into the variable *total*. Afterwards, the value of total is adjusted based on the string parameter pay_method. An additional 10% is added to variable total if the variable pay_method contains the string "credit". However, if pay_method contains the string "cash," the variable total should be deducted by 1%. In all other cases, the value of the variable total should remain the same. Lastly, the code should return the value of total. In Fig. 3 (middle), the learner successfully identified the erroneous LOC in the code (highlighted in turquoise) and is now asked to identify the expected output of the code for each given test case. In Fig. 3 (right), the learner correctly identified the expected output in the given test case (shown in green font). Notice that because the given first parameter (pay_method) contained "eftpos," the expected output should be the sum of all integers in the given list without any changes. If the question asked for the actual output, because of the error in the code (highlighted in turquoise in Fig. 3, middle), the answer would have been 108.9. However, the question was asking about the expected output so the correct answer was 110.

PyKinetic_DbgOut has 21 problems provided in a fixed order. There were seven levels of complexity, each containing 2–4 problems (Table 2). 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 activity. For example, problem one is a code reading problem, containing only one Dbg_Read activity. 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.

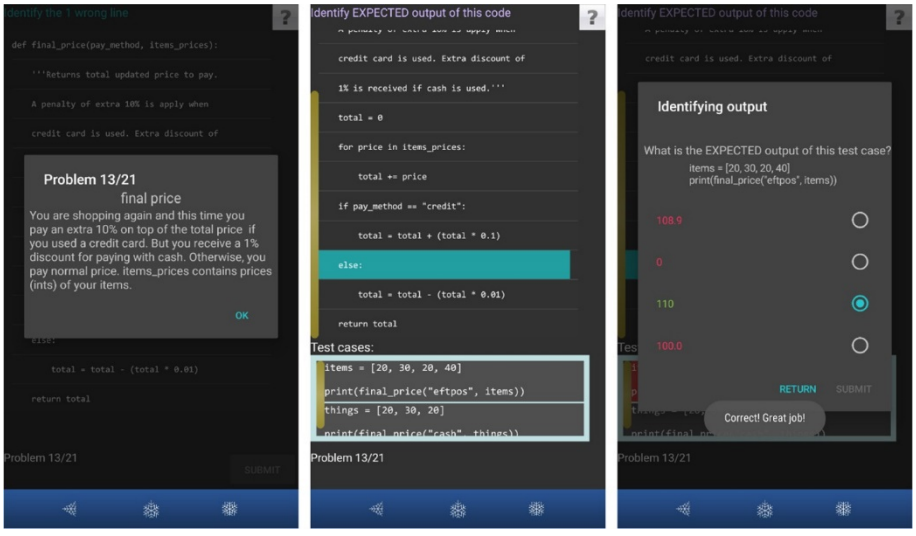


Fig. 3 Example of an Out_Exp activity in PyKinetic_DbgOut (left: problem description; middle: code with erroneous LOC already identified, right: output prediction question for the first testcase)

Table 2 Combinations of activities in levels 1–7

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

Each problem in levels 4–7 contains 2–3 activities and covered same topics as in the earlier levels as well as *for* loops, *while* loops, and importing a module. Problems on these levels started by requiring the student to identify incorrect LOCs (Dbg_Ident). After that, levels four and five are followed by output prediction activities: 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 activity. The code fixing was achieved by tapping through the options with the lines changing for each tap, instead of showing a separate dialog for the options like work of Ihantola et al. (2013). Lastly, level seven contains three types of activities in each problem: identifying erroneous LOCs (Dbg_Ident), identifying actual output (Out_Act), and fixing erroneous LOCs (Dbg_Fix). The problem illustrated in Fig. 2 belongs to level seven. It is important to note that the ordering of the problems was 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 has fewer lines and topics.

Experimental design

We conducted a controlled lab study with PyKinetic_DbgOut, focused on investigating the effectiveness of the activities in the tutor. We did not start by conducting a semester-long classroom study as we wanted to ensure the effectiveness of the activities first before exploring the effectiveness of the tutor as a supplement to a course.

Participants

In the controlled study, we had 37 participants recruited from an introductory programming course at the University of Canterbury. We eliminated data about two participants as they have not finished the study and present the findings from 35 participants (23 males and 12 females) in the rest of the paper. The study included a brief survey on demographics. Four participants did not disclose their ages, so the aver- age age of the rest of the participants was 21 (sd = 6.9). There was one older participant aged 54, who was found to perform similarly with other participants.

English was the first language of 23 participants (66%). Only eight participants (23%) had previous programming knowledge. We asked students to rate themselves on their programming experience on a Likert scale from one (less experienced) to ten (more experienced), and the average rating was 3.43 (sd = 1.9). We also asked students if they have previously used a Python programming tutor which is not necessarily on a mobile device, and only five participants (14%) have used one before, while the rest have not used any.

Sixteen participants (46%) owned an Android smartphone, 16 participants (46%) owned an iOS smartphone, and one participant owned both an Android smartphone and an iOS smartphone. Interestingly, two participants said that they did not own a smartphone. For those 33 participants who owned smartphones, we asked their average daily time spent on their devices; 16 participants (49%) claimed to use their smart- phones for 3–8 h, 12 learners (34%) use their smartphones for 1–3 h, 3 participants (9%) use their smartphones for less than an hour, and 2 participants (6%) use their smartphones for more than 8 h.

Method and materials

Each student participated in a single session. The sessions were 2 hr long, with 1–9 participants per session. The participants provided informed consent, followed by an 18-min pre-test, which included questions on demographics and programming back- ground. We then gave brief instructions on using the tutor and provided Android smartphones which were mostly running on Android 6.0 (Marshmallow) and had screen sizes of 5–5.2 in with resolutions of at least 720 × 1280 pixels. Some phones had higher screen resolutions, but this was unlikely to have had any effects since the interface elements were implemented to scale relative to the screen resolution. The Android phones given to the participants already had PyKinetic_DbgOut installed. Participants interacted with the tutor for roughly an hour. Lastly, participants were given an 18-min post-test, which included open-ended questions for comments and suggestions about the tutor. The post-test was given either when time had run out, or when a participant had finished all problems. There were two tests of comparable complexity that were alternatively given as the pre-test for half of the participants. 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 the lectures of the introductory programming course. The pre/post-tests had six questions each and were administered on paper. The tests contained same types of questions from Table 1 (worth one mark each), and additionally a code-writing question (worth five marks). The maximum mark for both tests is ten marks. The participants were not accustomed to doing any programming exercises on paper, because all lab quizzes and assessment in the course are completed using computers. Therefore, the 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 questions had an ideal solution of five LOCs (without any comments), which was the reason for a maximum of five marks on this question. The participants did not receive scores for the problems completed in the tutor.

Findings

We present the results from the 35 participants who completed the study in Table 3. There were no significant differences on any reported pre- and post-test scores in Table 3. The problems in PyKinetic_DbgOut are of different nature to the problems the participants were used to in the course, where they were mostly asked to write code. For that reason, we investigated whether there is a difference between the participants based on their code-writing skills. Before the study, the participants were assessed in a 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 2 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 4 presents the pre- and post-test results for novices and advanced students. We used non-parametric statistical tests to compare the results, as the data was not distributed normally. The novices and advanced students spent comparable time on solving each problem. However, advanced students outperformed novices by completing more problems (U = 35, p = .037), and by getting higher pre/post-test scores. Although the overall pre-test score was significantly different for the two subgroups of students, there was no significant difference on the score for the code-writing question alone. Furthermore, only advanced students improved their score on the code writing question (W = 75, p = .039). On the other hand, we did not find a significant improvement for novices on the code writing question. Lastly, it seemed that output prediction questions were unfavorable for the learning of advanced students, as the advanced students had a significantly higher score than the novices on the pre-test but no significant differences between novices and advanced students in the post-test.

We calculated the Spearman's correlations between the pre- and post-test scores of novice students. Novices had a significant positive correlation on pre- and post-test scores on all questions (r_s = .61, p = .012). However, novices had a significant negative correlation on their pre- and post-test scores on output prediction questions (r_s = − .53, p = .036). Furthermore, we found no significant correlation between the pre- and post-test scores of novices on debugging questions. There were no significant correlations between pre- and post-test scores for advanced students.

Performance in PyKinetic

We present a comparison of the performance measures of novices and advanced participants within PyKinetic_DbgOut in Table 5. As some problems contain multiple

Table 3 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 4 Novices vs. advanced students (*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	.09 (.53)	.30 (.46)	ns
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	$W = 75$, $p = .039$	
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 output questions	ns	$W = 10$, $p = .022$	

activities, we report the performance measures for each type of activities, not per prob- lem. We did not calculate the performance of the participants on the code reading ac- tivity (Dbg_Read), since we only provided two of these questions, and we only allowed participants to attempt these once, as these are true or false questions. The average attempts reported on Table 5 were based on the number of submissions on each activity. We have calculated the normalized gain using two formulas. When the learning gain was positive, we calculated the quotient of gain (post-test score − pre-test score) and (100 − pre-test score). However, when the learning gain was negative, we calculated the quotient of gain and the pre-test score.

The novice students made significantly more attempts in comparison to advanced students on output prediction and code fixing activities, but not for identifying errors. There was no significant difference on the time spent per activity between two groups

Table 5 Novices vs. advanced students performance measures (*ns* denotes not significant)

	Identifying error	Output prediction	Code fixing
Attempts			
Novices	5.33 (2.98)	3.26 (.51)	8.07 (4.02)
Advanced	4.21 (.94)	2.63 (.28)	5.37 (1.52)
	ns	$p = .000$, $U = 256$	$p = .002$, $U = 243$
Time per activity (min)			
Novices	2.01 (0.69)	1.48 (0.63)	1.63 (0.82)
Advanced	2.23 (0.43)	1.30 (0.30)	1.59 (0.47)
	ns	ns	ns
Time per attempt			
Novices	0.87 (0.47)	0.56 (0.27)	0.25 (0.14)
Advanced	0.99 (0.32)	0.56 (0.15)	0.38 (0.14)
	ns	ns	$p = .003$, $U = 65$
Score			
Novices (16)	0.47 (0.16)	0.63 (0.13)	0.25 (0.16)
Advanced (19)	0.60 (0.16)	0.75 (0.09)	0.47 (0.15)
	$p = .020$, $U = 82.5$	$p = .006$, $U = 70.5$	$p = .000$, $U = 40$

of students. The average times per attempt for novices and advanced students were similar apart from the average time on the code fixing activity. The score on each activ- ity was calculated based on the number of correct answers in their first attempt. The maximum score for each activity is 1. For example, in an identifying error activity (Dbg_Ident), if there were three errors to be identified, all three should have been iden- tified correctly on the first attempt to get a score of 1. If only two out of three were correct, a score of 0.67 was given, and a score of 0.33 if only one out of three was cor- rect. However, if the activity only had one required answer, then the score given will be either 0 or 1. The advanced students outperformed the novices on the average score based on their first attempts, on all three activities in Table 5.

We also investigated whether there were any correlations between the scores on the first attempt on an activity and the post-test performance. Table 6 presents significant correla- tions for average scores based on the first attempt on the following activities: identifying error, output prediction, and code fixing. We also present significant correlations between the average scores and the normalized gains. All reported correlations for advanced stu- dents are moderate to high positive correlations. Results revealed that the average scores of advanced students on identifying errors strongly correlated with their scores on output prediction (r_s = 0.68, p = .001). Their scores on identifying errors also showed a moderate positive correlation with their scores on fixing code (r_s = 0.56, p = .014). On the contrary, there were no significant correlations for novices. There were no significant correlations between the average scores on output prediction and average scores on code fixing for both the advanced and novice students. Based on the results presented on Table 6, ad- vanced students benefitted most with identifying error activities as it revealed statistically significant medium to strong positive correlations with their post-test performance on several types of questions: all questions together, code writing, and output prediction questions. Furthermore, we found a medium positive correlation between scores of advanced students on output prediction and their normalized gains on output prediction (r_s = 0.55, p = .016). This was unexpected since advanced students showed a significant decrease with their scores on pre- to post-test on output prediction questions (Table 4).

Table 6 Spearman's correlations between scores, post-test and normalized gains (*ns* denotes not significant)

	Novices (16)	Advanced (19)
Correlation on *score on identifying error* and:		
Normalized gain	ns	r_s = 0.55, p = .016
Normalized gain on output prediction	ns	r_s = 0.58, p = .009
Normalized gain on code writing	ns	r_s = 0.46, p = .048
Post-test code writing	ns	r_s = 0.59, p = .008
Post-test all questions	ns	r_s = 0.66, p = .002
Post-test output prediction	ns	r_s = 0.55, p = .015
Correlation on *score on output prediction* and:		
Score on identifying error	ns	r_s = 0.68, p = .001
Normalized gain on output prediction	ns	r_s = 0.55, p = .016
Post-test output prediction	ns	r_s = 0.55, p = .015
Correlation on *score on code fixing* and:		
Score on identifying error	ns	r_s = 0.56, p = .014

The results shown in Table 6 seem to show that the disparity between novices and advanced students is astoundingly evident. So, we were wondering whether advanced students were just better on their first attempts because of higher prior knowledge. Therefore, we delved deeper and considered average performance on the entire dur- ation of each activity rather than focusing only on first attempts. More specifically, we calculated correlations between the average time per attempt on each activity type with the average normalized gains (Table 7). We found significant high positive correlations between time per attempts on identifying error, output prediction, and code fixing. However, there were clear differences between the correlations found in novices and advanced students. For novices, time per attempt on identifying error was positively correlated with their time per attempt on output prediction (r_s = 0.81, p = .000). Furthermore, their time per attempt on identifying error is also positively correlated with their time per attempt on code fixing (r_s = 0.70, p = .002). Finally, their time per at- tempt on code fixing is also positively correlated with their time per attempt on output prediction (r_s = 0.67, p = .004). For advanced students, we only found one significant correlation between their time per attempt within the three activities which was be- tween identifying error and output prediction (r_s = 0.66, p = .002). Notice that the corre- lations between time per attempt on identifying error and time per attempt on output prediction were both significant for novices and advanced students. However, for nov- ices, we obtained a stronger positive correlation (r_s = 0.81, p = .000) compared to ad- vanced students (r_s = 0.66, p = .002). To sum up, all three activities are positively correlated with each other for novices, but only identifying error and time per attempt was significant for advanced students (Fig. 4).

A similar disparity (like reported values in Table 6) was revealed between novices and advanced students when correlations between time per attempts and normalized gains were calculated (Table 7). However, unlike the values reported in Table 6, we found one significant correlation for novices between time per attempt on output prediction and normalized gain on output prediction (r_s = 0.52, p = .038). On the other hand, for advanced students, similar significant correlations are reported on Tables 6 and 7. However, in Table 7, all significant correlations for advanced students were only on their time per attempt on identifying errors. Furthermore, most of the significant

Table 7 Spearman's correlations between time per attempt and normalized gains (*ns* denotes not significant)

	Novices (16)	Advanced (19)
Correlation on *time per attempt on identifying error* and:		
Time per attempt on output prediction	r_s = 0.81, p = .000	r_s = 0.66, p = .002
Normalized gain on all questions	ns	r_s = 0.50, p = .031
Normalized gain on output prediction	ns	r_s = 0.55, p = .014
Normalized gain on code writing	ns	r_s = 0.61, p = .005
Post-test output prediction	ns	r_s = 0.48, p = .037
Correlation on *time per attempt on output prediction* and:		
Time per attempt on code fixing	r_s = 0.67, p = .004	ns
Normalized gain on output prediction	r_s = 0.52, p = .038	ns
Correlation on *time per attempt on code fixing* and:		
Time per attempt on identifying error	r_s = 0.70, p = .002	ns

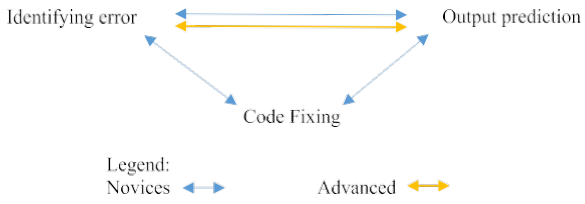


Fig. 4 Relationship between activities for novices vs. advanced students

correlations were on normalized gains apart from one correlation with post-test output prediction (r_s = 0.48, p = .037).

Since we used both scores and time per attempts to calculate correlations, we also calculated correlations between these measures (Table 8). We found related results for advanced students as in Table 7. Table 8 shows significant positive correlations for advanced students, but only between activities of output prediction and identifying errors (r_s = 0.51, p = .026) as visualized in Fig. 4. Furthermore, correlations were found between scores and time per attempt of advanced students for both activities of identifying errors (r_s = 0.71, p = .001) and code fixing (r_s = 0.55, p = .015). For novices, values in Table 8 also show significant correlations between identifying errors and output prediction (r_s = 0.56, p = .025; r_s = 0.50, p = .049), as well as identifying errors and code fixing (r_s = 0.53, p = .036). Moreover, a correlation was found between score and time per attempt on output prediction (r_s = 0.56, p = .024). There were no correlations found between code fixing and output prediction for novices when looking at scores and time per attempt, unlike in Table 7 (r_s = 0.67, p = .004).

Analyses based on demographics

We used the Mann-Whitney U test to check differences between the participants based on demographics, focusing on gender, whether English was their first language, and personal smartphone device. There were 11 advanced and 12 novice male students, and there were 8 advanced and 4 novice female students. When results of male and female students were compared, no measures were significantly different.

We performed the Mann-Whitney U tests between participants who had English as their first language (23) to those who had another language (12). The participants with English as their first language had significantly higher scores on the following: pre-test

Table 8 Spearman's correlations between score and time per attempt (*ns* denotes not significant)

	Novices (16)	Advanced (19)
Correlation on *score on identifying error* and:		
Time per attempt on identifying error	ns	r_s = 0.71, p = .001
Time per attempt on output prediction	r_s = 0.56, p = .025	ns
Time per attempt on code fixing	r_s = 0.53, p = .036	ns
Correlation on *score on output prediction* and:		
Time per attempt on identifying error	r_s = 0.50, p = .049	r_s = 0.51, p = .026
Time per attempt on output prediction	r_s = 0.56, p = .024	ns
Correlation on *score on code fixing* and:		
Time per attempt on code fixing	ns	r_s = 0.55, p = .015

on expected output prediction ($p = .041$, $U = 79.5$), pre-test on all output predic- tions ($p = .037$, $U = 78.5$), post-test scores on code writing ($p = .049$, $U = 81.5$), score on code reading in PyKinetic_DbgOut ($p = .007$, $U = 61$), score on output prediction in PyKinetic_DbgOut ($p = .037$, $U = 78.5$), and score on code fixing in PyKinetic_Dbg- Out ($p = .002$, $U = 53$). Note that all scores within PyKinetic_DbgOut were based on the correctness of their first attempt on each activity. Furthermore, 15 (65%) participants who had English as their first language were advanced students, whereas only (33%) 4 participants had another first language who were also ad- vanced students.

When comparing results of participants based on their personal smartphone devices with Android (16) compared to iOS (16) users, participants who used Android devices had a significantly higher pre-test score on code fixing questions ($p = .002$, $U = 48$), and pre-test score on debugging questions ($p = .002$, $U = 47.5$). The Android users also had a significantly higher score on identifying errors within PyKinetic_DbgOut based on their first attempt ($p = .035$, $U = 72$). Contrary to that, Apple users were significantly faster on their average time per problem with an average of 2.49 min (sd = .52) and average of 2.91 min (sd = .67) for Android users ($p = .023$, $U = 68$). However, Apple users were not significantly faster than Android users on their average time per activity. Furthermore, we examined whether there was a correlation on the length of average use of participants with their smartphones with their normalized gain scores, but we did not find any significant correlations.

When comparing novices with advanced students, their estimated programming ex- perience was statistically comparable. No significant correlations were found between indicated programming experience by participants and their normalized gains. Also, no significant correlations were found between indicated programming experience by par- ticipants and their scores within PyKinetic_DbgOut. However, we found correlations between programming experience as stated by participants and some of their pre-test scores: on code writing ($r_s = 0.36$, $p < .05$).

Comments and suggestions from participants

The post-test included a few questions about PyKinetic_DbgOut. We asked participants an open-ended question on whether they would like to use PyKinetic_DbgOut again and to state their reasons on why they would or would not do so (Q1). Three partici- pants did not respond to the question. Twenty-seven participants (77%) said that they would like to use it again (including 1 participant who gave a condition that he/she will only do so if PyKinetic_DbgOut was improved with more feedback), and 5 participants (14%) said that they would not. Some of the participants who refused to use PyKine- tic_DbgOut again mentioned that they learn better by writing their own code. However, despite their preference to practice on a personal computer, three out of these five par- ticipants showed an improvement in their learning in using PyKinetic_DbgOut based on their scores from pre- to post-test.

The questionnaire also contained an open-ended question asking participants for any comments or suggestions for PyKinetic_DbgOut (Q2). Since we already asked the par- ticipants to state their reasoning (Q1) on wanting (or not) to use PyKinetic_DbgOut again, some participants wrote their comments and suggestions as an answer to (Q1).

For this reason, we combined all the comments and suggestions from both (Q1) and (Q2) and classified them. Some participants wrote multiple comments, hence the rea- son for the numbers below. We classified the comments into following categories:

(C1) PyKinetic_DbgOut was helpful and/or had a good range of content
(C2) PyKinetic_DbgOut is good for debugging and code understanding
(C3) PyKinetic_DbgOut is intuitive and has a good user interface
(C4) Some questions were too difficult.
(C5) PyKinetic_DbgOut is convenient especially when not near a computer

Category C1 contains comments from 23 participants (63%), such as *It was fun and helpful* and "There are very good questions with a good degree of challenge.". Category C2 contains comments from six participants (17%), examples of which are "It is good for extra practice and I know how to point out mistakes and fix them after using it." and "It was good practice at reading a program and understanding what it does." Cat- egory C3 contains comments the usability and user interface of the tutor from four par- ticipants (11%), such as "It is quick to pick up and very intuitive". Three participants provided negative remarks (C4), who found some of the questions in PyKinetic_Dbg- Out too difficult. Lastly, in category C5, there are comments from two participants (6%), who noted that PyKinetic_DbgOut is convenient to use when not near a com- puter. There were some interesting comments. One of the participants commented about the interface: "I felt the app made good use of the limited screen space."

We also combined the suggestions and classified them as follows:

(S1) Feedback needs to be improved
(S2) Include code writing questions
(S3) Usability can be improved particularly in code fixing
(S4) Implement gamification on PyKinetic_DbgOut

The most common suggestion (S1), made by ten participants, was to improve feedback, such as "Give some guide notes when people made mistakes." Seven participants suggested to add some code writing questions (S2), as some of them mentioned that the activities in PyKinetic_DbgOut does not fit their style of learning. Six participants mentioned that the usability of the PyKinetic_DbgOut could be improved particularly for the code fixing activ- ity (S3). Some of them mentioned that they found it easy to accidentally submit a random answer by mistake when fixing LOCs. Finally, three participants suggested to add gamifica- tion (S4), specifically to add a scoring and penalty system, as it was easy to cheat via trial and error, specifically on the code fixing activity. A notable suggestion from one partici- pant was the following: "Potential integration into a lecture environment would be very cool. ". Another participant mentioned a similar suggestion: "The tutor would be good to use for testing students where small simple tests might be appropriate, such as during lectures when the lecturer may want to ask students to interact with the content."

Discussion

We found several differences between participants with lower prior knowledge (novices) and those with higher prior knowledge (advanced). We reported differences

between their learning improvement, performance in PyKinetic_DbgOut, and relationships between their coding skills. We discuss those differences separately in the three subsections ("Learning improvement" to "Relationships between coding skills") to fol- low. In the last subsection "Analysis with demographics," we discuss findings when participants were grouped based their demographics.

Learning improvement

Our results revealed significant correlations on the pre- and post-test scores of novices, implying that novice learners mostly relied on their prior knowledge to answer the post-test. Significant positive correlations were found on the following: pre- and post-test on all questions, and pre- and post-test on output prediction questions. However, for the advanced students, there were no significant correlations on any of the pre- to post-test scores, including pre- and post-test scores on debugging questions. Since most activities in the tutor are mostly composed of debugging activities, we suspect that the reason why only advanced students showed significant improvement in learning was because of the hierarchy of coding skills. As discussed in "Coding skills" section, researchers found that code tracing needs to be mastered before being able to be proficient in code writing (Lopez et al. 2008; Thompson et al. 2008; Harrington and Cheng 2018). Furthermore, Ahmadzadeh et al. (2005) found that debugging code written by someone else requires higher order of knowledge than code writing. Therefore, we have seen evidence of the coding hierarchy of skills in our study. We propose that this may not necessarily mean that the activities were too dif- ficult for novices. Instead, it seems to be that learning with activities on the higher end of hierarchy of programming skills proved to be not beneficial for novices. This is because they are most likely lacking the understanding of the core concepts of pro- gramming, and unless these are learned, students may not be able to benefit. Like what Anderson (1982) mentioned, both declarative and procedural knowledge is needed by students.

We presented more evidence showing that advanced students learned more than novices with PyKinetic_DbgOut, based on the correlations between their scores in the tutor and their post-tests, and their scores in the tutor when correlated with their normalized gains. The correlations revealed that identifying error activities benefitted the advanced students most, as it positively correlated with their post-test scores and normalized gains in various activities. Similar correlations were found when we used scores as a measure, and time per attempt as presented in Tables 6 and 7 respectively. It appeared that advanced students performed worse on pre- to post-test on output prediction questions (Table 4). However, we found a significant positive correlation between their scores on output prediction in the tutor with their normalized gains on output prediction ($r_s = 0.55$, $p = .016$). We propose that the learning benefit was most likely not reflected on the post-test due to the ceiling effect on output prediction problems (pre-test avg. = 97.37%, sd= 7.88). The score of advanced students for iden- tifying errors was also correlated with their normalized gain for all questions and their normalized gain on code writing. These results are consistent with the literature as (Ahmadzadeh et al. 2005) found that most learners with good debugging skills are also advanced programmers.

Performance in PyKinetic_DbgOut

The differences between the performance of novices and advanced students in PyKinetic_DbgOut were interesting. Firstly, one might expect that advanced students will be faster than novices, but we found that there were no significant differences with the average time they spent in the three activities: identifying errors, fixing errors, and out- put prediction. It might have been because, regardless of their abilities, the participants were not accustomed to doing activities in PyKinetic_DbgOut in their course, as they were mostly doing code writing. However, the advanced students outperformed the novices when we calculated their scores based on their first attempts. Furthermore, novices had significantly more attempts on output prediction and code fixing activities. This leads us to believe that some novices might be using trial and error strategies. Furthermore, a trial and error strategy are most likely being utilized more in the code fixing activity as it was revealed that novice students had a significantly lower time per attempt than advanced students in this activity. This result suggests that they com- pleted the code fixing activities with more attempts but with less time, in comparison with the advanced students. Furthermore, although they did not admit to using a trial and error strategy, some participants mentioned in their suggestions that we should im- prove the usability specifically for the code fixing activity. Participants found that the controls in the code fixing activities made it easy to use a trial and error strategy. Therefore, we should improve PyKinetic by providing support for optimal strategies, to prevent learners from guessing.

Relationships between coding skills

We presented correlations between time per attempts and normalized gains for both the novice and advanced students. Bearing in mind that learners were not allowed to skip any activity, the average time per attempt is indicative of the performance of the learners. We showed evidence that the performance of novices on debugging code is positively correlated with their performance in tracing code. Moreover, their perform- ance on code tracing is also positively correlated with their performance on fixing code. Lastly, their performance on fixing code is also positively correlated with their perform- ance in debugging. All these are significant strong correlations, showing evidence that students with lower prior knowledge perform similarly across various coding activities of debugging, tracing and fixing code. However, we did not yield the same result be- tween code tracing and code fixing when we considered both score and time per at- tempt (Table 8). However, we think that this might be related to the possible issue of controls in code fixing activities mentioned by the participants ("Comments and suggestions from participants" section). Therefore, we have reasons to accept that our illustration in Fig. 4 still holds. Our results showing relationships between skills is con- sistent with literature as Harrington and Cheng (2018) also found indications that underachieving students usually have a large gap between coding skills as they are most likely struggling in core programming concepts. Core programming concepts are essen- tial in learning coding skills of debugging, tracing, and fixing. We see further confirm- ation of this when we looked at correlations between the same measures for students with higher prior knowledge. For advanced students, only their time per attempt on de- bugging revealed a significant correlation with code tracing. The performance of

advanced students in identifying errors showed a strong positive correlation with their performance on output prediction, which demonstrated the relationship between debugging and code tracing skills (Table 7). Similarly, a moderate positive correlation was shown between debugging and code tracing when we had taken both scores and time per attempts into consideration (Table 8). Our results demonstrated how various coding skills are interrelated and how it differs between students of varying prior knowledge.

Analysis with demographics

We also performed analysis by grouping the participants based on their demographics. We did not find any significant result when comparing male and female students, most likely because of the lower proportion of female students. When we compared partici- pants based on whether English was their first language, it seemed like participants who had English as their first language performed better than the other students. How- ever, it was probably due to the English group having more advanced learners com- pared to the other group. We also compared the participants based on their personal devices, Android users compared to iOS users. Android users achieved a significantly higher score on identifying errors within PyKinetic_DbgOut but this was most likely because Android users had a significantly higher pre-test score on debugging questions. Lastly, one might expect regardless of the smartphone that they are using, learners who often use their smartphones would learn more in using a mobile tutor such as PyKine- tic_DbgOut. However, we found that was not the case. This might be an indication that PyKinetic_DbgOut can be useful to a learner despite their limited smartphone experi- ence. We also did not find any significant correlations on their average use of their smartphones with their scores within PyKinetic_DbgOut.

Conclusions

We presented a controlled lab study which investigated the effectiveness of activities in PyKinetic: PyKinetic_DbgOut and our findings from the study. We had three research questions for this study. Firstly, we investigated whether the combination of coding activities is effective for learning programming (R1). We found that the group learned from pre- to post-test, but the learning gain was not statistically significant. So, we did a post-hoc division of the participants by their median scores from their course lab test, as we believe this is gives a better representation of their knowledge in programming. Participants who scored less than the median were labeled as novices (16), and the rest as advanced students (19). When we repeated the analysis, we found that only ad- vanced students showed significant learning improvement. Therefore, we found enough evidence to answer our first research question (R1) and found that the combination of coding activities was effective, but primarily for advanced students.

In our second research question (R2), we explored on how the activities affected novices and advanced students. We reported evidence that for novices, all three activities (identifying errors, output prediction, and code fixing) are interrelated with each other. We showed that these activities were positively correlated with each other, showing evidence that if for example a novice learner is proficient at identifying errors, he/she will also be skillful in output prediction, and fixing code. This can also be interpreted that debugging, code tracing, and a subset of code writing skills are interrelated for novices.

We found that although their performance on the activities seem to be equally related, novices did not show any evidence that it was correlated to their learning gains (Table 6). On the other hand, advanced students showed evidence that only their perform- ance on identifying errors and output prediction was positively correlated. Therefore, de- bugging and code tracing skills are interrelated for advanced students. Furthermore, we found evidence that the performance of advanced students in PyKinetic was correlated with their normalized gains. More importantly, activities on debugging (identifying errors) seemed to be most beneficial for advanced students. We presented evidence that their performance on those activities were positively correlated with their normalized gain for all questions, and for specific activities: output prediction, and code writing. Therefore, one of the contributions of the paper is that we confirmed results from the literature that there is a hierarchy in programming skills; evident even when learning via a mobile tutor. Firstly, as mentioned by Anderson (1982), learners need both declarative and procedural knowledge. Specifically, code tracing needs to be learned before writing code (Lopez et al. 2008; Thompson et al. 2008; Harrington and Cheng 2018), and debugging someone else's program requires higher order of skill than code writing (Ahmadzadeh et al. 2005). We reported evidence which supports work of Ahmadzadeh et al. (2005). Our second contribution is that we also confirmed that debugging someone else's programs is beneficial for advanced students. Moreover, another contribution is that we have provided evidence showing that programming can also be learned on a mobile phone, even when (in our study) they only learned with PyKinetic for an hour. We also reported sufficient evidence that the same programming hierarchy of skills applies when learning through a mobile.

We also asked the research question of how we can improve the usability of PyKi- netic (R3). Firstly, we found that the daily smartphone usage of the participants was not correlated with their learning gains in PyKinetic. Furthermore, their smartphone experience was also not correlated with their scores within the app. This could be an indication that the tutor already has a user-friendly interface since we did not find any evidence that their performance and learning gains are dependent on their experience with smartphones. Moreover, we asked the participants a few questions on whether they would use it again and asked for comments and suggestions. Overall, we received a positive response from the participants. Four participants complimented PyKinetic that it was intuitive and easy to use. Furthermore, 63% (22) of the participants said that the tutor was helpful and/or had a good range of content. However, three participants said that they found some of the questions too difficult.

We also found helpful suggestions to improve PyKinetic. The most popular suggestion was that the feedback needs to be improved. We expected this, as we had time limitations for implementation which only allowed us to provide one pre-determined feedback for each activity. The second most popular suggestion from six participants was to include code writing questions. Although work was done in providing code writing exercises on a phone (Mbogo et al. 2016), the evaluation presented qualitative results without investigating learning effects. Therefore, we are not convinced that it can be effective for learning in a smartphone. More work needs to be done in this area. Furthermore, six participants also remarked that the code fixing exercise was easy to cheat, and it was easy to submit an answer by mistake. This was an unexpected suggestion and will be taken it into consideration for improvement. Enhancing the code fixing activity is relevant to the last suggestion which was to implement game features in

PyKinetic. If we add game features, we can for example add limitations on the number of tries when solving a code fixing activity. The findings from this study enabled us to develop an adaptive version of PyKinetic which had personalized problem selection (Fabic et al. 2018).

PyKinetic is designed to improve coding skills in Python, and our findings support this. Our research ultimately aims to fill the gap in the literature and investigates the effectiveness of a combination of several coding tasks within a mobile tutor to provide learners with opportunities to continue enhancing their programming skills while away from computers and classrooms. The limitations of this study include the small set of participants, and limited feedback provided by PyKinetic. Our future work includes repeating the study with more participants. We also plan to conduct a longer study to investigate the effectiveness of PyKinetic as a supplement for a course. Such a study would allow participants to use PyKinetic for as much as they want, since our future goal for PyKinetic is to develop it with low transactional distance and individualized learning activities (LI) (Park, 2011). We aim for PyKinetic to be a supplement to intro- ductory programming courses where learners can independently hone their Python skills anytime and anywhere. Furthermore, we aim to continue developing PyKinetic with a component-skills pedagogical strategy (McArthur et al., 1988) and provide vari- ous tasks which are effectively sequenced to target multiple coding skills.

Abbreviations
LMS: Learning management system; LOC: Line of code; MMLS: Micro-Lecture Mobile Learning System; MOOC: Massive open online course; SE: Self-explanation

Acknowledgements
We would like to take this opportunity to thank our participants and colleagues, especially Dr Richard Lobb, who helped us to administer the study.

Funding
Not applicable.

Availability of data and materials
Data will not be available, due to the constraints posed by the Human Ethics Committee of our University. Only the project team is allowed access to the data.

Authors' contributions
The study presented in this paper is a part of the PhD project conducted by GF. AM is her senior supervisor, and KN is the associate supervisor. AM has been working closely with GF on the design of the experiment, data analysis, and paper writing. KN has contributed to data analyses and paper writing. All authors read and approved the final manuscript.

Competing interests
The authors declare that they have no competing interests.

Publisher's Note

Received: 11 September 2018 Accepted: 23 November 2018
Published online: 18 December 2018

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.
Anderson, J. R. (1982). Acquisition of cognitive skill. *Psychological Review, 89*(4), 369.
Anshari, M., Almunawar, M. N., Shahrill, M., Wicaksono, D. K., & Huda, M. (2017). Smartphones usage in the classrooms: learning aid or interference? *Education and Information Technologies, 22*(6), 3063–3079.
Au, M., Lam, J., & Chan, R. (2015). Social media education: Barriers and critical issues. In *Technology in Education. Transforming educational practices with technology* (pp. 199–205). Springer, Berlin, Heidelberg.
Boticki, I., Barisic, A., Martin, S., & Drljevic, N. (2013). Teaching and learning computer science sorting algorithms with mobile devices: a case study. *Computer Applications in Engineering Education, 21*(S1), E41–E50.

Cass, S., & Parthasaradhi, B. (2018). Interactive: The top programming languages 2018. https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2018. Accessed 4 Sept 2018.

Dukic, Z., Chiu, D. K., & Lo, P. (2015). How useful are smartphones for learning? Perceptions and practices of library and information science students from Hong Kong and Japan. *Library Hi Tech, 33*(4), 545–561.

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. (2017a). Learning with Engaging Activities via a Mobile Python Tutor. In Proc. Int. Conf. Artificial Intelligence in Education (pp. 613–616). Springer, Cham.

Fabic, G. V. F., Mitrovic A., & Neshatian, K. (2017b) 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. (2017c). *Investigating the effectiveness of menu-based self-explanation prompts in a mobile Python tutor*, *Int. Conf. Artificial intelligence in education* (pp. 498–501). Cham: Springer.

Fabic, G. V. F., Mitrovic, A., & Neshatian, K. (2018). Adaptive Problem Selection in a Mobile Python Tutor. In Adjunct Proceedings of the 26th Conference on User Modeling, Adaptation and Personalization (pp. 269–274). ACM.

Gavali, M. Y., Khismatrao, D. S., Gavali, Y. V., & Patil, K. B. (2017). Smartphone, the new learning aid amongst medical students. *Journal of Clinical and Diagnostic Research, 11*(5), JC05.

Grandl, M., Ebner, M., Slany, W., & Janisch, S. (2018). *It's in your pocket: a MOOC about programming for kids and the role of OER in teaching and learning contexts*. Open Education Global Conference. Delft University of Technology.

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, New York.

Harrington, B., & Cheng, N. (2018). *Tracing vs. writing code: beyond the learning hierarchy*. In Proceedings of the 49th ACM Technical Symposium on Computer Science Education (pp. 423–428). ACM, New York.

Hürst, W., Lauer, T., & Nold, E. (2007). A study of algorithm animations on mobile devices. *ACM SIGCSE Bulletin, 39*(1), 160–164.

Hwang, G. J., & Chang, S. C. (2016). Effects of a peer competition-based mobile learning approach on students' affective domain exhibition in social studies courses. *British Journal of Educational Technology, 47*(6), 1217–1231.

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, New York.

Jones, A. C., Scanlon, E., & Clough, G. (2013). Mobile learning: two case studies of supporting inquiry learning in informal and semiformal settings. *Computers & Education, 61*, 21–32.

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, New York.

Kim, H., & Kwon, Y. (2012). Exploring smartphone applications for effective mobile-assisted language learning. *Multimed Assist Lang Learn, 15*(1), 31–57.

Klopfer, E., Sheldon, J., Perry, J., & Chen, V. H. (2012). Ubiquitous games for learning (UbiqGames): Weatherlings, a worked example. *Journal of Computer Assisted Learning, 28*(5), 465–476.

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.

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.

Liu, T. C., Lin, Y. C., Tsai, M. J., & Paas, F. (2012). Split-attention and redundancy effects on mobile learning in physical environments. *Computers & Education, 58*(1), 172–180.

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, New York.

Mbogo, C., Blake, E., & Suleman, H. (2016). *Design and use of static scaffolding techniques to support Java programming on a mobile phone*. In Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education (pp. 314–319). ACM, New York.

McArthur, D., Stasz, C., Hotta, J., Peter, O., & Burdorf, C. (1988). Skill-oriented task sequencing in an intelligent tutor for basic algebra. *Instructional Science, 17*(4), 281–307.

Nakaya, K., & Murota, M. (2013). Development and evaluation of an interactive English conversation learning system with a mobile device using topics based on the life of the learner. *Research & Practice in Technology Enhanced Learning, 8*(1), 65–89.

Nouri, J., Cerratto-Pargman, T., Rossitto, C., & Ramberg, R. (2014). Learning with or without mobile devices? A comparison of traditional school field trips and inquiry-based mobile learning activities. *Research & Practice in Technology Enhanced Learning, 9*(2), 241–262.

O'Malley, C., Vavoula, G., Glew, J. P., Taylor, J., Sharples, M., Lefrere, P., & Waycott, J. (2005). *Guidelines for learning/teaching/tutoring in a mobile environment*. Public deliverable from the MOBILearn project.

Oyelere, S. S., Suhonen, J., Wajiga, G. M., & Sutinen, E. (2018). Design, development, and evaluation of a mobile learning application for computing education. *Education and Information Technologies, 23*(1), 467–495.

Park, Y. (2011). A pedagogical framework for mobile learning: categorizing educational applications of mobile technologies into four types. *International Review of Research in Open and Distributed Learning, 12*(2), 78–102.

Parsons, D., & Haden, P. (2006). *Parson's programming puzzles: a fun and effective learning tool for first programming courses*, *Proc. 8th Australasian Conf. Computing education* (Vol. 52, pp. 157–163).

Pea, R. D. (1986). Language-independent conceptual "bugs" in novice programming. *Educational Computing Research, 2*(1), 25–36.

Pea, R. D., & Maldonado, H. (2006). *WILD for learning: Interacting through new computing devices anytime, anywhere. The Cambridge handbook of the learning sciences* (pp. 852–886). New York: Cambridge University Press.

Perry, J., & Klopfer, E. (2014). UbiqBio: Adoptions and outcomes of mobile biology games in the ecology of school. *Computers in the Schools, 31*(1–2), 43–64.

Roschelle, J., Rafanan, K., Bhanot, R., Estrella, G., Penuel, B., Nussbaum, M., & Claro, S. (2010). Scaffolding group explanation and feedback with handheld technology: impact on students' mathematics learning. *Educational Technology Research and Development, 58*(4), 399–419.

Shih, J. L., Chuang, C. W., & Hwang, G. J. (2010). An inquiry-based mobile learning approach to enhancing social science learning effectiveness. *Journal of Educational Technology & Society, 13*(4), 180–191.

Su, C. H., & Cheng, C. H. (2015). A mobile gamification learning system for improving the learning motivation and achievements. *Journal of Computer Assisted Learning, 31*(3), 268–286.

Sun, D., & Looi, C. K. (2017). Focusing a mobile science learning process: difference in activity participation. *Research and Practice in Technology Enhanced Learning, 12*(1), 3.

Sun, J. C. Y., Chang, K. Y., & Chen, Y. H. (2015). GPS sensor-based mobile learning for English: an exploratory study on self-efficacy, self-regulation and student achievement. *Research and Practice in Technology Enhanced Learning, 10*(1), 23.

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, Sydney.

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

Vinay, K. V., & Vishal, K. (2013). Smartphone applications for medical students and professionals. *Nitte University Journal of Health Science, 3*(1), 59–62.

Vinay, S., Vaseekharan, M., & Mohamedally, D. (2013). RoboRun: a gamification approach to control flow learning for young students with TouchDevelop. arXiv preprint arXiv:1310.0810.

Wang, M., Shen, R., Novak, D., & Pan, X. (2009). The impact of mobile learning on students' learning behaviours and performance: Report from a large blended classroom. *British Journal of Educational Technology, 40*(4), 673–695.

Wang, Y. H. (2016). Could a mobile-assisted learning system support flipped classrooms for classical Chinese learning? *Journal of Computer Assisted Learning, 32*(5), 391–415.

Wen, C., & Zhang, J. (2015). Design of a microlecture mobile learning system based on smartphone and web platforms. *IEEE Transactions on Education, 58*(3), 203–207.

Winslow, L. E. (1996). Programming pedagogy—a psychological overview. *ACM SIGCSE Bulletin, 28*(3), 17–22.

Wong, L. H., Looi, C. K., & Boticki, I. (2017). Improving the design of a mCSCL Chinese character forming game with a distributed scaffolding design framework. *Research and Practice in Technology Enhanced Learning, 12*(1), 27.

Wylie, R., & Chi, M. T. H. (2014). The self-explanation principle in multimedia learning. In R. E. Mayer (Ed.), *The Cambridge handbook of multimedia learning* (pp. 413–432).

Appendix I. ITS2016 Poster Paper

Towards a Mobile Python Tutor: Understanding Differences in Strategies Used by Novices and Experts

Geela Fabic, Antonija Mitrovic and Kourosh Neshatian

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

geela.fabic@pg.canterbury.ac.nz
{tanja.mitrovic,kourosh.neshatian}@canterbury.ac.nz

Abstract. We have recently started developing PyKinetic, a mobile tutor for Python. The first type of activities implemented in the tutor is Parsons problems. We conducted a study to evaluate the interface and usability of PyKinetic and to identify and contrast strategies used by novice learners with those of experts. Great feedback and enthusiasm was received for the prospect of PyKinetic and interesting strategies were revealed from both groups.

Keywords: Mobile Python tutor, Parsons problems, novice/expert differences

1 Experiment Design

Parsons problems or Parsons programming puzzles [1] consist of a set of randomized lines of code which need to be put in the correct order by dragging and dropping, to produce the desired outcome. We present a prototype of PyKinetic, a Python tutor aimed as a complement to traditional lecture and lab-based introductory programming courses. The prototype contains Parsons problems (with and without distractors), but in the future we plan to add additional types of learning activities. As an initial step towards an intelligent tutor for Python, we performed a study with PyKinetic, which had two goals: to evaluate the usability of Parsons problems implemented in PyKinetic, and also to identify and contrast problem-solving strategies of novice and expert users. Our hypothesis was that the experts would outperform novices in speed and effectiveness in solving problems, and use optimal problem-solving strategies. The participants were 8 novice and 5 expert participants, students and tutors from an introductory programming course at the University of Canterbury. The study consisted of individual sessions (one-hour long). The version of PyKinetic used in the study contained 7 topics with 21 problems in total: for each topic, there were two problems with distractors and one without. After providing informed consent, the participants interacted with the tutor. Thinkaloud protocol was used and the screen of the device was recorded as well as verbal comments of the participants. The novices were free to choose problems as they wished, but were asked to attempt at least one problem from each topic. The experts

were asked to attempt the problems that majority of novices attempted, to compare the problem-solving strategies used. At the end, participants filled a questionnaire, which included questions about their background and questions about PyKinetic. A similar study [2] with a mobile tutor using Parsons problems was previously conducted by Karavita et al. [3].

2 Results and Conclusions

As expected, the experts were generally faster in solving problems. A wide range of strategies was observed from both groups. A common strategy was to focus on a particular type of statement and move it i.e. variable declarations, function calls and print statements. A specific version of this strategy was used for problems with functions, when the participants moved the function statement first, followed by the docstring. Half of the novices grouped lines superficially based on indentations. Such a strategy shows reliance on a superficial feature rather than trying to understand the context of code. This strategy however, allowed novices to eliminate distractors and arrange the lines logically, especially with conditional statements. After applying this strategy, the novices either tried to reason about the lines in each group, or used the trial and error strategy. One novice used a unique strategy but eventually abandoned the problem, when he/she deleted all lines, and then retrieved the necessary ones from the trash. The experts used superior strategies in comparison to novices. A common strategy used by experts was to build the solution from top to bottom, which was verbally explained by some. This strategy shows that experts have a model solution, and are working towards matching it. All experts used this strategy, but not always exclusively. One expert alternated between this strategy and another strategy, which consisted of combining syntactically and logically similar LOCs with similar indentations and then logically placing them in the correct order. The strategies used by experts demonstrated a higher level of knowledge. In future work, we will enhance PyKinetic by developing a constraint-based model [4] of the domain. Such domain model will allow the tutor to identify mistakes as well as sub-optimal strategies and take suitable instructional actions.

References

1. Parsons, D., Haden, P.: Parson's Programming Puzzles: a Fun and Effective Learning Tool for First Programming Courses. Proc. 8th Australasian Conf. Computing Education, pp. 157-163, Australian Computer Society (2006)
2. Ihantola, P., Karavirta, V.: Two-Dimensional Parson's Puzzles: the Concept, Tools, and First Observations. Information Technology Education, 10 (2011)
3. Karavirta, V., J. Helminen, Ihantola, P.: A Mobile Learning Application for Parsons Problems with Automatic Feedback. Proc. 12th Koli Calling Int. Conf. Computing Education Research, pp. 11-18 (2012)
4. Mitrovic, A.: Fifteen years of Constraint-Based Tutors: What we have achieved and where we are going. *User Modeling and User-Adapted Interaction*, 22(1-2), 39-72 (2012)

Appendix J. ICCE2016 Workshop Paper

Chen, W. et al. (Eds.) (2016). Proceedings of the 24th International Conference on Computers in Education. India: Asia-Pacific Society for Computers in Education

Investigating Strategies used by Novice and Expert Users to Solve Parsons Problems in a Mobile Python Tutor

Geela FABIC*, 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 PyKinetic, a mobile tutor for Python. The tutor is aimed at novices and meant to be a complement to traditional lectures and labs. The first type of activities implemented in the tutor is Parsons problems, which present code snippets to be ordered by the student to produce the desired output. As a starting point towards an intelligent tutor, we conducted a pilot study to evaluate the interface and usability of PyKinetic, and to identify and contrast strategies used by novice learners with those of experts. Great feedback and enthusiasm was received for the prospect of PyKinetic and interesting strategies were revealed from both groups. The study revealed that experts, as can be expected, outperformed novice users and used superior problem-solving strategies. In future work, we will improve PyKinetic's problems, feedback, activities and extend PyKinetic to provide instruction on optimal problem-solving strategies.

Keywords: Mobile Python tutor; Parsons problems; novice/expert differences; problem-solving strategies

1. Introduction

Learning programming is challenging: the student has to learn the syntax and semantics of the programming language and understand its purpose in context, perform problem solving tasks, logical thinking, as well as develop design skills and strategies (Linn and Dalbey, 1985). Apart from learning programming concepts, the student must also understand concepts behind programming like how code structures are being compiled and translated, as well as how programs are executed. Novice programmers find it rather difficult to grasp these concepts, which might lower their motivation to learn more and to practice. It takes about ten years of experience for a novice programmer to become an expert (Winslow, 1996).

Intelligent Tutoring Systems (ITSs) are knowledge-based systems that simulate the behavior of good human teachers and provide individualized feedback to students (Woolf, 2010). ITSs have been proven effective in supporting student learning in different domains (Koedinger et al., 1997; Heift and Nicholson, 2001; Melis et al., 2001, Mitrovic, 2003, 2012; van Lehn et al., 2005; Weber and Brusilovsky, 2001). When learning a programming language, having access to an ITS is valuable for students to practice, as it is impossible to have a human tutor available at all times. The activities included in the ITS can be designed to focus on increasing engagement, improving students' self-efficacy and motivation for learning. Some students may also find that using an ITS is less socially awkward since it does not involve directly communicating with a human tutor. Hence, this may motivate some students to use an ITS more frequently in their own time and will therefore contribute to opportunities of gaining deeper understanding of the domain.

Python is widely used as a programming language in universities nowadays to teach introductory programming (Guo, 2013). This project aims to develop *PyKinetic* (Fabic, Mitrovic and Neshatian, 2016), a mobile tutor hoping that it would appeal better to a new generation of students, compared to desktop or Web-based educational tools. Apart from the thriving popularity of smart phones and mobile applications, a mobile tutor could potentially target engagement.

We present a prototype of PyKinetic, aimed to be developed as a constraint-based intelligent tutoring system (Ohlsson, 1994). PyKinetic will be a complement to traditional lecture and lab-based

introductory programming courses. The current version of PyKinetic contains only one type of activity – *Parsons problems*. Parsons problems (Parsons and Haden, 2006) are exercises requiring the student to rearrange a given set of randomized lines of code to produce an expected outcome. The prototype currently contains two types of Parsons problems, with and without distractors (extra lines of code). In the future, we plan to add additional types of learning activities.

As an initial step towards an intelligent tutor for Python, we performed a study with PyKinetic, in order to identify problem-solving strategies used by novices and experts. Our hypothesis was that experts would outperform novices in speed and efficiency in solving problems, and use optimal problem-solving strategies. The motivation of the study is to enhance PyKinetic to teach students not only about Python, but also to provide instruction about optimal problem-solving strategies. In the following section, we present research done on Parsons problems, educational systems for Python and evaluations of problem-solving strategies used in solving Parsons problems. We then introduce PyKinetic, followed by the experiment design and the findings. Section 7 discusses the problem-solving strategies used by novices and experts. Lastly, we present our conclusions and compare problem-solving strategies observed in our study with those of other studies.

2. Related Work

Parsons problems were originally designed to promote a fun way for students of an introductory course in Turbo Pascal to improve their skills in syntactic constructs (Parsons and Haden, 2006). These programming activities are suitable for novices, as they contain syntactically correct code that only needs to be put in the right order. A variation includes puzzles with syntactically incorrect or unnecessary lines of code (referred to as *distractors*) which students need to eliminate.

Denny et al. (2008) considered five variants of Parsons. The first two variants contain no distractors, with the difference 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. The next two variants are composed of paired options for each line of code given in a randomized order but the paired options are clearly placed right next to each other. These variants were available with and without scaffolding. The last variant contains pairs of options for each line of code, but these pairs are provided in a random 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, 7 lines of code for the puzzle becomes 14 and having these 14 lines of code in a completely randomized order may be viewed by students to be overwhelming to attempt.

There is no widespread agreement on how Parsons problems compare to other types of exercises typically used in introductory programming courses, such as code reading, tracing, writing and explaining. Code tracing falls into lower categories in Bloom's taxonomy, while code writing requires higher order skills (Thompson et al., 2008). Some researchers find Parsons problems are easier than code tracing (Lopez et al., 2008), while others view Parsons problems to lie in between code tracing and code writing (Lister et al., 2010). Denny et al. (2008) found a moderate positive correlation between scores on Parsons problems and code writing questions, but only a weak correlation between Parsons problems and code tracing questions. Therefore, they suggested that Parsons problems were similar to code writing. There are also 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). Other possible factors could include the interface used (on paper or online), and scaffolding and feedback provided.

There are some Python learning environments developed as mobile applications. An example is Quiz&Learn Python[1], which is a game to test and improve knowledge on Python 2.x programming available on Android and iOS devices. The aim of the game is to answer 20 multi-choice questions and to answer them correctly within one minute for each question as fast as possible to gain more points. There are four help options that can be used only once each for every game: remove two incorrect answers, skip a question, debug the code and stop the timer. Remove incorrect answers removes two incorrect choices out of the four given. Choosing to skip a question, skips the current question to move

[1] http://www.villekaravirta.com/projects/quizlearn-python/

on to the next without answering it. Choosing the help option to debug the code gives the users an access to a debugger which shows a line by line visualization of the execution of the given code snippet. The last option (stop the timer) gives the user unlimited time to answer a question. The game ends when the user answers a question incorrectly, has ran out of time, or has successfully answered all 20 questions. The application is developed using Apache Cordova, Zepto.js, Topcoat, SASS, Node.js and PostgreSQL. Based on observations while using the application, it seems that it is presented more as a game rather than a tutor with game elements. It seems to be more focused on gaming features rather than providing pedagogical aspects to support learning.

There are some educational environments for Parsons problems. Ihantola and Karavirta (2011) present js-parsons, a JavaScript library for developing Parsons problems. The library supports "two dimensional" Parsons problems, which allow students to drag lines of code (LOCs) from a set on the left-hand side of the screen and drop it on the solution space on the right-hand side. The second dimension feature is that indentations are supported and students can change the indentations of the LOCs in their solutions. The library is language independent, and can be used to develop Parsons problems for any programming language. Since js-parsons is a JavaScript library, it can be used to develop Parsons problems on webpages designed for personal computers or mobile webpages for tablets and smartphone devices.

There are limited results in literature about problem-solving strategies used by novices and experts for Parsons problems. Ihantola and Karavirta (2011) report on a small study involving four experts. The study was conducted in JSParsons, a Web environment for Parsons problems built using the js-parsons library. The study presented ten Parsons problems with distractors, which required indentations to be specified. For each problem, the name of the algorithm was provided (such as insertion sort). All experts used the same strategy, starting with method signatures, then proceeding with loops and conditionals, and only at the end dealing with initialization of variables and indentation. Another study conducted with students (Helminen et al., 2012) found that students followed a top-to-bottom strategy for simple Parsons problems. In two problems, the first step in 98.5-99.3% of the solutions was to position the function signature. The same researchers also developed MobileParsons (Karavirta et al., 2012) for iOS and Android mobile devices. The interface presented the problem area on top and the solution area below in portrait mode, and side by side in landscape mode. MobileParsons was further developed to allow limited editing of lines (Ihantola et al., 2013).

3. PyKinetic

PyKinetic is a Python tutor developed using Android SDK to be used on smartphones. The tutor is aimed to serve as an additional resource for novice learners to enhance their Python 3.x programming skills. The prototype currently contains 53 Parsons problems, covering the following topics: *String Manipulation*, *Conditional Statements*, *Lists*, *For Loops*, *While Loops*, *Dictionaries*, *Tuples* and *Data Types*. The learner needs to rearrange given LOCs to form a correct code snippet that would produce the expected result. There are two types of problems, with or without distractors. The number of LOCs per problem ranges from 3 to 16, with a maximum of 5 distractors. The learner can expand any topic to see available problems (Figure 1, left), and select either a specific problem or ask for a random problem. The selected problem is then presented to the learner, together with the problem statement (Figure 1, middle). The learner can view the problem statement at any time during problem solving (by clicking on the "?" icon on the top-right hand corner). Distractors can be removed by tapping on the red X on the right of each LOC. Deleted lines can be retrieved by tapping on the trash icon and selecting desired LOCs (Figure 1, right).

In this prototype, the problem space containing LOCs also serves as the solution space. This is different to other implementations of Parsons problems (Ihantola and Karavirta, 2011; Helminen et al., 2012; Karavirta et al., 2012; Ihantola et al., 2013), where LOCs need to be dragged across from the problem area to the solution area. We decided to combine the problem and solution into a single area in order to maximize the use of space.

There are problems of varying difficulty and complexity in the tutor. Each problem is assigned the complexity level, ranging from 1 (the easiest problems) to 9. Most problems are only code snippets, but some are functions and include function calls.

The learner can submit his/her solution to be checked at any time. The tutor contains correct solutions for problems including alternative solutions, and the student's solution is matched to them. The prototype currently only provides simple feedback, informing the learner that the solution is correct, or specifying that there are still some distractors left, or LOCs missing. Feedback also informs the student whether the order of LOCs is right, when LOCs are selected correctly. We plan to enhance the diagnosis process in the next version by developing a constraint-based model of the domain (Mitrovic, 2012).

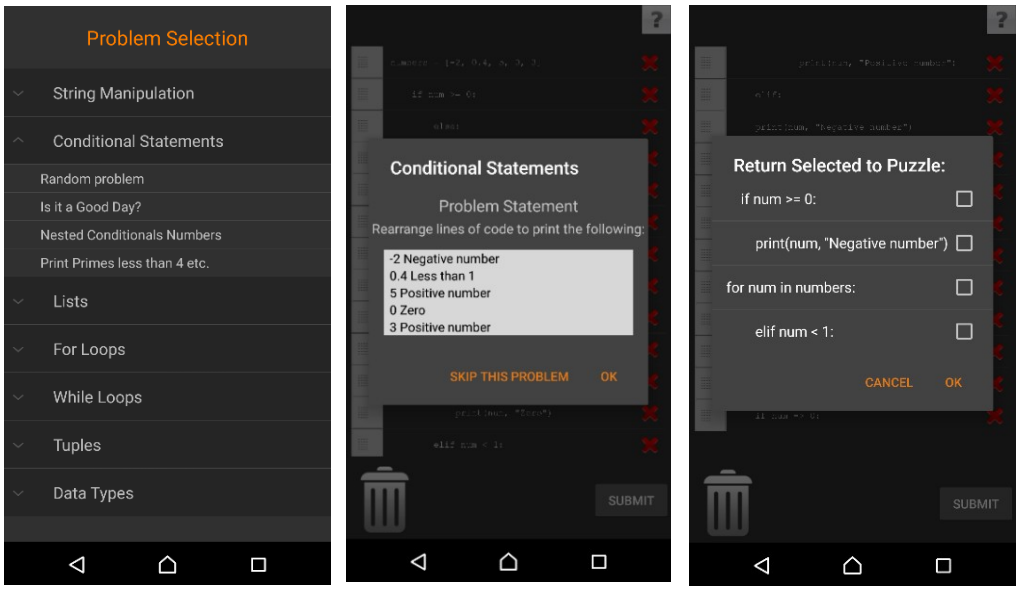


Figure 1. Topic/problem selection screen (left); an example Parsons problem (middle); Retrieving LOCs from trash (right)

4. Experiment Design

The novice participants were 8 volunteers (4 male, 4 female) recruited from an introductory programming course at University of Canterbury. The five expert participants were tutors teaching the same course. The study consisted of individual, one-hour long sessions. The pilot study was conducted in September 2015, by which time the students had learnt about seven topics covered in PyKinetic (problems on Dictionaries were not included in the pilot). The version of PyKinetic used in the study contained 21 problems in total: for each topic, there were two problems with distractors and one without. The problems used in the study had 3 - 16 LOCs, with a maximum of 5 distractors. Four problems were forced to the landscape mode since they contained long LOCs, while the rest were in the portrait mode.

After providing informed consent, the participants interacted with PyKinetic. The novices were free to choose problems as they wished, but were asked to attempt at least one problem from each topic. We used the think-aloud protocol (Ericsson and Simon, 1993), asking the participants to verbalize their thoughts while interacting with the tutor. The screen of the device used for the study was recorded including audio verbal comments of participants. At the end, participants filled a questionnaire, the first part of which included questions about their background, while the second part included multi-choice and open questions about PyKinetic.

We conducted sessions with experts later on, and asked them to attempt the problems that the majority of novice participants attempted, in order to compare the problem-solving strategies used. The experiment design was similar to the setup by (Ihantola and Karavirta, 2011). In their study, they have observed strategies used by experts in solving Parsons problems. Ihantola and Karavirta also conducted a study with novices to identify other strategies (Helminen et al., 2012).

5. General Findings

When asked how much experience the participants had with Python, using the Likert scale from 1 (*Not so experienced*) to 7 (*Highly experienced*), the mean reply of novice participants was 2.12 (sd = 1.25), with only one novice judging his/her experience as 5, and the rest as either 1 or 2. The mean on the same question for experts was 5.4 (sd = 1.34), ranging from 4 to 7.

We eliminated data about problems which the participants only viewed but performed no actions on, and also data about two problems that were found to be buggy. A problem is considered as attempted if the participant made at least one action on it, either by dragging and/or deleting LOCs, viewing the trash, or submitting the solution. A move is defined as the moving of LOCs when attempting a problem. In the way the Parsons problems were setup in the tutor, each move could affect the positions of other LOCs. The analysis was made simpler by counting a move as one regardless of the difference between the starting and ending positions. However, a move is considered as an abandoned move if it was dropped on the same position where it was dragged from.

We used the Mann Whitney U test to analyze similarities and differences between the two groups, with the significance level of 0.05. Table 1 reports the averages (standard deviations are given in parentheses) for the number of abandoned and completed problems. The table also reports the averages for attempted problems: submissions, moves, abandoned moves, time taken, problem complexity, the number of LOCs/distractors, and the number of times problem statement was viewed. The experts have not abandoned any problems, while two novices abandoned two problems each, and two other novices abandoned a single problem each. The experts solved more problems in fewer submissions/moves and in a shorter time, as expected. The only significant difference on the distributions of the two groups was found for the number of submissions per problem ($p = .002$), and there was a marginally significant difference on the number of moves per problem.

Table 1. Overall Results (** denotes significance on the .01 level)

Measure	**Novices**	**Experts**	**U, p**
Abandoned problems	.75 (.89)	0 (0)	ns
Completed problems	10 (3)	12.8 (3.7)	ns
Submissions	2.72 (1.63)	1.29 (.08)	$U = 0, p < .005$**
Moves	13.65 (11.51)	7.3 (1.45)	$U = 8, p = .093$
Abandoned moves	.62 (1.02)	.26 (.17)	ns
Time taken (min)	4.02 (2.6)	2.82 (1.1)	ns
Problem complexity	3.79 (.78)	4.36 (.55)	ns
Distractors	1.66 (.67)	2.03 (.34)	ns
LOCs	8.75 (1.52)	8.87 (1.3)	ns
Problem statement viewed	3.1 (.85)	3.7 (.8)	ns

We also categorized problems by topic (for the seven Python topics used in the study), as well as by the number of LOCs and distractors, and calculated the same measures. Using the number of LOCs, we divided problems into long (11-16 LOCs), medium (7-10 LOCs) and short (3-6 LOCs). The significant and marginally significant differences found are reported in Table 2. The experts were faster in solving problems of most types, apart from While loops, but the only significant difference in time was for Conditionals. The reason why the experts needed more time for the While loop problems is that they attempted more complex problems of this category (in terms of the problem difficulty, and the numbers of distractors and LOCs – all three differences are significant). The experts also attempted more complex problems on Data types, and were significantly faster (in terms of time and the number of submissions) in problems containing many distractors. They needed fewer submissions to complete problems on Lists (marginally significant difference) and also fewer submissions for long problems.

The highest number of errors for both groups was for the problems on Lists. One source of confusion was related to indexing lists (e.g. *my_list[:2:-1]*). All novices and two experts commented

that they were used to using only one colon indexing a list. This was probably one of the reasons for the marginal difference between average submissions for this category.

Both groups were advised that the problems were presented in the increasing order of difficulty. We observed that most novices started with easier problems, while the experts randomly picked a problem from each topic without focusing on their difficulty. There was a significant difference for difficulty and the number of LOCs for short problems. The experts also needed significantly fewer moves for problems with a medium number of distractors (2-3 distractors). In addition, the experts viewed the problem statement significantly more often in the case of problems with few distractors (0 or 1). A potential reason for this difference is because experts use better strategies: many novices used trial and error (as discussed in the following section), while experts are likely to think about the problem more and review its requirements.

Table 2. Results by Problem Category (* denotes significance at the .05 level)

Problem Category	**Measure**	**Novices**	**Experts**	**U, p**
Conditionals	Time	6.1 (2.96)	4.26 (.71)	6, .045*
Lists	Submissions	3.6 (2.98)	1.33 (.33)	7, .065
While Loops	LOCs	5.25(2.74)	10.3 (1.56)	38, .006**
While Loops	Distractors	.75 (.8)	2.7 (.67)	38, .006**
While Loops	Difficulty	1.87 (1.25)	4.6 (.89)	38, .006**
Data Types	Difficulty	3.46 (2)	5.53 (.96)	35, .03*
Long problems	Submissions	2.76 (1.58)	1.18 (.25)	5, .03*
Short problems	LOCs	4.53 (.68)	5.17 (.47)	35.5, .019*
Short problems	Difficulty	1.78 (.97)	2.37 (.44)	34, .045*
Many Distractors	Time	6.76 (2.77)	3.18 (.52)	1, .003**
Many Distractors	Submissions	3.85 (2.35)	1.22 (.22)	.5, .002**
Medium Distractors	Moves	11.6 (9.82)	5.97 (.74)	6.5, .045*
Few Distractors	Distractors	0 (0)	.21 (.04)	40, .002**
Few Distractors	Viewed	1.65 (.49)	2.96 (.95)	38, .006**

6. Questionnaire Responses

Overall, the participants were enthusiastic about the tutor, as seen from the questionnaire responses, summarized in Table 3. The participants from both groups found PyKinetic intuitive, easy to use and fun (the average ratings ranged from 5 to 5.6). Some participants seemed to appreciate the interface and commented: "*It's nice how it pops up showing you what to do*" and "*Oh wow, that's cool how you can like slide them up... that's nice*."

When asked whether they improved their skills by interacting with PyKinetic, the average response from novices was significantly higher than that of experts ($U = .5$, $p = .002$). It is not surprising that the experts' responses to this question are much lower, as the problems were designed for novices. A few novices seemed surprised that they learned something from the tutor: some comments were "*I'm actually learning something here!*", "*Oh cool I didn't know you could do something like that*." and "*I'm learning stuff while doing it so that's always a plus*."

Both groups seemed to perceive the provided problems having the right amount of difficulty (the experts were asked whether problems are at the appropriate level of difficulty for novice learners). The lowest rating was received from the novices about the amount of feedback provided by the tutor (2.88). This was expected, since the prototype only provides simple feedback which is only available upon submission. It is interesting that experts scored the feedback much higher. There was a statistically significant difference on feedback rating ($U = 34$, $p = .045$).

When asked whether they would use the tutor again, seven out of eight novices agreed (the novice who disagreed specified he/she was not interested in learning more about programming). Two experts also stated they would like to use the tutor again since they gained new knowledge from the tutor, specifically on indexing lists. One participant mentioned "*I can really see myself practicing Python with this on the bus or if I'm waiting for someone*." Both groups were also asked to select programming skills they used in the tutor (reading, syntax and structure and/or logical and semantic reasoning skills). Half of the novices responded they used all those skills, while the other novices selected 2/3 skills. All experts responded they used all the skills.

Table 3. Summary of questionnaire responses

Question (1 Lowest to 7 Highest)	**Novices**	**Experts**
Was the tutor's interface intuitive and easy to use?	5.13 (0.83)	5.6 (0.55)
Was the tutor fun to interact with?	5.13 (0.99)	5 (1.41)
Would you say you have learned some new things and/or enhanced your skills by interacting with the tutor?	5.75 (0.89)	2.4 (1.34)
Do you think it is beneficial that this tutor is developed on a mobile platform?	5.25 (1.04)	4.8 (1.92)
Were problem statements clear enough to understand what needed to be done?	4.5 (1.41)	4.8 (1.92)
Please rate the average difficulty of the problems in the tutor.	4.5 (0.53)	4 (0.71)
Do you think there is enough feedback given when attempting a problem?	2.88 (0.99)	4.2 (0.84)

7. Problem-Solving Strategies

We watched the video recordings of the participants' interactions with PyKinetic and manually observed and identified strategies made by the participants. A wide range of strategies was observed, some of which were used by participants in both groups. An example is to focus on a particular type of LOC and move it (referred to as *Selecting a LOC*). The participants usually looked for variable declarations, function calls and print statements, possibly because variable declarations and function calls are normally located somewhere at the beginning of a program, whilst print statements are normally positioned at the end. One participant made a comment along these lines: s/he just started a problem, noticed a print statement and mentioned the following while dragging the LOC in position: "*Print statements at the end*." This strategy was used at least once by each novice. One expert used this strategy. However, it is important to note that this strategy was not used in all problems; it was observed that the participants' strategies changed depending on the nature of the problem and its expected output.

A more specific version of this strategy was used for problems with functions, when the participants moved the function statement first, followed by the docstring. This strategy was very evident in both groups: five novices and four experts used this strategy for all problems that contained functions. The only situations when this strategy was not used were when those statements were already in place (please note that LOCs were presented in a random order), or when the participant was clearly missing the relevant declarative knowledge. The latter was observed only with novices who used sub-optimal strategies (discussed in Section 7.1).

Another strategy used by both novices and experts was to move distractors (except the very obvious ones) to the end of the solution. Some of the statements novices made during this strategy were "*just in case I still need it*" or "*I don't want to delete the other print lines yet just in case I do need them, but I'll put them down the bottom*." Only two experts used this strategy since experts were generally better at eliminating distractors. Having said that, it seemed that the experts were only doing so because they were too focused on their model solutions to deal with distractors immediately.

All of these strategies require domain knowledge: knowing relative position for specific types of statements, or being able to identify distractors. However, the majority of other observed strategies were used exclusively by one group of participants; those strategies clearly show the difference in domain knowledge between novices and experts. We present those strategies in the following subsections.

7.1 Strategies Used by Novices

Half of the novices grouped LOCs on the basis of their indentations. Such a strategy shows lack of knowledge, as novices were relying on a superficial feature rather than trying to understand the meaning of LOCs. The reliance on the indentations as scaffolding was also mentioned by a participant: "*Sometimes with the loops ... the indentations give away a lot and you can just you know ... without having to read much on what they mean*." This strategy allowed novices to eliminate distractors. The strategy was also useful for arranging the LOCs logically, especially with conditional statements. After applying this strategy, the novices either tried to reason about the LOCs in each group, or used the trial and error strategy. One participant mentioned "*Okay let's put all the indentations at the same…*" then tried to read the code, to find the correct lines. Another novice mentioned: "*So I'm like trying to find the systematic way of like sorting it*." Following this, the novice also mentioned "*So now I'm gonna work out which ones would make sense*."

A common strategy used by novices was trial and error. After solving parts of the problem the participant was knowledgeable about, the participant then tried to solve the rest of the problem by exploring possible solutions, which resulted in multiple submissions. For example, the participant would move a single LOC and submit the solution immediately, in order to eliminate wrong solutions. In some of the situations, the novices asked the researcher for help. This strategy was used when the novices were struggling with problems, therefore illustrating lack of knowledge. Additional evidence can be observed from their utterances, such as "*I'm just gonna get to try all of them and figure out why*" and "*This is one of the questions that is probably more complex than my brain ... whether or not I give up ... I don't know*". Three out of eight novices used trial and error, and two other novices commented that they could see that trial and error can be used as a strategy.

One novice used a unique strategy, when he/she deleted all the LOCs, and then retrieved the necessary ones from the trash. The participant eventually abandoned the problem, so this strategy might be due to high cognitive load.

7.2 Strategies Observed in Experts

A common strategy used by experts was to build the solution from top to bottom (referred to as the top-down strategy). For example, some experts mentioned that function statements have to be first, so they looked for this line and moved it first, then the docstring and other LOCs, until the return or print statement. This strategy shows that experts have a model of the solution, and are working towards matching it. All experts used this strategy, but not always exclusively. One expert alternated between this and another strategy, which consisted of combining syntactically and logically similar LOCs with similar indentations and then logically placing them in the correct order (e.g. similar print statements with similar indentations placed at the bottom).

While the experts were searching for LOCs according to their model solution, three of them were at the same time deleting distractors which were syntactically incorrect lines of code. The other two, on the contrary, left such LOCs and deleted them at the end, although it was clear they understood those LOCs were distractors. Generally, the experts were good at identifying distractors.

8. Discussion and Conclusions

We reported on a pilot study with a prototype of a mobile Python tutor which contained Parsons problems. Our primary goal was to investigate problem-solving strategies used by novices and experts. The study was conducted with 8 novice students and 5 experts. The participants were generally enthusiastic about the prospect of using PyKinetic as an additional tool to learn Python, and found the

problems of appropriate nature and complexity. We received good suggestions for further improvement of PyKinetic, such as adding a short tutorial for first-time users, improved feedback and hints on solving problems. The enthusiasm from the participants was encouraging, with seven out of eight novices and two out of five experts interested to use the tutor again.

Feedback received also included suggestions for the interface. Overall, the interface was considered to be intuitive and user-friendly. As mentioned earlier, Parsons problems were presented in PyKinetic in either portrait or landscape mode, which gave us additional insights about the interface. It was observed that in problems presented in the landscape mode, most LOCs were obscured, which seemed to increase extraneous cognitive load for many novices. Some participants commented that the problems in the landscape mode seemed more difficult because the full view of the problem was not available.

We have observed several effective problem-solving strategies used by both novices and experts such as using declarative knowledge to focus on particular LOCs and position them first, and moving distractors to the end of the code. The strategies used by experts demonstrated a higher level of knowledge, as they mostly used the top-down strategy. One expert used an optimal strategy of grouping LOCs with similar indentations, syntax and semantics then logically placing them in their respective positions. We have also observed several strategies when dealing with distractors. Experts appeared to be better in identifying distractors compared to novices.

As mentioned in Section 5, our experiment design was similar with the study conducted by Ihantola and Karavirta (2011). The number of experts in their study were similar to ours. Most experts in our study followed a top-down strategy, solving the problem from the function statement through to the return or print statement. Ihantola and Karavirta reported a similar top-down strategy. However, they have not observed the experts to move all lines perfectly (in the correct order). This is maybe because of the algorithmic nature of their problems compared to ours, which focused on honing basic Python programming skills for novices. Nevertheless, we have confirmed their findings on the top-down strategy observed in experts. This shows that experts solving Parsons puzzles are working towards a mental model solution.

The novices used sub-optimal strategies such as trial and error. None of the novices used the top-down strategy; this contradicts the findings reported by Helminen et al. (2012), where majority of the novices were observed to follow the top-down strategy. The reason for this may be the noticeable difference between the length and complexity of the problems used in their study (five problems with 3-8 LOCs without distractors), compared to our study involving 21 problems with 3-16 LOCs and 0-5 distractors per problem. Helminen et al. (2012) also focused on analyzing only three out of the five problems, which made their data set smaller. However, they have also observed a more specific strategy for Selecting a LOC which was to select a *for* loop or an *if* statement first (Helminen et al., 2012).

Another strategy we have observed for novices was to group LOCs by indentation, which is based on superficial scaffolding feature rather than on code logic and semantics. One expert was observed to have used an optimal variation of the strategy to group LOCs by indentation. The expert demonstrated a strategy of grouping syntactically similar statements with the same indentation while also positioning LOCs in place and removing distractors. Both groups were also observed to have strategies on dealing with distractors.

A limitation of this study is the low number of participants. PyKinetic is still in the early ages of development and several evaluation studies will be designed and conducted using the next versions of the tutor. Based on our observations and feedback received from the study, we plan to improve PyKinetic in various aspects: problem authoring, feedback and activities included in the tutor. The buggy problems discovered in the study have since been fixed, and we have added context for problems. We also aim to develop additional types of activities for the tutor such as Parsons problems with missing keywords, erroneous examples, and predicting output.

As mentioned in Section 1, we plan to extend PyKinetic to provide instruction about optimal problem-solving strategies. For example, the system could offer instruction on specific topics the students are struggling with, or the system could refer the student to other potential sources. The system could also advise students about more effective problem-solving strategies, observed in experts. Lastly, we plan to include support for self-explanation, an important meta-cognitive skill which improves learning outcomes, and also to introduce game elements to maximize engagement (Mayer and Johnson, 2010).

References

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.

Ericsson, K. A., & Simon, H. A. (1993). *Protocol Analysis: Verbal Reports as Data* (Revised Ed). Cambridge: MIT Press.

Fabic, G., Mitrovic, A., & Neshatian, K. (2016). Towards a Mobile Python Tutor: Understanding Differences in Strategies Used by Novices and Experts. In *Proc. 13th Int. Conf. on Intelligent Tutoring Systems,* Zagreb, Croatia, June 7-10, 2016. (Vol. 9684, p. 447). Springer.

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.

Heift, T., & Nicholson, D. (2001). Web delivery of adaptive and interactive language tutoring. *Artificial Intelligence in Education,12(4)*, 310-325.

Helminen, J., Ihantola, P., Karavirta, V., & Malmi, L. (2012). How do students solve parsons programming problems?: an analysis of interaction traces. In *Proc. 9th International computing education researc*h conference (pp. 119-126). ACM.

Ihantola, P., & Karavirta, V. (2011). Two-dimensional parson's puzzles: The concept, tools, and first observations. *Journal of Information Technology Education, 10 (IIP),* 119-132.

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. on Computing Education Research* (pp. 51-58). ACM.

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.

Koedinger, K. R., Anderson, J. R., Hadley, W. H. & Mark, M.A. (1997). Intelligent Tutoring goes to school in the big city. *Artificial Intelligence in Education*, 8(1), 30-43.

Linn, M. C., & Dalbey, J. (1985). Cognitive consequences of programming instruction: Instruction, access, and ability. *Educational Psychologist, 20(4)*, 191-206.

Lister, R., Clear, T., Bouvier, D. J., Carter, P., Eckerdal, A., Jacková, J., ... & Thompson, E. (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, September). Relationships between reading, tracing and writing skills in introductory programming. In *Proc. 4th Int. Workshop on Computing education research* (pp. 101-112). ACM.

Mayer, R. E., & Johnson, C. I. (2010). Adding instructional features that promote learning in a game-like environment. *Journal of Educational Computing Research, 42(3)*, 241-265.

Melis, E., Andres, E., Budenbender, J., Frischauf, A., Goduadze, G., Libbrecht, P., ... & Ullrich, C. (2001). ActiveMath: A generic and adaptive web-based learning environment. *Artificial Intelligence in Education, 12*, 385-407.

Mitrovic, A. (2003). An intelligent SQL tutor on the web. *Artificial Intelligence in Education*, 13(2-4), 173-197.

Mitrovic, A. (2012). Fifteen years of constraint-based tutors: what we have achieved and where we are going. *User Modeling and User-Adapted Interaction*, 22(1-2), 39-72.

Ohlsson, S. (1994). Constraint-based student modeling. In *Student modelling: the key to individualized knowledge-based instruction* (pp. 167-189). Springer Berlin Heidelberg.

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

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.

VanLehn, K., Lynch, C., Schulze, K., Shapiro, J.A., Shelby, R., Taylor, L., Treacy, D., Weinstein, A. & Wintersgill, M. (2005). The Andes Physics Tutoring System: Lessons Learned. *Artificial Intelligence in Education*, 15(1), 147-204.

Weber, G., & Brusilovsky, P. (2001). ELM-ART: An adaptive versatile system for Web-based instruction. *Artificial Intelligence in Education,* 12(4), 351-384.

Winslow, L. E. (1996). Programming pedagogy—a psychological overview. *ACM SIGCSE Bulletin*, 28(3), 17-22.

Woolf, B. P. (2010). *Building intelligent interactive tutors: Student-centered strategies for revolutionizing e-learning*. Morgan Kaufmann.

Appendix K. AIED2017 Poster Paper

Investigating the Effectiveness of Menu-Based Self-Explanation Prompts in a Mobile Python Tutor

Geela Venise Firmalo Fabic, Antonija Mitrovic, Kourosh Neshatian

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

geela.fabic@pg.canterbury.ac.nz
{tanja.mitrovic,kourosh.neshatian}@canterbury.ac.nz

Abstract. *PyKinetic* is a mobile tutor for Python, which offers Parsons problems with incomplete lines of code (LOCs). This paper reports the results of a study in which we investigated the effect of menu-based self-explanation (SE) prompts. Students were asked to self-explain concepts related to incomplete LOCs they have solved. The goals of the study were 1) to investigate whether students are learning with PyKinetic and 2) to determine the effect of SE prompts. The scores of participants have significantly improved from the pre-test to the post-test. There was also a significant difference on the post-test scores of participants from the experimental group compared to the control group. In future work, we aim to add other activities to PyKinetic, and introduce a student model and a pedagogical model for an adaptive version of PyKinetic.

Keywords: Mobile Python tutor, Self-explanation, Parsons problems

1 Introduction

Parsons problems are puzzle-like exercises consisting of a set of randomized lines of code to be rearranged in the correct order to produce a desired outcome [1]. These problems are usually solved by drag and drop actions which make them more suitable for smartphones than program writing exercises. Previous work on Parsons problems in a mobile tutor were reported by Karavirta et al. [2], for Android and iOS.

Self-explanation (SE) is an activity which aims to engage the student in reasoning about elements of a problem which are not directly presented, to promote deeper learning [3]. SE prompts were first introduced as open-ended questions, which encourage learners to think without any set limitations. Other forms of SE prompts have emerged: open-ended, focused, scaffolded, resource-based and menu-based prompts [4]. Johnson and Mayer found that menu-based SE prompts were more effective than open-ended SE prompts in a game-like application about electrical circuits [5]. The authors explained that menu-based SE may have increased the effectiveness of learning because they minimize extraneous cognitive load while fostering germane load.

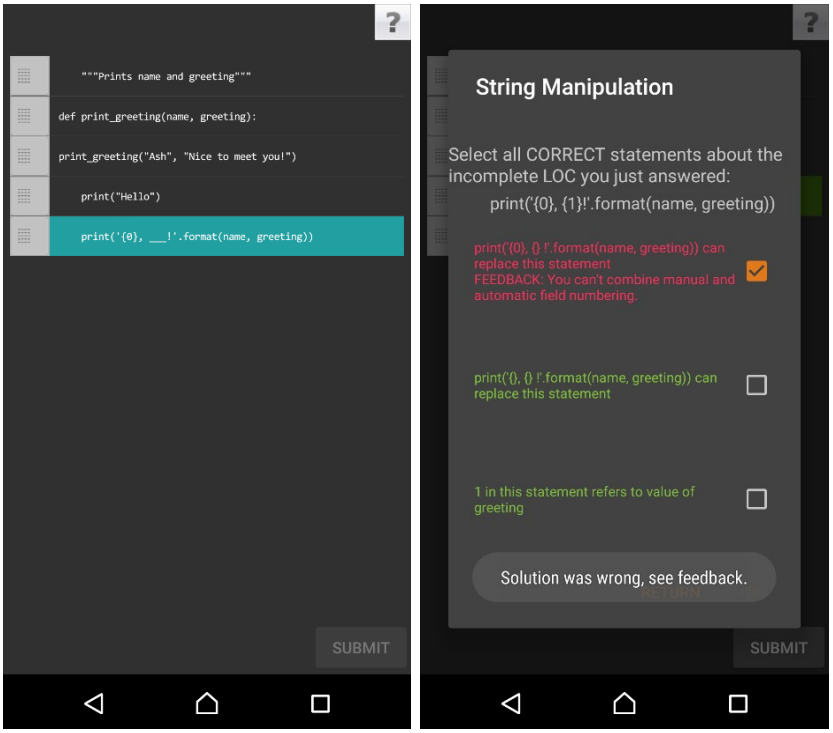


Fig. 1. A Parsons Problem with Incomplete LOCs (left); SE prompt (right)

We present PyKinetic, a mobile Python 3.x tutor for novices aimed as a complement to traditional lectures [6,7]. PyKinetic provides Parsons problems with incomplete LOCs. Parsons Problems in PyKinetic are completed by dragging and dropping single LOCs in the correct order. An incomplete LOC is completed by long-clicking on a LOC to select it (highlighted in blue in Figure 1, left), and tapping the selected LOC to choose the right answer from provided options. All incomplete LOCs must be answered correctly before the solution to the Parsons problem can be submitted. In the situation illustrated in Figure 1 (left), SUBMIT is disabled as the student has not completed the selected line. Once the student completes the line, he/she is given an SE prompt (Figure 1 right). Each SE prompt allows only one attempt. Feedback "Correct! Great job!" is displayed when an SE prompt is answered correctly. If the learner's answer is incorrect, an explanation is shown for all wrong options (right screenshot in Figure 1).

2 Evaluation, Findings and Conclusions

The version of PyKinetic used in the study contained 15 problems, which had between 3 and 16 LOCs, with a maximum of 3 incomplete LOCs. The problems were presented in a fixed order of increasing difficulty. The first two problems were given

as practice. There were 22 SE prompts in total: 14 conceptual questions and 8 procedural questions. There were two conditions in the study: the experimental group received menu-based SE prompts after completing incomplete LOCs, while the control group did not. Our first hypothesis was that all participants, irrespective of the group, would improve their Python skills by interacting with PyKinetic (H1). Secondly, SE prompts would help experimental group participants learn more than control group (H2).

We recruited 83 participants: 70 students enrolled in introductory programming courses from two universities (University of Canterbury and Ateneo de Manila University), as well as 13 high school students from a Digital Technology class from a local high school. The study was approved by the high school and the Human Ethics Committees of both universities. The participants were randomly assigned into the experimental or control group. Each student participated in a group session that lasted for 1.5-2 hours. A 15-minute pre-test was administered first (on paper), after which the participants interacted with PyKinetic. This was followed by a 15-minute post-test. The pre/post-test each had eight questions: six conceptual questions and two procedural questions.

Table 1. Pre-test and Post-test Scores (%)

	Experimental (36)	**Control (40)**	**U, p**
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	
Pre-test Conc.	62.40 (19.05)	63.41 (14.31)	ns
Post-test Conc.	75.71 (16.91)	69.19 (16.71)	U = 550, p = .077
Impr. Conc.	z = -3.221, p = .001	z = -2.37, p = .018	
Pre-test Proc.	71.82 (26.98)	74.42 (19.48)	ns
Post-test Proc.	76.17 (21.42)	74.58 (19.95)	ns
Norm. Gain	14.94 (77.79)	4.82 (63.71)	U = 530, p = .048*
Time/problem (min)	4 (1.57)	3.18 (1.13)	U = 502, p = .023*

We had to eliminate the data related to seven participants because of incomplete logs due to network issues. The analyses presented in the paper were performed on the data collected from the remaining 76 participants. The populations from the three institutions had comparable levels of pre-existing knowledge as there was no significant difference on their pre-test scores. Table 1 reports the pre/post-test scores of the two groups on all questions, and on conceptual/procedural questions separately. We used the paired non-parametric Wilcoxon Signed Ranks test to verify hypothesis H1. Both groups have significantly improved between pre- and post-test overall (the *Improvement* row), and on conceptual questions (the *Impr. Conc.* row), but there was no significant improvement on procedural questions only. These results show that there is

enough evidence to accept our first hypothesis H1, which was that PyKinetic help improve Python skills of the participants.

Table 1 also reports the results of the Mann Whitney U test for checking significant differences between the groups. There was no difference on the pre-test scores, butthe experimental group performed better on the post-test ($p < .05$, $U = 529.5$). There was also a significant difference on the normalized gain ($p < .05$, $U = 530$). Both groups had a positive Cohen's d effect size, but the effect size was higher for the experimental group. These results provide evidence to accept our second hypothesis H2, which was that SE prompts would help experimental group participants learn more than the control group. The experimental group participants spent significantly more time per problem in comparison to the control group, which was expected, as they needed to answer SE prompts ($p < .05$, $U = 502$).

Therefore, both groups improved their performance after interacting with PyKinetic. The participants who received SE prompts performed better on the post-test than the participants who did not self-explain. This result is consistent with work by Johnson and Mayer [5], who also found menu-based SE prompts to be effective in increasing learning in a game-like environment. Our study shows that menu-based SE prompts are also effective on a mobile platform. Future work includes adding more problems, and developing other kinds of activities for PyKinetic. We also endeavor to complement PyKinetic with a student model and a pedagogical model, towards an adaptive version of PyKinetic with personalized problem selection and feedback.

References

1. Parsons, D., & Haden, P. (2006). Parson's programming puzzles: a fun and effective learning tool for first programming courses. In *Proc.8th Australasian Conf.Computing Education, vol. 52* (pp. 157-163). Australian Computer Society, Inc..
2. 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
3. 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.
4. Wylie, R. and Chi, M.T.H. (2014) 17 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.
5. 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.
6. Fabic, G., Mitrovic, A., & Neshatian, K. (2016). 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.
7. 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 PQTEL 2016, pp. 434-444, APSCE.

Appendix L. AIED2017 Doctoral Consortium Track Paper

Learning with Engaging Activities via a Mobile Python Tutor

Geela Venise Firmalo Fabic, Antonija Mitrovic, Kourosh Neshatian

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

geela.fabic@pg.canterbury.ac.nz
{tanja.mitrovic,kourosh.neshatian}@canterbury.ac.nz

Abstract. This paper presents work on a new mobile Python tutor – *PyKinetic*. The tutor is designed to be used by novices, as a complement to traditional labs and lectures. PyKinetic currently contains one type of activity – Parsons problems, which require learners to re-order lines of code to produce a desired output. We present results of studies conducted to evaluate the usability and effectiveness of PyKinetic for learning. The enthusiasm from the participants was encouraging. We have also evaluated menu-based self-explanation prompts in PyKinetic. Results revealed that participants significantly improved their scores from pre- to post-test. Furthermore, participants who self-explained learned more than those who did not. We aim to develop more activities for PyKinetic to support code reading and code writing skills. We also plan to improve the tutor by providing engaging features to maximise learning, and to provide adaptive pedagogical support. Evaluation studies will also be conducted for future versions of PyKinetic.

Keywords: Mobile Python tutor, Parsons problems, self-explanation

1 Introduction

It takes about ten years for one to become an expert programmer [1]. Novice learners find it difficult to grasp programming concepts, which may lower their motivation to learn more. Moreover, most novice programmers of this age are millennials, who usually have short attention spans [2]. It is essential for educators to explore more effective avenues in teaching programming catered to millennial novice programmers.

Python is a popular programming language, widely used nowadays to teach introductory programming, especially in the United States [3]. In New Zealand and Australia, a survey conducted in 2013 on 38 introductory programming courses revealed that majorityof the courses are taught using Python [4]. This project aims to develop a mobile tutor hoping that it would appeal better to new generation of students, compared to desktop or Web-based educational tools. Apart from the booming popularity of smart phones, a mobile tutor could potentially be an effective vessel for engaging activities, which is one of the emphases of our project. The aim is not to focus on the strengths of

a mobile device, but to use it effectively to the best of its advantages for the tutor. The goals of our project are to: (R1) investigate the effectiveness of a mobile tutor with engaging activities to maximize learning, (R2) explore different activities for improving code reading, and code writing skills. Section 2 presents some background, while Section 3 describes PyKinetic and the studies we have performed.

2 Background

Parsons problems [5] are programming exercises which require a given set of randomized lines of code to be rearranged towards producing the expected output, usually by a drag and drop motion. Parsons problems are fitting for a mobile device and for novices since Lines Of Code (LOCs) only need to be rearranged to form the solution. Similarly, Ihantola et al. [7,8] perceived the same insight and have developed Parsons problems for both mobile and web interfaces. Parsons problems have many variations, such as problems with and without scaffolding (curly braces and/or indentations), with and without distractors (extra lines of code) and limited editing of lines [6-9].

Self-explanation (SE), first introduced as open-ended questions, is an activity which requires the student to reason about the problem and generate justifications which are not directly presented by the material to promote deeper learning [10]. Self-explanation has been shown to improve learning outcomes in many domains, such as in database modeling [11], data normalization [12] and electrical circuits [13]. However, some studies like that of Johnson and Mayer [13] show that open-ended SE is not always suitable. They compared open-ended SE prompts to menu-based SE prompts using a computer application teaching electric circuits in a game-like environment. Participants were randomly assigned to an open-based SE group, menu-based SE group and without SE group. Their results revealed that menu-based SE group outperformed both the open-based SE group and without SE group [13].

3 PyKinetic

We have developed a mobile Python tutor, PyKinetic [14], which is designed to be a fun way for novices to learn Python while "on the go", and as a complement to lecture and lab-based courses. PyKinetic is developed using Android SDK and teaches Python 3.x. We have developed PyKinetic with three variants of Parsons problems: regular problems, problems with distractors, and with incomplete LOCs. The first prototype of PyKinetic contained 53 Parsons problems, with 0 up to a maximum of five distractors. The number of LOCs in problems ranges from 3 to 16.

We conducted a pilot study with students enrolled in an introductory programming course in Python and tutors involved in the same course. The pilot study had two goals: to evaluate the usability and the interface of the first prototype, and identify and compare strategies used by novices and experts. As expected, experts outperformed the novices in terms of speed and problem-solving strategies. Experts demonstrated having a mental model of the solution by moving LOCs in the correct order from top to bottom. On the other hand, novices displayed strategies showing lack of knowledge, such as

trial and error, and moving lines based on indentations. Furthermore, enthusiasm from the participants was encouraging, with seven out of eight novices and two out of five experts interested to use the tutor again [15].

The current version of PyKinetic offers incomplete LOCs, and provides menu-based SE prompts after every correctly answered incomplete LOC [16]. An evaluation study was conducted in 2016. We recruited 83 volunteers: 13 high school students from Middleton Grange School and university students (47 from the University of Canterbury, and 23 from the Ateneo de Manila University). All participants were enrolled in an introductory programming course using Python and have had adequate knowledge for the study. Participants were randomly assigned into two groups, with the only difference between the control and experimental condition being that the latter received SE prompts. The study had two hypotheses: (H1) all participants will improve their Python skills by interacting with PyKinetic, and (H2) the experimental group will have higher learning gains than control group. Sessions were conducted in groups which lasted from 1.5-2 hours and had a maximum of 13 participants. The study included a pre-test and post-test completed on paper. Both tests had eight questions: six conceptual questions composed of True/False and multiple choice, and two procedural questions (an output prediction question and a Parsons problem). All actions made during the study in PyKinetic were recorded.

We eliminated data collected from some participants due to unforeseen circumstances. We present results of the data collected from the remaining 76 participants. We used the Mann-Whitney U test to check for a significant difference between the prior knowledge of different populations, and between experimental and control groups. Results showed no difference on pre-test scores between populations and between groups. We used the Wilcoxon Signed Ranks test for measuring learning gains. The results revealed significant improvements for both groups from the pre- to post-test (experimental: $z = -3.315$, $p < .005$; control: $z = -2.45$, $p < .05$). Additionally, there was a significant improvement on conceptual questions ($z = -3.221$, $p < .005$; control: $z = -2.37$, $p < .05$), revealing that hypothesis H1 was supported. The experimental group had significantly higher post-test scores ($U = 529.5$, $p < .05$). Moreover, the normalized gain of the experimental group was also significantly higher ($U = 530$, $p < .05$). We have also calculated the Cohen's d effect size for both groups: experimental $d = .64$, control $d = .34$. Therefore, hypothesis H2 was also supported.

Towards achieving our research goals, other types of activities will be designed and developed, aimed at code reading and code writing skills. Erroneous examples and output prediction exercises are some examples of activities that we will implement in PyKinetic. For addressing our research goal R1, game elements will be introduced targeted to maximize engagement. An adaptive version of PyKinetic is also to be implemented to support enhanced learning experience with personalized problem selection and feedback. The contributions of this project include investigating the effectiveness of a mobile tutor in teaching Python and effective learning activities and pedagogical strategies within a mobile tutor. Several evaluation studies will be conducted with future versions of the tutor.

References

1. Winslow, L.E. (1996) Programming Pedagogy—a Psychological Overview. ACM SIGCSE Bulletin, 28(3), 17-22.
2. 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.
3. Guo, P.J. (2013) Online Python Tutor: Embeddable Web-based Program Visualization for CS Education. Proc. 44th ACM Technical Symposium on Computer Science Education, pp. 579-584, ACM.
4. Mason, R., & Cooper, G. (2014). Introductory Programming Courses in Australia and New Zealand in 2013-trends and reasons. In *Proc. 16th Australasian Computing Education Conference-Volume 148* (pp. 139-147). Australian Computer Society, Inc..
5. Parsons, D., Haden, P. (2006) Parson's Programming Puzzles: a Fun and Effective Learning Tool for First Programming Courses. Proc. 8th Australasian Conf. Computing Education, pp. 157-163, Australian Computer Society.
6. Denny, P., Luxton-Reilly, A., Simon, B. (2008) Evaluating a New Exam Question: Parsons Problems. Proc. 4th Int. Workshop on Computing Education Research, pp. 113-124.
7. Ihantola, P., Karavirta, V. (2011) Two-Dimensional Parson's Puzzles: the Concept, Tools, and First Observations. Information Technology Education, 10.
8. Karavirta, V., J. Helminen, Ihantola, P. (2012) A Mobile Learning Application for Parsons Problems with Automatic Feedback. Proc. 12th Koli Calling Int. Conf. Computing Education Research, pp. 11-18.
9. Ihantola, P., J. Helminen, Karavirta, V. (2013) How to Study Programming on Mobile Touch Devices: Interactive Python Code Exercises. Proc. 13th Koli Calling Int. Conf. Computing Education Research, pp. 51-58.
10. 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.
11. Weerasinghe, A., Mitrovic, A. (2006) Facilitating Deep Learning through Self-Explanation in an Open-ended Domain. *Int. J. of Knowledge-based and Intelligent Engineering Systems* (KES), IOS Press, 10(1), 3-19.
12. Mitrovic, A. (2005) Scaffolding Answer Explanation in a Data Normalization Tutor. *Facta Universitatis, Series Elec. Energ., 18*(2), 151-163.
13. 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.
14. Fabic, G., Mitrovic, A., & Neshatian, K. (2016). Towards a Mobile Python Tutor: Understanding Differences in Strategies Used by Novices and Experts. In Proc. 13th Int. Conf. Intelligent Tutoring Systems, (Vol. 9684, p. 447). Springer.
15. 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 PQTEL 2016, pp. 434-444, APSCE.
16. Fabic, G. V. F., Mitrovic, A., Neshatian, K. (2017) Investigating the effectiveness of menu-based self-explanation prompts in a mobile Python Tutor. Proc. AIED 2017 (in print).

Appendix M. ICCE2017 Paper

Chen, W. et al. (Eds.) (2017). Proceedings of the 25th International Conference on Computers in Education. New Zealand: Asia-Pacific Society for Computers in Education

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 economy[1], 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.

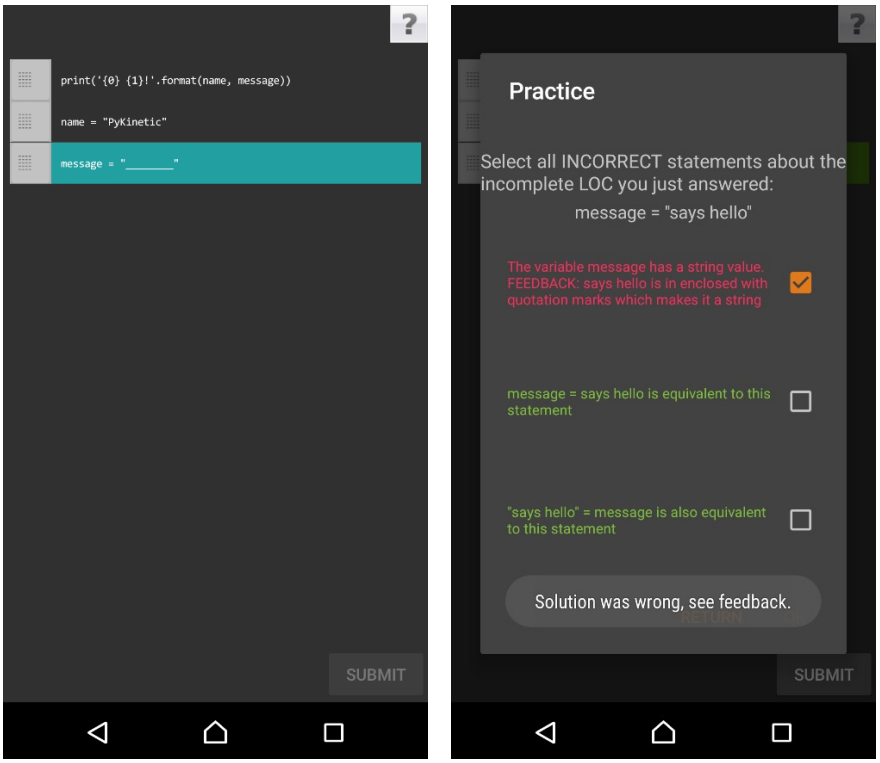


Figure 1. Example of a problem in PyKinetic_IncLOCs (left) and SE prompt (right)

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? (*Yes* or *No)*
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).

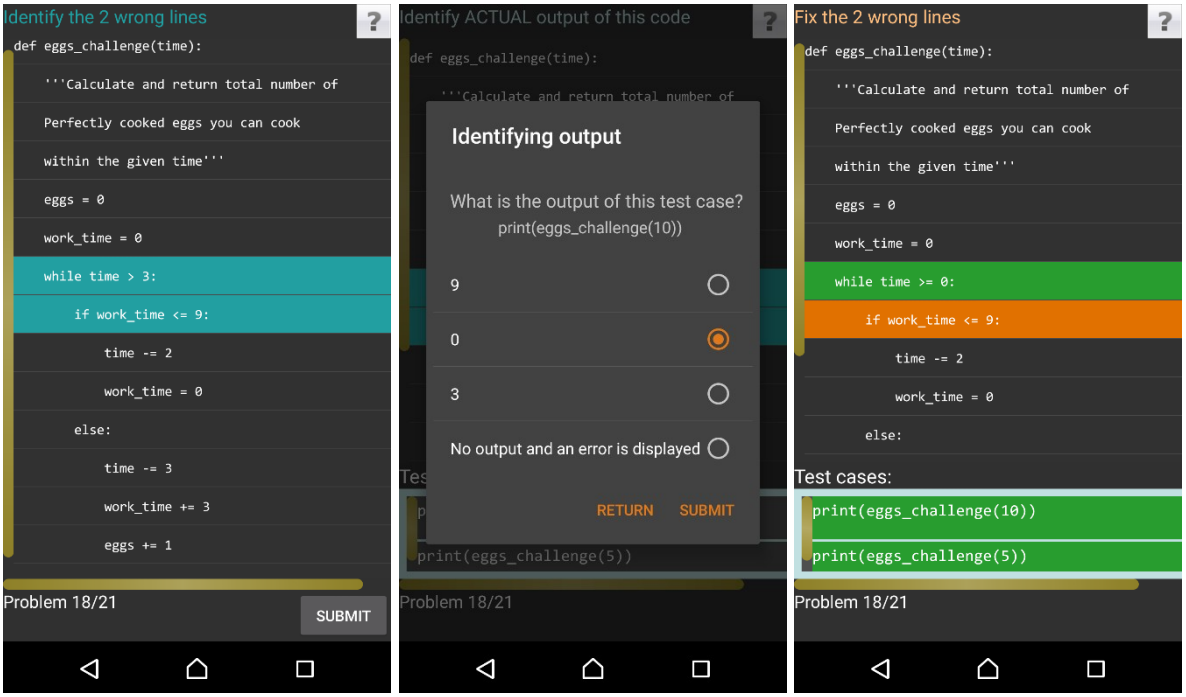


Figure 2. Screenshots from PyKinetic_DbgOut (left Dbg_Ident, middle Out_Act, right Dbg_Fix)

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	Dbg_Read *(2 problems)*	Test cases with actual output	Conditionals	4-6
2	Dbg_Ident *(4 problems)*	Test cases with actual output	String Formatting and Conditionals	4-8
3	Out_Act *(4 problems)*	Test cases	String Formatting, Conditionals, List, Tuples	4-8
4	Dbg_Ident -> Out_Act *(2 problems)*	Test cases	String formatting, Conditionals, List, Tuples, For loops	10
5	Dbg_Ident -> Out_Exp *(2 problems)*	Test cases	String formatting, Conditionals, Lists, For loops	8-9
6	Dbg_Ident -> Dbg_Fix *(3 problems)*	Test cases with expected output	String formatting, Conditionals, Lists, For/While loops, Importing a module	9-11
7	Dbg_Ident -> Out_Act -> Dbg_Fix *(4 problems)*	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.

Appendix N. UMAP2018 Paper

Adaptive Problem Selection in a Mobile Python Tutor

Geela Venise Firmalo Fabic
University of Canterbury, NZ
geela.fabic@pg.canterbury.ac.nz

Antonija Mitrovic
University of Canterbury, NZ
tanja.mitrovic@canterbury.ac.nz

Kourosh Neshatian
University of Canterbury, NZ
kourosh.neshatian@canterbury.ac.nz

ABSTRACT

With the vast majority of students taking introductory programming courses nowadays being millennials, new learning environments are needed. Smartphones are ubiquitous devices primarily used for communication, but are also used for entertainment, services and education. We have developed PyKinetic, a mobile Python tutor for novices aimed as a supplement to other course resources. We present our study on PyKinetic with various activities to target several skills: code tracing, debugging, code understanding and code writing. We compared a version with a fixed sequence of learning activities to an adaptive version, containing the same set of activities but with personalized problem selection. We had two hypotheses for the study: (H1) the combination of activities is effective for learning, and (H2) the adaptive problem selection is beneficial. The results show that PyKinetic is effective for learning, and the adaptive version provides additional benefits for learners.

CCS CONCEPTS

• Applied Computing → Education → Interactive learning environments

KEYWORDS

mobile Python tutor, adaptive problem selection, Parsons problems, debugging, output prediction

ACM Reference format:
G. V. F. Fabic, A. Mitrovic, and K. Neshatian. 2018. In *Proceedings of ACM User Modeling, Adaptation and Personalization conference, Singapore, July 2018 (UMAP 2018)*, x pages. https://doi.org/xxx

1 INTRODUCTION

Programming is difficult to learn but widely sought out by students, as the demands of programmers continue to increase [1]. Nowadays, most novice programmers are millennials who expect fast-paced interactions [2]. We developed PyKinetic, a mobile Python tutor [3, 4]. Our motivation was to better support the new generation of programming students. We first define some terminology used. A *problem* is an exercise which may contain one or more coding activities, where activities are given one at a time. A problem is based on a description, a code snippet, and possibly additional information. An *activity* is one type of coding exercise given within a problem. A *test case* is an executable test which may contain more than one line of code, and consists of parameters and calls to one or more functions.

UMAP 2018, July 2018, Singapore

We conducted several studies with PyKinetic. The version used in the pilot study offered *Parsons problems* with and without distractors (i.e. extra lines of code LOCs) [5]. Parsons problems consist of lines of code (LOCs) that need to be rearranged in the correct order given a description and/or an expected output. We later performed a study using Parsons problems with incomplete LOCs with additional menu-based Self-Explanation (SE) prompts. Our results revealed that Parsons problems with incomplete LOCs were more effective with SE prompts; and that these problems were more beneficial for novice learners [6]. We also conducted a study with debugging and output prediction activities, which revealed that debugging activities were more beneficial for advanced students [7].

In this paper, we present a version of our mobile tutor (referred to as PyKinetic_Fixed) which combines several activities in order to target multiple coding skills. We implemented PyKinetic on a component-skills perspective [8], recognizing that programming requires a set of interrelated skills. Like McArthur et al. [8], we regard sequencing of tasks to be an attribute of the pedagogical strategy in PyKinetic. Furthermore, results from our previous studies encouraged us to develop an adaptive version (PyKinetic_Adaptive) which offers the same activities in PyKinetic_Fixed, but provides adaptive problem selection. Five types of learning activities are offered: regular Parsons problems, Parsons problems with incomplete LOCs, identifying erroneous LOCs, fixing erroneous LOCs, and output prediction. This paper presents a study in which we compared PyKinetic_Fixed and PyKinetic_Adaptive. There were two hypotheses for this study: (H1) the combination of activities is effective for learning, and (H2) the adaptive problem selection will provide additional benefits.

2 RELATED WORK

Novices are slow in solving problems due to lack of declarative and/or procedural knowledge [9]. In programming, declarative knowledge includes the syntax of the programming language and familiarity with code constructs. According to Winslow [10], it takes ten years for a learner to become an expert programmer. Learners often lack mental models, and are unable to translate a problem into manageable tasks [10]. The difficulty in structuring code might be evidence of a deficiency of procedural knowledge. Some students perceive code as series of instructions that are expected to happen when the code is executed [11]. Students have difficulties in comprehending the execution order, predicting the output, debugging and code writing [12].

A study focusing on the debugging patterns of novices [11] found that most learners skilled at debugging were also competent programmers (66%). Furthermore, it was also revealed that only 39% of advanced programmers had good debugging skills. The participants were tested on their debugging

skills by working on erroneous programs. A potential explanation for these findings is that understanding someone else's code requires a higher order of skill. The ability to read and comprehend code which was not written by the learner will expand their capacity as a programmer. We have implemented debugging and output prediction activities in the versions of PyKinetic presented in this paper to help enhance their code tracing and debugging skills.

Parsons problems were originally proposed as an enjoyable way to improve syntactic skills [13]. Since these exercises contain syntactically correct code that only needs to be put in the right order, they are suitable for novices. Variations of Parsons problems include extra LOCs (distractors), or incomplete LOCs which require the learner to provide missing elements. Another variant is two-dimensional Parsons problems [14] which requires learners to specify correct indentations for each LOC in addition to re-ordering the LOCs.

One of the common logical programming bugs classified by Pea [12] is *parallelism*, where an error is caused because a learner thinks that multiple LOCs are executed at the same time. Thus, the learner is completely unaware of the significance of the sequence of the code. Parsons problems may be good practice for novices to grasp the skill of understanding the sequential nature of a program. We include both regular Parsons problems and Parsons problems with incomplete LOCs in our tutor.

More research is encouraged to precisely determine the skills most benefitted by doing Parsons problems. Lopez et al. [15] believe Parsons problems to be simpler than code tracing, while some find that Parsons problems lie between code tracing and code writing depending on their complexity [16]. Code tracing falls into lower categories in Bloom's taxonomy while code writing requires higher order skills [17]. Furthermore, a weak correlation was found between scores on Parsons problems with code tracing questions, and a moderate positive correlation with code writing [18]. Denny et al. [18] propose that Parsons problems are similar to code writing. A study [19] 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 difficulty, variant, and device used. Moreover, there are similar 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 [14]. Other factors that could influence learning are scaffolding and feedback provided.

Recent work by Ericson et al. [20] compared learning gains of three groups of students given the same set of problems but using diverse ways of problem-solving. One group was solving problems by code writing, the other group by code fixing, and the third group by two-dimensional Parsons problems. Authors found that all three groups had statistically comparable learning gains on fixing and writing code. Moreover, the group solving Parsons problems were significantly faster than the other groups, giving evidence that learning with Parsons problems may be more efficient. The learning gains on code writing are consistent with positive correlation of scores on Parsons problems with that of code writing found by Denny et al. [18].

Self-explanation (SE) is a learning activity which aims to promote deeper learning, by producing inference rules and justifications which are not directly presented by the material [21]. 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 electrical circuits [22], probability calculations [23, 24], geometry [25], chemistry [26], database modelling [27] and data normalization [28]. Wylie and Chi [29] review diverse types of SE prompts that have emerged with varying amounts of provided support. There were several studies comparing different forms of self-explanation in different domains. In PyKinetic, we have used *menu-based SE prompts* which provide choices from a menu, instead of traditional open-ended questions [6]. Johnson and Mayer [22] compared menu-based to open-ended SE prompts in the domain of electrical circuits within a game-like environment. The authors found that menu-based SE were more effective than open-ended SE. They have established that it was possibly due to menu-based SE fostering germane and intrinsic load, and minimising extraneous cognitive load at the same time. Thus, Johnson and Mayer [22] concluded that menu-based SE are more suitable for complex environments.

3 TYPES OF PROBLEMS IN PYKINETIC

We defined seven levels of problems covering six Python topics: string manipulation, conditional statements, while loops, for loops, lists, and tuples. Levels on Table 1 are ordered by increasing complexity. Problems on levels 1–4 consist of a single activity each, whereas problems on levels 5–7 are combinations of two or more activities (completed one after another). Problems containing debugging activities contain code with 1–3 incorrect LOCs whereas other problems contain error-free code.

Table 1: Problem Types in PyKinetic

Level	Problem	Additional Information Given
1	Regular Parsons problem (Reg_Pars)	Expected output
2	Output prediction (Out)	Test cases
3	Identify erroneous LOCs (Dbg)	Test cases with actual output
4	Parsons Problem with incomplete LOCs and SE prompt (Pars_Inc)	Expected output
5	Dbg -> Out	Test cases
6	Dbg -> Fix	Test cases with expected output
7	Dbg -> Out -> Fix	Test cases

The simplest problem type (**Level 1**) is a regular Parsons problem (*Reg_Pars*) which only requires the LOCs to be reordered (by dragging and dropping). Correct indentations are provided for all LOCs as scaffolding. Reg_Pars in PyKinetic may contain multiple test cases with different parameters. The learner needs to position the test cases in the correct order to match the expected output. On the learners' first attempt, if the solution is incorrect for the test cases, PyKinetic gives simple feedback, such as "*Test cases must also be reordered to match the expected output.*" If only the rest of the LOCs were ordered wrongly, the feedback is "*Check the order of your solution.*" A hint is provided to the learner on the second incorrect attempt. An example of a hint is "*The counter is decreasing, so the while condition should be?*" On subsequent incorrect attempts,

feedback toggles between simple feedback and a hint. There is one predefined hint for each problem. When the learner successfully rearranges the LOCs in the correct order, PyKinetic provides the following feedback: "*Correct! Great job!*"

On **Level 2** (*Out*), the student is given a problem description, and needs to specify the actual output of the given code for each given test case. Each Out activity contains 1–3 test cases. The example shown in Figure 1 is a combination of activities including an Out activity. In Figure 1 (middle), the learner selects an incorrect answer, and a hint is given for the choice selected. The example in Figure 1 contains 3 test cases, which the learner eventually successfully answered, showing all test cases highlighted in green (Figure 1, right). For each test case, there are three incorrect and one correct choice. If the learner answers incorrectly, he/she cannot select another choice without first closing the output prediction dialog box (shown in Figure 1, middle where submit button is disabled). This was to encourage learners to review the code. Furthermore, only one hint (specifically attached to an incorrect choice) was shown for every output prediction activity. For instance, Figure 1 (middle) shows the learner selected the incorrect choice with detailed feedback of "*The code is currently printing in upper case.*"

Level 3 problems consist of a single 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. An example is shown in Figure 1 (left): at the beginning, the problem description was presented and indicated that the code must print the mocktail contents as 'My mocktail contents are:' followed by printing '*n* mls of the *ingredient name*'; where n was the amount (in mls) included in the drink and with the first letter of the ingredient capitalized. In Figure 1 (left), the learner correctly identified the *n* erroneous LOCs (highlighted in turquoise). If the solution is correct, the student receives positive feedback "*Correct! Great job!*". When the solution is incorrect, the given feedback depends on the circumstances. Firstly, PyKinetic checks whether the learner selected all incorrect LOCs. Feedback is given if the student selected too many or too few incorrect LOCs. If the learner selected exactly *n* lines, but the selections were all incorrect, simple feedback was given "*Sorry, solution was incorrect.*". Moreover, simple feedback of "*Almost there! You are partially correct.*" is given when the selections were partially correct. On a learner's second incorrect try, a hint is given. Subsequent incorrect attempts result in alternating simple feedback and a hint. An example hint is "*Notice the test cases are using print()? Shouldn't the function need a return value?*".

We treat Out problems (level 2) as less complex than Dbg problems (level 3), because the results from our previous studies revealed that Dbg were more beneficial for advanced students [7], and debugging requires a higher order of skill [11].

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_Ins problem contains up to three incomplete lines. An incomplete LOC contains a blank space, which may require one or more keywords. To complete the line, the student needs to select one of the provided options, by tapping between alternatives. The length of the blank space indicates the number of keywords needed. When the learner selects the correct option, PyKinetic highlights the line in green. The learner then gets the SE prompt, which is related to the LOC just completed. We use menu-based SE prompts, which were proven to be effective in our previous study [6]. The learner is not allowed to skip the SE prompt, and can only attempt it once. The Pars_Inc problems target code understanding and code writing skills. It may be argued that Pars_Inc could be easier than Dbg (level 3); however, in previous studies we found that Pars_Inc were more time consuming and required more effort [7].

Level 5 problems are a combination of debugging and output prediction (*Dbg -> Out*). These problems start with the learner identifying incorrect LOCs (Figure 1, left). When this is completed, the learner selects the output for each test case (Figure 1, middle). Learners should consider that the code may not have any output due to the errors that they had just identified in the Dbg activity. Notice one of the choices in Figure 1 (middle) was "*No error but no output is displayed.*" A Dbg -> Out problem is solved once all the output prediction questions are completed.

Dbg -> Fix (**Level 6**) is a combination of Dbg and Fix activities. Each problem of this type starts with the learner identifying incorrect LOCs, which he/she then needs to fix. Code fixing activities are completed in the same manner as in Level 4, by tapping through given choices.

Lastly, the most complex problem type is *Dbg -> Out -> Fix* (**Level 7**), which is a combination of three activities: identifying the erroneous LOCs, predicting the output for the same erroneous code, and finally fixing the errors (Figure 1). The first two activities are the same as Dbg -> Out, but after identifying the output, the learner is also required to fix the erroneous code.

4 EXPERIMENTAL DESIGN

Our hypotheses for this study were: (H1) the problems in PyKinetic will be effective for learning and (H2), the adaptive selection would be superior to a fixed sequence of problems.

Participants: We recruited 32 participants from an introductory programming course at the University of Canterbury, of whom 30 participants completed all phases of the study. The participants learnt about all Python topics covered in PyKinetic before the study. The study was approved by the Human Ethics committee of the University of Canterbury.

Method: The participants were randomly assigned into control and experimental group. The control group had a fixed set of problems, with two problems on each level (Table 2). The experimental group used a version of PyKinetic which provided adaptive problem selection (discussed in Section 5). Both groups had 14 problems to solve (increasing difficulty, each covering up to 5 topics). Learners need to complete one problem before continuing to the next one. The session length was two hours. The participants were first given a brief introduction of the study, then completed a pre-test on lab computers, which had a time limit of 18 minutes. The participants were not allowed to get help and open any integrated development environment to test their answers. Afterwards, an instruction sheet was given for PyKinetic, together with an Android phone with the app installed. For the experimental group, we have entered the pre- test scores into PyKinetic, for the adaptive strategy to select the first problem based on their pre-test scores. The participants interacted with PyKinetic for an hour; a post-test was given afterwards, with the same restrictions as the pre-test.

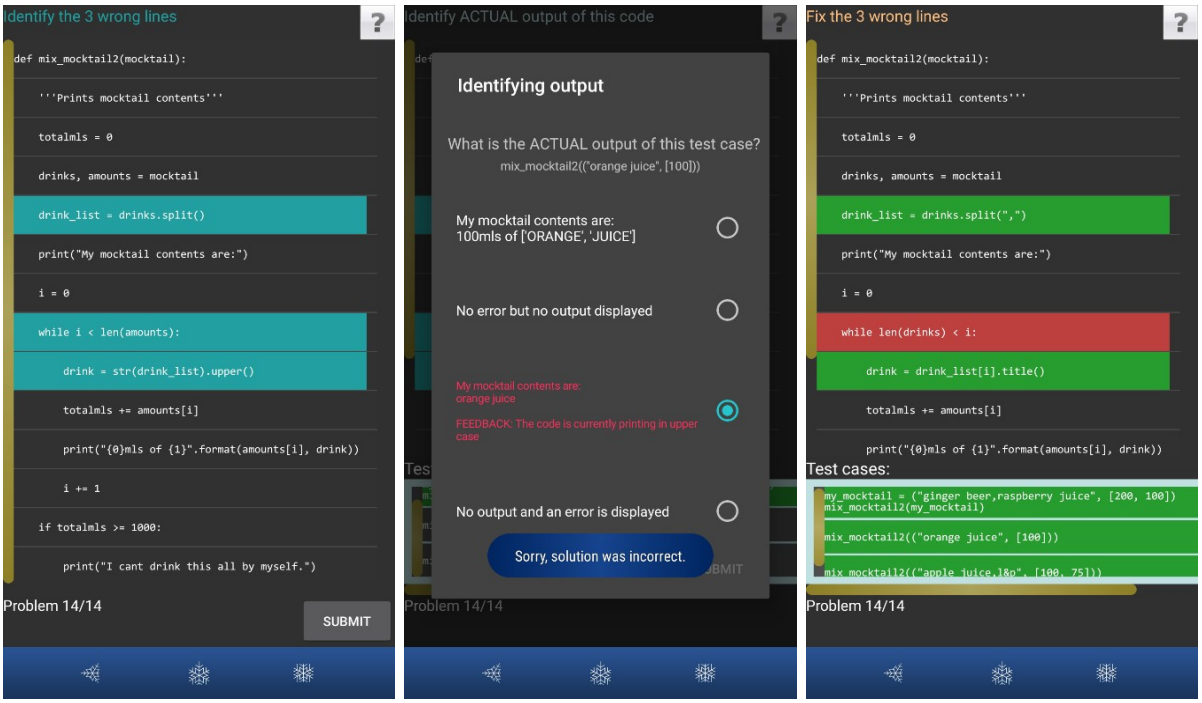


Figure 1: mix_mocktail2 problem, an example of Dbg -> Out -> Fix (left Dbg, middle Out, right Fix)

Table 2: Problems for each step Fixed vs. Adaptive

Step	Fixed (Control)	Adaptive (Experimental)
1	Reg_Pars	Difficulty level 1: Reg_Pars Difficulty level 2: Out Difficulty level 3: Dbg
2		
3	Out	
4		
5	Dbg	
6		
7	Pars_Inc	Difficulty level 1: Pars_Inc Difficulty level 2: Dbg -> Out Difficulty level 3: Dbg -> Fix
8		
9	Dbg -> Out	
10		
11	Dbg -> Fix	Difficulty level 1: Pars_Inc Difficulty level 2: Dbg -> Out Difficulty level 3: Dbg -> Out -> Fix
12		
13	Dbg -> Out -> Fix	
14		

Pre/Post-tests: We had two versions of the test of similar complexity that were alternatively used as the pre-test for half of the participants. Both tests had six questions (Table 3), of the same types as in PyKinetic. However, instead of having two Parsons problems (Reg_Pars and Pars_Inc), there was only one Parsons problem with three extra LOCs (distractors). We refer to this type of Parsons problems as *Pars_Dis*. The problem description for Pars_Dis clearly stated that not all lines were necessary. The three distractors were variants of a single LOC. Pars_Dis problems were used instead of Pars_Inc due to the restrictions in the online quiz system that was used for the pre-/post-tests (the system only allowed fill-in-the-blanks or drag-and-drop activities, but not both at the same time). The Pars_Dis had a drag and drop interface, similar to work by [14], where LOCs were dragged from the problem area onto the solution area.

The code writing question required the learners to type their code without being able to run it. Other questions were given as multiple choice and drop-down list choices (Table 3). Questions 1–3 were worth 1 mark each, questions 4 and 5 were worth 2 marks each, and the code writing exercise worth 5 marks. Pars_Dis was worth 2 marks as it requires more effort. Question 5 was also worth 2 marks since there were two problems in this question (Out and Dbg -> Fix). The Pars_Dis question had 8 LOCs including the function definition statement, docstring and the test case. For the code writing question, the student received a 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.

Table 3: Questions on Pre/Post-tests

Question	Problems	Question Type	Maximum Mark
1	Out	Multiple choice	1
2	Dbg	Multiple choice	1
3	Fix	Multiple choice	1
4	Pars_Dis	Drag and drop	2
5	Out Dbg -> Fix	Drop-down choices	2
6	Code writing	Key in solution	5

Conditions: The control group received problems in fixed order, with two problems from each of the seven levels, as shown in Table 2. On the other hand, the problems given to the experimental group were selected adaptively based on each student's performance. Table 2 shows the problem types available at different steps of the study. In steps 1–7, the participants could receive a regular Parsons problem

(Reg_Pars), predict output (Out) or identify erroneous LOCs (Dbg). Steps 8–14 were composed of more difficult problems (levels 4–7). The fourteen 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.

On each step, the adaptive strategy selected the problem type based on the student's score on the previous 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 between 50% and 74%, and a difficulty level 3 if the score is at least 75%. The calculation for the problem score is discussed in the next section.

5 ADAPTIVE PROBLEM SELECTION

The first problem was selected based on the participant's pre-test score. If the participant scored 49% and below, a Reg_Pars was given. Participants who scored 50% or more were given an Out (level 2) problem; however, if the participant scored higher on Out questions compared to Dbg questions, a Dbg problem was given (level 3). A Dbg problem was also given to participants who performed equally on output prediction and debugging questions and scored more than 75%.

After the first problem, the adaptive strategy uses the performance on the previous step to select the next problem. Problems contain 1-3 activities (as in Table 1). 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 (*AttemptsScore*). We used the data from previous studies to estimate the ideal time needed to solve each activity. The ideal number of attempts for an activity is the minimum number of submissions[1] needed to complete it. TimeScore is calculated as the quotient of ideal time and the actual time the student took. Similarly, AttemptsScore is the quotient of the ideal and the actual number of attempts the student made. Both scores are then combined to compute the score for the activity (Equation 1).

$$ActivityScore = (0.5 * TimeScore + 0.5 * AttemptsScore) - Penalty \quad (1)$$

Penalty is applied if the time taken per attempt is less than 10 seconds (0.17 min). In previous studies, we observed students using the trial-and-error strategy, and in those situations the time per attempts was up to 10 seconds. The penalty is calculated based on the time taken per attempt (*AttemptTime*) and on 10 seconds threshold (Equation 2). The shorter the attempt time, the bigger the penalty.

$$Penalty = 0.17 - AttemptTime \quad (2)$$

[1] By submission, we mean explicit submissions (i.e. clicking the Submit button), and implicit submissions (e.g. finalizing an incomplete line).

6 FINDINGS

Due to the fixed session length, only 13 participants (40%) finished all 14 problems (7 from control and 6 from experimental group). On average, the participants completed 90% of the problems (12.56). We used the Wilcoxon Signed Ranks test to investigate our first hypothesis H1. Taken together, the participants improved their scores significantly between the pre- and post-test (the *Improvement* row in Table 4, p = .031), and also their scores on code writing questions (the *Improvement code writing* row), which is evidence to confirm that PyKinetic supports learning (H1). There was no significant difference on the scores and learning gains for other types of questions.

Table 4: Pre-test and Post-test results

Scores (%)	All (30)	Experimental	Control
Pre-test	73.54 (19.4)	67.22 (19.7)	79.86 (17.5)
Post-test	81.94 (14.8)	76.67 (16.1)	87.22 (11.7)
Improvement	**W = 317 p = .031**	ns	ns
Pre-test Code Writing	67.00 (35.3)	60.00 (38.5)	74.00 (31.6)
Post-test Code Writing	83.00 (25.8)	74.00 (31.1)	92.00 (15.2)
Improvement Code Writing	**W = 177.5 p = .03**	ns	**W = 43 p = .014**

There were no significant differences between the two groups on the number of completed problems and the average time per problem (Table 5). The experimental group participants solved the initial seven problems significantly faster (U=173, p = .01). Furthermore, the experimental group solved significantly more difficult problems (higher average difficulty level) than the control group (U=165.5, p = .004). Figure 2 displays the cumulative product of the average problem difficulty level and the problem-solving score until step 12 (cut-off point set to 67% of participants).

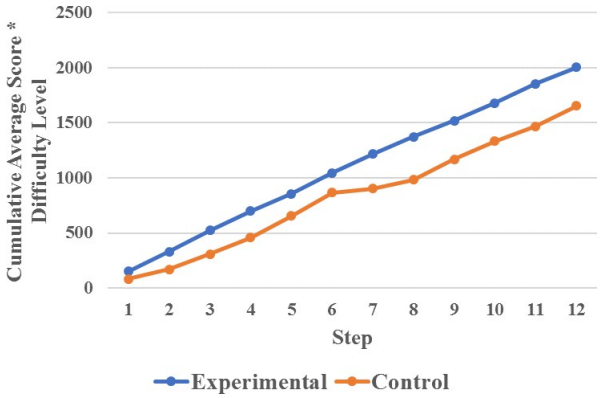
Table 5: Some performance measures

	Exp (15)	Ctrl (15)
Problems Completed	12.67 (1.3)	12.47 (1.8)
Time per Problem	3.98 (1)	4.13 (0.9)
Time 1st Half	2.46 (0.9)	3.13 (0.7)
Time 2nd half	5.56 (1.8)	5.19 (1.4)
Problem Difficulty Level	2.27 (.3)	2.03 (.1)

The control group received two problems of each type, while the experimental group received problems adaptively, based on their performance. The experimental group solved significantly less (avg = 1.13, sd = 1.2) Reg_Pars, the easiest problem type (U = 165, p = 0.029); at the same time, they solved significantly more Dbg problems (avg = 3.2, sd = 1.6), which were more challenging (U = 45, p = 0.004). Note that

these problems were in the first half of the session; therefore, regardless of receiving more difficult problems, the experimental group participants were still significantly faster in completing them. These results support our hypothesis H2.

Figure 2: Cumulative Average Score * Difficulty Level



7 DISCUSSION AND CONCLUSIONS

We presented a study with PyKinetic, a mobile Python tutor, aimed at various programming skills. The results show that students improved their performance after solving problems in PyKinetic for an hour. The adaptive version of PyKinetic provided a further benefit in comparison to the fixed strategy. Although the experimental group received harder problems than the control group, they achieved comparative scores on the post test, and were significantly faster on the first half of the problems. These results provide evidence for our hypotheses.

The limitations of this study include the small set of participants and limited feedback provided by PyKinetic. For each activity, there is only one pre-defined hint. This resulted in situations when feedback was not helpful to students, especially those who were using the trial-and-error strategy. However, it did not prove to be a major impediment for learning. One direction for future improvement of PyKinetic is to add more hints. A good balance of scaffolding and difficulty must be maintained to avoid students from floundering.

PyKinetic is designed to enhance coding skills in Python, and our findings support this. This study has the potential of revealing which activities are most effective for specific coding skills. However, the small set of participants limits our analysis. Our future work includes repeating the study with more participants.

REFERENCES

[1] Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer science education*, *13*(2), 137-172.

[2] Oblinger, D., & Oblinger, J. L. (2005). 2 Is It Age or IT: First Steps Toward Understanding the Net Generation. Educating the next generation. Boulder, Colorado: EDUCAUSE., pp. 12-31.

[3] Fabic, G., Mitrovic, A., & Neshatian, K. (2016). Towards a Mobile Python Tutor: Understanding Differences in Strategies used by Novices and Experts. Proc. 13th Int. Conf. Intelligent Tutoring Systems, (pp. 447-448). Springer.

[4] Fabic, G. V. F., Mitrovic, A., & Neshatian, K. (2017). Learning with Engaging Activities via a Mobile Python Tutor. Proc. Int. Conf. Artificial Intelligence in Education (pp. 613-616). Springer.

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

[6] Fabic, G., Mitrovic, A., Neshatian, K. (2017) Investigating the Effectiveness of Menu-Based Self-Explanation Prompts in a Mobile Python Tutor. Proc. 18th Int. Conf. Artificial Intelligence in Education, Springer LNAI 10331, pp. 498-501

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

[8] McArthur, D., Stasz, C., Hotta, J., Peter, O., & Burdorf, C. (1988). Skill-oriented task sequencing in an intelligent tutor for basic algebra. *Instructional Science*, 17(4), 281-307.

[9] Anderson, J. R. (1982). Acquisition of cognitive skill. *Psychological review*, 89(4), 369.

[10] Winslow, L. E. (1996). Programming pedagogy—a psychological overview. *ACM SIGCSE Bulletin*, 28(3), 17-22.

[11] 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.

[12] Pea, R. D. (1986). Language-independent conceptual "bugs" in novice programming. *Educational Computing Research*, 2(1), 25-36.

[13] Parsons, D., & Haden, P. (2006). Parson's programming puzzles: a fun and effective learning tool for first programming courses. In Proc. 8th Australasian Conf. Computing Education,52 (pp. 157-163).

[14] Ihantola, P., & Karavirta, V. (2011). Two-dimensional Parson's puzzles: The concept, tools, and first observations. *Information Technology Education*, 10.

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

[16] 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.

[17] Thompson, E., Luxton-Reilly, A., Whalley, J. L., Hu, M., & Robbins, P. (2008). Bloom's taxonomy for CS assessment. Proc. 10th Conf. Australasian computing education-Volume 78 (pp. 155-161). Australian Computer Society.

[18] 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.

[19] Morrison, B. B., Margulieux, L. E., Ericson, B., & Guzdial, M. (2016). Subgoals Help Students Solve Parsons Problems. Proc. 47th Tech. Symp. Computing Science Education (pp. 42-47).

[20] Ericson, B. J., Margulieux, L. E., & Rick, J. (2017). Solving Parsons problems versus fixing and writing code. In Proc.17th Koli Calling Int. Conf. Computing Education Research (pp. 20-29). ACM.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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).

[26] 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.

[27] 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.

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

[29] 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, pp. 413–432.

Appendix O. ICCE2018 Paper

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[b] & Kourosh NESHATIAN[c]
[a,b,c] *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 strongpositive 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

Our hypotheses are: (H1) the problems in PyKinetic will be more effective for novices, and (H2) advanced students who used adaptive problem selection will benefit more than advanced students in the control group. 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).

Table 1

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

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

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.

$$ActivityScore = (0.5 * TimeScore + 0.5 * AttemptScore) - Penalty \quad (1)$$

$$Penalty = 0.17 - AttemptTime \quad (2)$$

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 (Equation 1). Furthermore, a penalty is applied if the time per attempt is less than 10 seconds (0.17 min). The penalty is calculated based on the time taken per attempt (*AttemptTime*) and on 10 seconds threshold (Equation 2). The shorter the *AttemptTime*, the bigger the penalty.

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). There were no significant differences between the normalized gains of the novices and advanced students. 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).

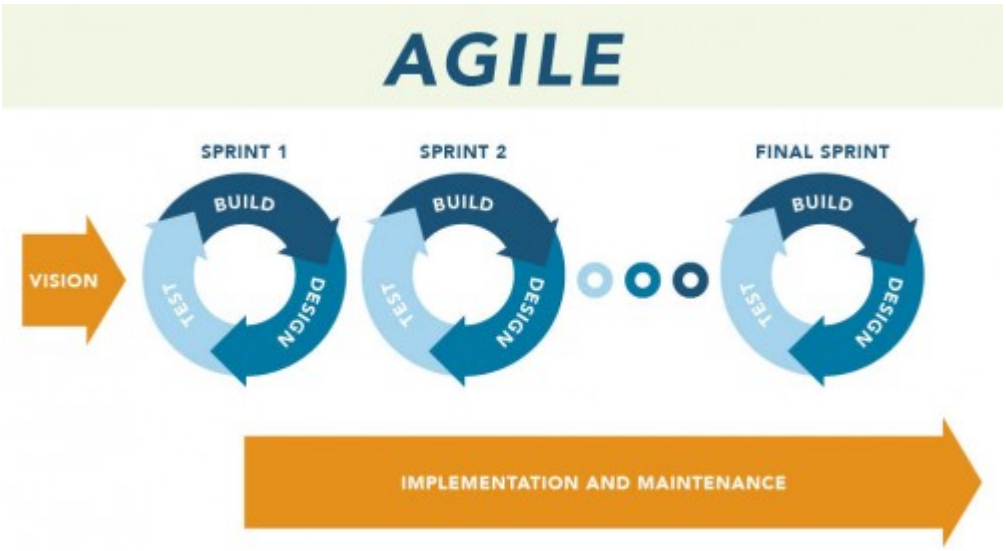
Venables, A., Tan, G., & Lister, R. (2009). A closer look at tracing, explaining and code writing skills in the novice programmer. In *Proc. 5th Int. workshop on Computing education research* (pp. 117-128). ACM.

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.

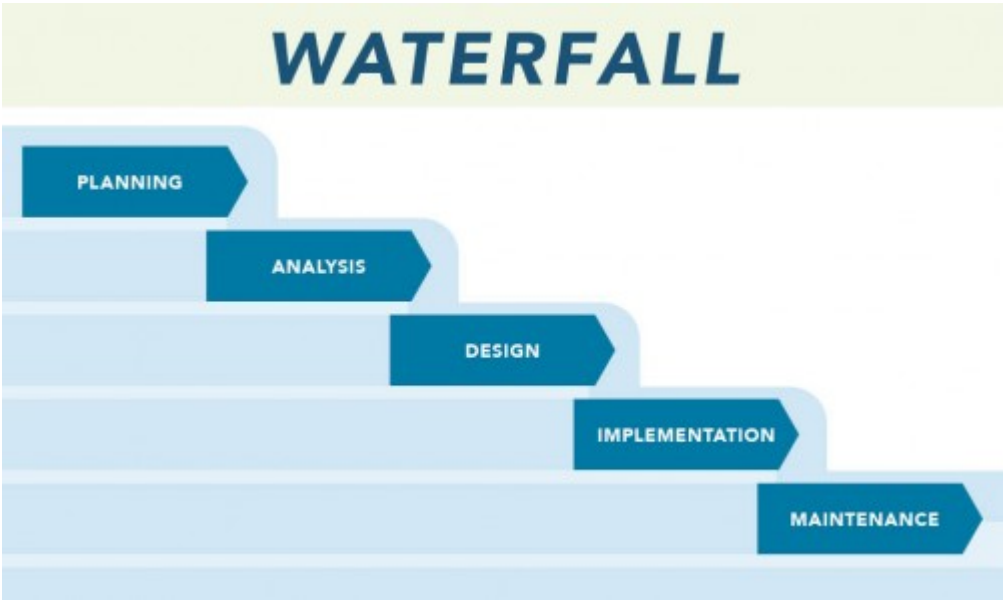
Appendix P. Miscellaneous

1. Agile



[cited 7 June 2019] Available from: https://www.mandsconsulting.com/wp-content/uploads/PM-Agile-500x298.jpg

2. Waterfall



[cited 7 June 2019] Available from: https://www.mandsconsulting.com/wp-content/uploads/PM-Waterfall-500x298.jpg

3. My thesis in cake format (with a Parsons problem), won 2nd best equal in University of Canterbury's "Bake your Thesis" competition.