Multi-Institutional Multi-National Studies of Parsons Problems

Barbara J. Ericson[*]
barbarer@umich.edu
University of Michigan
Ann Arbor, MI, USA

Janice L. Pearce[*]
jan_pearce@berea.edu/jpearce@ashesi.edu.gh
Berea College / Ashesi University
Berea, KY USA / Accra, Ghana

Susan H. Rodger[*]
rodger@cs.duke.edu
Duke University
Durham, NC, USA

Andrew Csizmadia
a.p.csizmadia@newman.ac.uk
Newman University
Birmingham, England, UK

Rita Garcia
rita.garcia@vuw.ac.nz
Victoria University of Wellington
Wellington, New Zealand

Francisco J. Gutierrez
frgutier@dcc.uchile.cl
DCC, University of Chile
Santiago, Chile

Konstantinos Liaskos
k.liaskos@strath.ac.uk
University of Strathclyde
Glasgow, Scotland, UK

Aadarsh Padiyath
aadarsh@umich.edu
University of Michigan
Ann Arbor, MI, USA

Michael James Scott
michael.scott@falmouth.ac.uk
Falmouth University
Cornwall, UK

David H. Smith IV
dhsmith2@illinois.edu
University of Illinois
Urbana, IL, USA

Jayakrishnan M Warriem
jkm@nptel.iitm.ac.in
Indian Institute of Technology Madras
Madras, India

Angela Zavaleta Bernuy
angelazb@cs.toronto.edu
University of Toronto
Toronto, Canada

ABSTRACT

Students are often asked to learn programming by writing code from scratch. However, many novices struggle to write code and get frustrated when their code does not work. Parsons problems can reduce the difficulty of a coding problem by providing mixed-up blocks that the learner assembles in the correct order. They can include distractor blocks with common errors that are not needed in a correct solution, but which may help students learn to recognize and fix errors. There is evidence that students find Parsons problems engaging, useful for learning to program, easier than writing code from scratch, typically faster to solve than writing code from scratch with equivalent learning gains, and useful for learning patterns. Most of the research on Parsons problems prior to this work has been in single institution studies, so we saw a need for replication across multiple contexts.

A 2022 ITiCSE Parsons working group conducted an extensive literature review of Parsons problems, designed several experimental studies for Parsons problems in Python, and created 'study-in-a-box' materials to help instructors run the experimental studies, but the 2022 working group had only sufficient time to pilot two of the studies.

Our 2023 ITiCSE Parsons working group reviewed these studies, revised some studies, created new studies, conducted think-aloud observations on some studies, and ran revised as well as new experimental studies. The think-aloud observations and experimental studies provided evidence for using Parsons problems to help students learn common algorithms such as swap, and the usefulness of distractors in helping students learn to recognize, fix, and avoid common errors. In addition, we review Parsons problem papers published after the 2022 literature review and provide a literature review of multi-national (MIMN) studies conducted in computer science education to better understand the motivations and challenges in performing such MIMN studies.

In summary, this article contributes an analysis of recent Parsons problem research papers, an itemization of considerations for MIMN studies, the results from our MIMN studies of Parsons problems, and a discussion of recent and future directions for MIMN studies of Parsons problems and more generally.

[*]co-leader

CCS CONCEPTS

• **Social and professional topics** → *Computing education.*

KEYWORDS

Parsons Problems, Parsons Puzzles, Parson's Programming Puzzles, Parson's Problems, Parson's Puzzles, Code Puzzles, Multi-institutional study, Multi-national study

ACM Reference Format:
Barbara J. Ericson, Janice L. Pearce, Susan H. Rodger, Andrew Csizmadia, Rita Garcia, Francisco J. Gutierrez, Konstantinos Liaskos, Aadarsh Padiyath, Michael James Scott, David H. Smith IV, Jayakrishnan M Warriem, and Angela Zavaleta Bernuy. 2022. Multi-Institutional Multi-National Studies of Parsons Problems. In *2023 ITiCSE Working Group Reports (ITiCSE-WGR '23), July 7–12, 2023, Turku, Finland*. ACM, New York, NY, USA, 110 pages. https://doi.org/10.1145/3571785.3574127

1 INTRODUCTION

Learning to program is an inherently difficult task which an ever-increasing number of students take on every year [59]. Students of computing are expected to master a programming language's basic syntax and semantics, learn how to utilize these elements to construct programs that accomplish a given task, develop strategies for verifying the correctness of these programs, and debug them when they identify errors or bugs. In the majority of courses, students are expected to acquire these skills through code-writing exercises. In traditional code-writing exercises, where students are required to write their solutions from scratch in a text editor, the error space is quite large, and students' misconceptions make it difficult for them to succeed [85]. Beyond the difficulty of forming a correct solution, students often experience frustration, anxiety, and decreased self-efficacy when repeatedly faced with errors [53]. This motivates the need for tools and approaches for teaching introductory programming that successfully scaffold learning activities for struggling students. Improving success, especially early when learning a new skill, can improve self-efficacy [2].

With these goals in mind, Parsons and Haden [73] introduced "Parsons Programming Puzzles", which have come to be known more simply as "Parsons problems" or "Parsons puzzles". In Parsons problems, students are given mixed-up blocks that they place in order to accomplish a given task. Blocks which are incorrect or not needed to form a solution, called distractors, are designed based on common student errors. Parsons and Haden's goals for Parsons problems were to:

(1) Increase engagement compared to other non-puzzle-like exercises (e.g., syntax drills).
(2) Reduce the problem-solving space.
(3) Allow students to make and correct common errors through the use of distractor blocks.
(4) Model good code for students at the individual block level and the final solution.
(5) Provide immediate feedback in a reduced problem space to speed up debugging.

Subsequent literature reviews by Du et al. [20] and Ericson et al. [27] expanded this set of motivations to include 1) easing the process of identifying student difficulties and 2) reducing cognitive load relative to code writing exercises.

Since their introduction, the adoption of Parsons problems for teaching introductory programming and research has increased [20, 27]. Denny et al. [18] found a strong correlation between performance on Parsons problems and code writing questions in written exams suggesting that both measure a similar skill set. Subsequent studies have found that Parsons problems typically improve learning efficiency compared to fix code and write code exercises while reducing cognitive load [30]. Beyond the learning benefits, most students also report that Parsons problems are an enjoyable and engaging exercise [18, 29].

However, these benefits and positive perceptions are not seen universally. Some students would rather write code from scratch than solve a Parsons problem, and some students experience difficulty when faced with an uncommon solution [43]. Adaptive Parsons problems were created to modify the difficulty of the current or next problem to match the learner's skill level [25, 28]. Work in this area emphasizes both the need for careful consideration when designing Parsons problems for novices and the need for further research into the use of Parsons problems for teaching more advanced students, particularly beyond CS1 [27].

The ITiCSE 2022 working group led by Ericson, Denny, and Prather performed a systematic literature review of prior work on Parsons problems to identify gaps in the literature [27]. In doing so, they identified a need for large-scale, multi-institutional, multinational (MIMN) studies to strengthen the evidence for the benefits of Parsons problems. Furthermore, they identified a need for more research on newer variations of Parsons problems, such as adaptive Parsons [26, 28] and faded Parsons [115]. Finally, most prior work investigated the utility of Parsons problems in a CS1 context using Python or Java, with little work teaching other languages or more advanced concepts.

To fill these gaps, the 2022 working group designed several 'studies-in-a-box' on the Runestone Academy interactive textbook platform [31]. The central goal of these studies was to provide a central location for institutions wishing to use them with all the context and materials needed to run a Parsons problem study. The working group initially planned to run the studies in 2022; however, due to time constraints, studies were only piloted at two universities.

Our ITiCSE 2023 working group built on this work by performing think-aloud studies and running experimental studies created by the previous working group. The 2023 working group also developed additional 'studies-in-a-box' materials, and ran these experimental studies in a MIMN context. In addition, the 2023 working group conducted a literature review on MIMN studies performed in computing education, reflected on our own experiences performing a MIMN study, and updated the literature review on Parsons problems since 2022.

1.1 Related Theories

Experts acquire extensive declarative and procedural knowledge through intensive and sustained periods of study and practice. This affects what they observe and how they organize, represent, interpret, and communicate information within their subject domain [32, 98]. It also affects their ability to recognise patterns within provided data and to generate solutions to problems [38]. Therefore, practice is essential for learning [48]. Practice should include constructive feedback, and scaffolding to challenge and support individual learners to further develop their mastery of a specific domain [12]. Optimized learning occurs when the individual learner remains in their Zone of Proximal Development (ZPD) [112]. Learners without prior experience or familiarity with a particular language's syntax and grammar require guidance and constructive, supportive feedback rather than encountering compiler errors that require deciphering [3, 5, 83]. Parson problems are intended to provide learners with practice and immediate feedback, and a problem's difficulty can be adapted to maintain the learner in their Zone of Proximal Development.

In their systematic literature review of Parsons problems, Ericson et al. [27] identified research and discussed the following related theories associated with the usage of Parsons problems: Cognitive Load Theory, Worked Examples, Self-Efficacy and Metacognition and Self-Regulation. These are all relevant to our work as well.

1.1.1 *Cognitive Load Theory.*

Sweller initially proposed cognitive load theory in the 1980s and since then has refined it [68, 69, 103, 105]. This theory articulates three types of memory: sensory, working, and long-term. Learning occurs when new information is processed in working memory and added to knowledge representations (schemas) in long-term memory [12]. However, working memory has a limited capacity [65]. If its entire capacity is required to process new information, then simultaneously, it cannot be used to modify or create new schemas, which are essential for retaining new information long-term. Therefore, instructional resources should be designed to maximize the cognitive capacity available to create schemas.

The amount of cognitive load that a learner experiences is determined by three components: the difficulty of the material or task they are presented with, the design of the instruction, and strategies adopted for constructing knowledge. The difficulty of the material or task is dependent on learner's prior knowledge, learner's prior experience of addressing a similar task and the task's complexity [20].

Writing code from scratch is regarded as a high cognitive load task that can overwhelm novice programmers [110]. One approach to reduce cognitive load when novice programmers code is to use code completion tasks rather than code creation tasks [5]. Parsons problems, are regarded as a code completion problem, and therefore should have a lower cognitive load for a learner compared to tasks that require a learner to code the solution in its entirety. This is due to firstly, Parsons problems constrain the problem space that learners work with [110] and secondly, the task of creating a coding solution is transformed as the necessity of remembering the syntax of a programming language is reduced.

Distractors are code blocks that either contain errors or are not used in the correct solution to a Parsons problem [99]. Distractors tend to contain common programming errors and misconceptions [99]. The use of distractors can have a negative impact on students' cognitive load [114]. However, distractors provide a level of "desirable difficulties" [99] to challenge students when they are constructing a solution and avoid a trial and error approach.

1.1.2 *Worked Examples.*

An initial goal for Parsons problems was to expose novice learners to an expert's solution for a specific problem (a worked example) [74]. The worked example effect, in which learning is improved by studying worked examples versus solving problems, is one of the most well-known effects predicted by Cognitive Load Theory [1, 16, 104]. Research into worked examples has been undertaken in the fields of mathematics [103, 107, 122] and computer science [66, 76, 121]. Worked examples are used to promote cognitive skills acquisition and are regarded as being effective for initial procedural knowledge development, such as learning to code [90]. An alternative argument for worked examples is that students prefer to learn by studying examples rather than simply reading text [54]. Parsons problems have been used as a type of interleaved practice after worked examples [29, 45, 46].

Unfortunately, students do not always perceive the value of learning from worked examples [24], as learning itself requires cognitive effort [49]. The expertise reversal effect predicts that the worked example effect decreases and can even reverse as a learner develops into an expert [108].

In the future, it may even be possible to generate a personalized Parsons problem, one that is based on a student's incorrect code solution using Large Language Models (LLMs), and then utilise LLMs to generate an explanation if the student successfully completes the coding task but does not understand their solution [15].

1.1.3 *Self-Efficacy.*

Self-efficacy is the individual's internal belief that they can be successful in addressing a given situation or achieve a task [1]. Individuals often dismiss pursuing a particular career path if they believe that they will not be successful in that specific career [1]. Students who encounter errors while coding experience negative emotions that impact their computing self-efficacy [52]. High self-efficacy implies and improves persistence in a field, on the other hand low self-efficacy impacts a student's resilience and odds of continuing with a course or major [23]. Negative experiences in courses tend to affect female students more than male students [22, 61] and may be a contributing factor in female students deciding to leave a course [50]. Similarly, students from underrepresented groups tend to have less prior computing and coding experience [7, 60, 61, 97] which can contribute to initially lower computing self-efficacy and reduce the probability of success.

One argument for the use of Parsons problems is that they provide an opportunity to improve student success on early coding tasks which should increase students' self-efficacy. This in turn could lead to greater student retention and thus could serve to increase the diversity of the computing student population.

1.1.4 *Metacognition and Self-Regulation.*

Another reason for instructors adopting Parsons problems is to provide an opportunity for scaffolding novice programmer metacognition [82]. At the heart of metacognition is thinking about thinking. Self-regulation is a metacognitive skill, that relates to a learner's ability to self-reflect on their own learning processes, understand, and amend it if required. Other key concepts include goal-setting, self-motivation, process-inspection, and self-evaluation. One of the counter arguments of learning programming is that it is difficult for a novice programmer in a short space of time to master the required cognitive skills, such as learning and applying new syntax, thinking computationally, such that metacognitive skills are often underdeveloped or absent from the domain [57, 78, 80].

Recent interventions have been made to increase metacognition with novice programmers [19, 81]. However, only a small number of these interventions focus on the effects of Parsons problems on novice programmer metacognition [36, 37, 79].

1.1.5 *Desirable Difficulty.*

A desirable difficulty may reduce short-term performance but may serve to improve a learner's long-term performance on similar tasks. Desirable difficulties influence the construction of test items and learning activities and are implemented using distractors, however the type of distractors chosen can impact a learner's retrieval process necessary to successfully solve the task [8, 9].

Parsons problems were originally developed to help students acquire competence with structural programming syntax [18] and to occupy the space between reading and writing code. Within a

Parsons problem, distractor blocks are added with the deliberate intention of distracting students with seemly plausible but inaccurate alternatives. These distractors are chosen to either reflect common programming errors that students make in creating code or indicate programming misconceptions that an individual student might hold [20, 44, 73]. Therefore, a Parsons problem with distractors can be used as a formative diagnostic assessment tool to identify a student's programming misconceptions and misapplication of programming concepts. Distractors cannot only differentiate between students' performance in solving a specific task but can also increase the task's learning potential [99]. Additionally, the use of distractors addresses Denny et al.'s [18] concern that students can "game" the Parsons problem without learning. However, [99] recommends distractors should not be used in summative assessments because they significantly increase the problem's completion time without a significant increase in problem discrimination.

A faded Parsons problem, a variation on the original Parsons problem, requires a student to type to fill a blank area of a block to complete the code [35, 114]. As with other types of Parsons problems faded Parsons problems are intended to occur in the space between Parson problems and writing code.

2 PARSONS' LITERATURE REVIEW

An ITiCSE 2022 working group that was focused on Parsons problems wrote an extensive literature review on the use of Parsons problems in computing education research [27]. (We will refer to this as the 2022 WG paper). To collect published papers, they searched on several topic variants of Parsons problems, Parsons puzzles and Parsons programming in (1) ACM Digital Library (Guide to Computing Literature); (2) IEEE Xplore; and (3) Scopus and did forward snowballing on those papers that referenced the first paper on Parsons problems by Parsons and Hayden [73]. Using this process they found over 1000 papers to consider.

To determine whether or not an article was relevant to their review, they utilized three inclusion criteria and six exclusion criteria.

These inclusion criteria were the following, noting that only a single criteria was sufficient for inclusion:

IC1 Contains empirical results on the use of Parsons problems and/or collects data from the use of Parsons problems
IC2 Describes a system/tool for presenting/delivering Parsons problems
IC3 Describes the use of Parsons problems for teaching

The exclusion criteria were the following where fulfilling any one of the following criteria was sufficient for exclusion:

EC1 Article is not written in English
EC2 Article length is less than or equal to 2 pages
EC3 Article is not peer-reviewed
EC4 Article is a thesis or a dissertation
EC5 Parsons problems are not related to the research questions/-goals of the paper, and there is no relevant discussion in the methods or results
EC6 Not related to Parsons problems

For the papers that were identified for inclusion, they recorded information about each paper using a data extraction form to record information in a consistent format. This extraction process was guided by an iteration of several phases that included a training phrase and updating the extraction form based upon their discussions [13]. From the over 1000 papers they initially found, the data extraction process identified 141 papers relevant to Parsons problems for their literature review.

Building on the 2022 WG paper for this paper, we did a search to determine what articles have been published on the use of Parsons problems in the time since that literature review up until August 2023. We used the same search criteria and searched for papers on Parsons problems in the same three venues, namely (1) the ACM Digital library (Guide to Computing Literature); (2) IEEE Xplore; and (3) Scopus. This resulted in between 10 and 28 papers for each of the three searches. These results included a few papers already considered in the 2022 paper, so these were removed. With the remaining papers, we then used the same three inclusion and six exclusion criteria as were used by the 2022 ITiCSE working group. As a result, we identified 20 papers related to Parsons problems in computing education that were published since the 2022 WG paper that had not been listed or analyzed by the 2022 ITiCSE Working Group. The new papers that we identified are listed in Table 3 and Table 4.

In this section, we first review the category analysis and tags in the 2022 WG paper and then describe our findings on the subsequently published papers.

In the 2022 WG paper [27], category analysis with category tagging was used to identify and categorize research question themes in the Parsons problems literature. Two researchers worked together on defining and tagging the papers, reaching agreement on the categories as well as the tags. They identified 23 research themes with each theme in more than one article. Each article was tagged resulting in each article being tagged with between one and seven tags.

For this 2023 WG paper, the same two researchers from the 2022 WG paper worked together to categorize and tag the twenty new papers found, using the same categories from the 2022 WG paper. After independently tagging the new papers, these two researchers revisited each of the papers and came to consensus on all tags for each paper.

The categories found in these new Parsons problem papers are summarized in Table 1 where they are listed in descending order of occurrence. Here are some observations on the new tags:

(1) The top two tags in the identified 2023 papers are the same two top research question themes identified in the 2022 WG paper. Learning Programming (LP) was the top theme in 17 papers and Research Study Parsons-Focused (RSPF) was the second most common theme in 14 papers.
(2) Cognitive Load (CL), which occurred in 7 papers, and Interventive Scaffolding (IS), which occurred in 6 papers, were the next two most common research themes. They occurred in positions 10 and 11 in the 2022 papers, which clearly indicates that proportionally more work has recently been done in these areas.
(3) Research Study but Not Parsons-Focused (RSNPF) and Instructor Perceptions (IP) were the next two most common research themes, which both occurred in four papers. RSNPF was the third most common tag in the 2022 paper.
(4) The remaining categories were all found only 1 or 2 times.

Tags	Tag Description	Count
LP	Learning Programming	17
RSPF	Research Study Parsons-Focused	14
CL	Cognitive Load	7
IS	Interventive Scaffolding	6
IP	Instructor Perceptions	4
RSNPF	Research Study but Not Parsons-Focused	4
PPSS	Parsons to Teach Problem Solving Strategies	2
SP	Student Perception	2
CSP	Collaboratively Solving Problems	1
EA	Evolutionary Algorithms	1
GPP	Generating Parsons Problems	1
KT	Knowledge Transfer	1
LR	Literature Review	1
PSSP	Problem Solving Solution Path	1
SAS	Skill Acquisition Sequence	1
SE	Student Engagement	1

Table 1: An overview of the new Parsons problems paper research themes ordered by decreasing counts.

Conference/Journal	Count
ACE 2022	1
ACE 2023	2
ACM TOCE 2022	1
CHI EA 2023	1
EIT 2022	1
GECCO 2022 Companion	1
ICCE 2021	1
ICCSE 2022	1
ICER 2022	1
ITiCSE 2022	2
ITiCSE 2023	3
KOLI 2022	1
SIGCSE TS 2023	3
SIEE 2022	1

Table 2: Conference where Parsons problem were published for this literature review

(5) There were seven tags by the 2022 working group paper that were not found in any of the subsequent publications. They are: Expert Behavior (EB), Gender Identity (GI), Learning via Gamification (LG). Mobile Device (MD), Novices vs Near-Novice Learning (NNN), Predicting Student Success (PSS), and User Interface (UI).

The papers reported here were published in 12 different conferences and two journals, shown in Table 2, with the most papers published at a specific conference was three papers, as three were published both at SIGCSE TS 2023 and ITiCSE 2023. Combined with the venues from the 2022 Working Group paper, the top four conferences that published papers on Parsons problems are: SIGCSE TS with 16 papers; ITiCSE with 15 papers; ICER with 15 papers; and Koli Calling with 10 papers.

3 MIMN LITERATURE REVIEW

In this section, we present the multi-institutional, multi-national (MIMN) literature review conducted to support our work in understanding the challenges in conducting studies across institutions and countries. We describe our review in parts, with Section 3.1 describing the design of the MIMN literature review and Section 3.2 presenting the results.

3.1 MIMN Study Design

We conducted a systematic literature review [10] to analyze multi-institutional, multi-national (MIMN) studies performed within computer science education (CSE). We performed this review to understand the motivations and challenges associated with undertaking such studies so that we can learn from the prior experiences, and help us improve our studies and procedures.

For the MIMN literature review, we used the ACM Digital Library to identify peer-reviewed papers published between 2013 and 2023 using the following two search queries: *("multi-institutional" AND "computer science education")* and *("multi-national" AND "computer science education")*. Both queries looked for the search terms in the paper's title and body, identifying papers in PDF format published between January 2013 and July 2023 in journals and conference proceedings. We included the term *"multi-national"* to ensure the participating institutions were across countries and avoided studies conducted on different campuses at a single institution within one country. For example, a study by Stuurman et al. [101] was conducted with distance learning students from Belgium and the Netherlands, but the course was administered at one location, the Open University of Belgium. We wanted to review papers that conducted studies in two or more countries across multiple institutions within the CSE context. We decided to use country for the criteria because it concentrates on the geographic region. The two queries identified 135 papers which we filtered through our selection criteria.

During our initial review, we identified four MIMN studies, [34, 55, 63, 116], that were commonly referenced in the 135 papers related to MIMN in CSE. These four papers were not in our initial search results because they were not published between 2013 and 2023. However, we felt these papers were relevant to our goal of providing guidance on conducting MIMN studies. For their background in MIMN studies in CSE, we decided to include these four influential papers.

We also observed papers we believe were MIMN studies but were unable to verify as MIMN due to their study design and context. For example, during our Parsons problems literature review (See Section 2), the previously mentioned work by Hayatpur et al. [42] seemed to be a MIMN study with researchers representing the United States and Canada. The paper also referred to the participating institutions as "two well-known North American universities" so we could not confirm the countries and did not include it in our review because it did not meet our selection criteria. It is possible that there are more MIMN studies in CSE, but like Hayatpur et al. [42] some may lack the details on institutions and countries to classify as MIMN studies.

Paper	Description	Country	Size	Tags
A C Language Learning Platform Based on Parsons Problems [93]	Designs and implements a C learning platform with Parsons problems to reduce cognitive load in learning.	China	NA	CL, LP, RSPF
A Review of Worked Examples in Programming Activities [67]	Reviews the worked-example literature in the context of programming activities, focusing on code-tracing and code-generation	Canada	NA	LR
Adaptive Parsons Problems as Active Learning Activities During Lecture [25]	Tests the efficiency of solving adaptive Parsons problems versus writing the equivalent code as lecture assignments	USA	500	CL, LP, RSPF, IS
Cybersecurity Education in the Age of AI: Integrating AI Learning into Cybersecurity High School Curricula [40]	Teaches AI and Machine Learning intregrated into Cybersecurity to high school teachers to integrate into their curriculum using programming activites in NetsBlox	USA	12	IP, LP, RSNPF
Discovering, Autogenerating, and Evaluating Distractors for Python Parsons Problems in CS1 [99]	Makes contributions related to the selection and use of distractors in Parsons problems, including templates and a tool for generating distractors	USA	494	LP, RSPF, GPP
Exploring the Difficulty of Faded Parsons Problems for Programming Education [35]	Presents a novel open-source tool for delivering Faded Parsons problems, and exploring the relative difficulty of three distinct fading strategies as part of an evaluation in a first-year programming course	New Zealand	915	LP, PPP, RSPF
Evaluating the Performance of Code Generation Models for Solving Parsons Problems With Small Prompt Variations [89]	Explores the performance of the OpenAI Codex model for solving Parsons problems over various prompt variations.	USA, Finland, Ireland, New Zealand	NA	LP, RSPF
Genetic Algorithm Cleaning in Sequential Data Mining: Analyzing Solutions to Parsons' Puzzles [106]	Applies genetic algorithms to clean clustering sequence data based on actions taken by users while solving Parsons problems in order to provide understandable trajectories of the events and improve the quality of the clustering process	USA	NA	EA, LP, PSSP, RSPF
Integrating Parsons Puzzles within Scratch Enables Efficient Computational Thinking Learning [84]	Reviews architecture and implementation strategies developed to integrate Parsons Programming Puzzles with Scratch, and then analyzes the use of this new tool.	USA	75	CL, LP, RSPF, IP, SE

Table 3: Parsons Literature Review Results Part I

Paper	Description	Country	Size	Tags
Investigating the Role and Impact of Distractors on Parsons Problems in CS1 Assessments [100]	Runs a study that shows that the inclusion of distractors has a large impact on the amount of time students spend on solving Parsons problems questions.	USA	576	LP, RSPF
Learning Computational Thinking Efficiently How Parsons Programming Puzzles within Scratch Might Help [6]	Reviews architecture and implementation strategies developed to integrate Parsons Programming Puzzles with Scratch, and then analyzes the use of this new tool.	USA	38, 624	CL, IS, LP, RSPF
Metacodenition: Scaffolding the Problem-Solving Process for Novice Programmers [75]	Presents and investigates a new tool called Metacodenition, a programming environment for novices that provides metacognitive scaffolding around an existing problem-solving framework	Australia	821	IS, LP, PPSS, RSPF
Putting Computing on the Table: Using Physical Games to Teach Computer Science [70]	Introduces and investigates a new non-coding, physical game-based curriculum for middle school students that focuses on abstraction, representation, and algorithm development	USA	67, 53	IP, RSNPF, SP
Strategies to increase success in learning programming [33]	Describes a set of activities related to the initial learning of programming, with Parsons problems as one of the activities.	Portugal and Spain	87	LP, RSNPF
Structuring Collaboration in Programming Through Personal-Spaces [42]	Explores a novel collaboration paradigm that tackles potential pair-programming driver/navigator imbalances while students work Parsons problems	USA and Canada	18	CSP, LP, PPSS, RSPF
Teaching computational thinking using scenario-based learning tools [123]	Teaches computational thinking to generation Z students with scenario-based learning tools, including Parsons problems.	Greece	23	IP, SP, RSNPF, LP
Teaching Test-Writing as a Variably-Scaffolded Programming Pattern [56]	Presents a new system design that uses faded Parson's problems to teach test-writng and advanced programming patterns to more advanced students	USA	NA	IS, KT, PPP
The Impact of Solving Adaptive Parsons Problems with Common and Uncommon Solutions [43]	Presents the results from think-alouds and a mixed within-between-subjects experiment undergraduates exploring cognitive load when students solve Parsons problems with common vs uncommon solutions	USA	95	CL, IS, LP, RSPF, SAS
Using Adaptive Parsons Problems to Scaffold Write-Code Problems [47]	Explores two studies on how students can use Parsons problems as scaffolding to solve write-code problems.	USA	11, 81	CL, IS, LP, RSPF
Using Micro Parsons Problems to Scaffold the Learning of Regular Expressions [118]	Introduces micro Parsons problems for solving regular expressions, where the problem consists of one line of fragments that are assembled in a single line	USA	3752	CL, LP, RSPF, SP

Table 4: Parsons Literature Review Results Part II

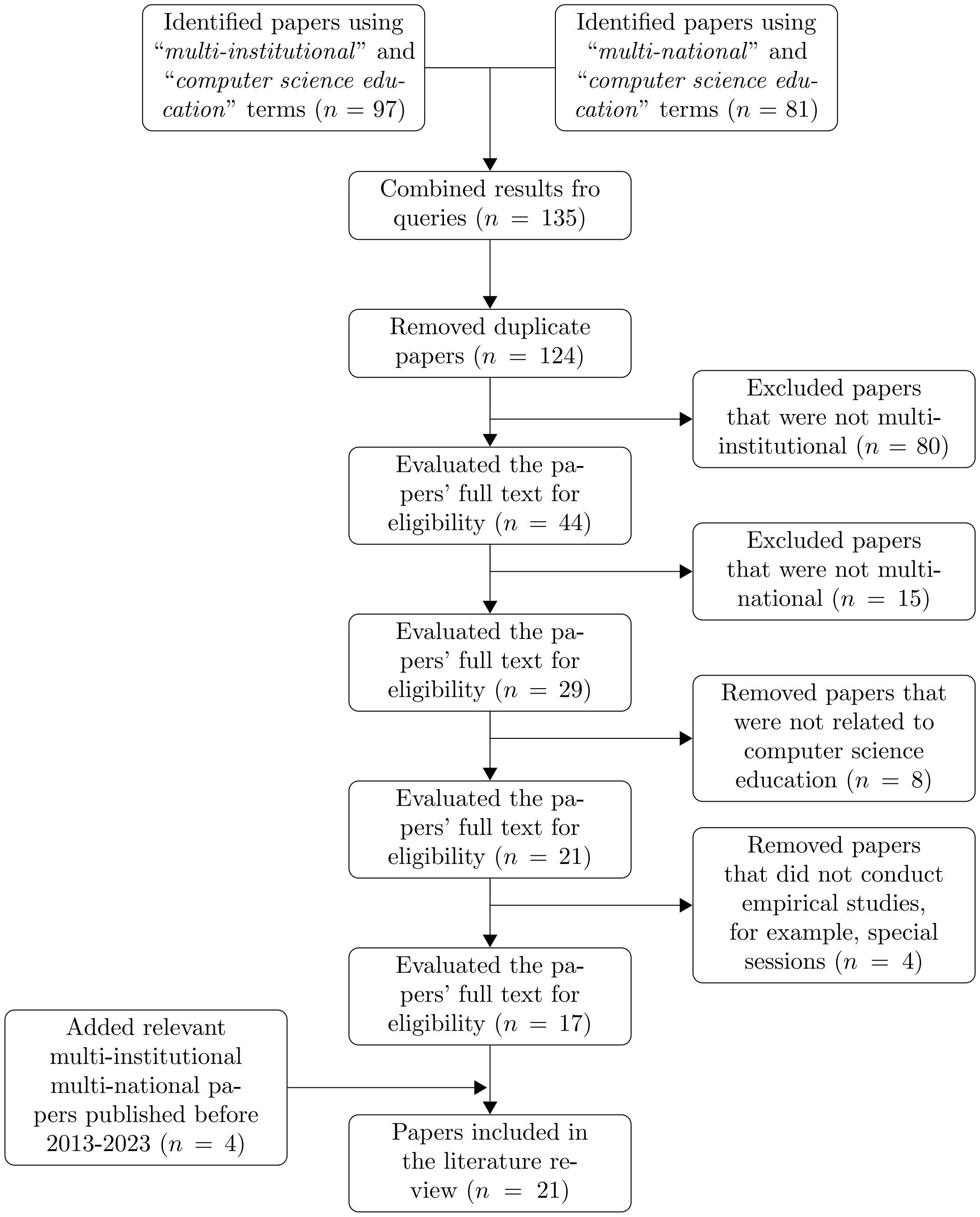

Identified papers using "multi-institutional" and "computer science education" terms ($n = 97$)
Identified papers using "multi-national" and "computer science education" terms ($n = 81$)
Combined results fro queries ($n = 135$)
Removed duplicate papers ($n = 124$)
Excluded papers that were not multi-institutional ($n = 80$)
Evaluated the papers' full text for eligibility ($n = 44$)
Excluded papers that were not multi-national ($n = 15$)
Evaluated the papers' full text for eligibility ($n = 29$)
Removed papers that were not related to computer science education ($n = 8$)
Evaluated the papers' full text for eligibility ($n = 21$)
Removed papers that did not conduct empirical studies, for example, special sessions ($n = 4$)
Evaluated the papers' full text for eligibility ($n = 17$)
Added relevant multi-institutional multi-national papers published before 2013-2023 ($n = 4$)
Papers included in the literature review ($n = 21$)

Figure 1: Exclusion Criteria for Literature Review

Figure 1 provides a visualization of the selection process, which shows from the original 135 papers, 17 (13%) met our selection criteria. The figure shows eleven (8%) papers removed because they were duplicates when collating the two queries' search results. Eighty (59%) papers were excluded because they referenced MIMN studies within the background, related work, or future work sections but were not MIMN studies. Fifteen (11%) papers were removed because they were multi-international studies but not multi-national. Eight (6%) were eliminated because the study was not conducted in the CSE context. For example, the work by Beecham et al. [4] examines challenges in global software engineering, a software engineering approach that involves software engineers from around the globe working together to develop software. The goal of this paper was to create recommendations for best practices while working in a globally distributed environment. The study evaluated the global teaming model that contained recommendations for GSE but did not evaluate within the CS discipline. The paper appeared in the query results only because it references the CSE space. Four (3%) papers were excluded because they did not conduct studies, such as special session papers. In total, the selection process removed 118 (87%) papers. As a result, we reviewed 21 papers along with the four previously mentioned early influential MIMN studies in CSE [34, 55, 63, 116].

Six co-authors reviewed the 21 papers. Each co-author was responsible for two to five papers, extracting key points from them and saving the information to a Google Spreadsheet. The key points on the paper included a brief description of the paper, challenges the researchers experienced, and the motivations and lessons learned from conducting the study. We also extracted the research instruments used in the study, their proceedings, keywords, countries, institutions, and participants involved.

After collecting the papers' information in a spreadsheet, we quantified some of the attributes, such as the number of countries, participants, and institutions to display them in a numeric format. Some [92, 111] conducted studies with educators and students, which we separated into two groups to better show the participants involved.

For the studies' instruments, challenges, and goals, we performed thematic content analysis [62], to group the common themes together into categories. For the instruments, we created categories from the emerging instruments used in the study. Two co-authors reviewed the instruments listed in the previously mentioned spreadsheet and discussed how to classify the practical activities, such as *Programming Tests* and *Trial Assessments*. The co-authors decided to place practical assessments into the *Practical Tests* category while activities that measures students' knowledge of CS concepts were classified into the *Concept Tests* category. After the coding process, we extracted a matrix table to present the coding frequencies for the instruments.

For the studies' challenges, we used thematic content analysis with an initial coding framework of the challenges and guidelines provided by the influential papers described in Section 3.3. Two co-authors were involved in this process, coding the challenges with the coding framework so that we could discuss the trends. We discussed the identified research trends within the context of our study, enabling us to compare our work with the findings. When discussing the challenges presented in the reviewed papers, we discussed how to address these challenges, so that we could avoid them when conducting our research.

For the final analysis on the MIMN papers, we used the papers' keywords to organize them into high-level goals. Two co-authors were involved in the process, where one classified the papers and the other confirmed the emerging categories. The authors discussed the classification of a paper by Parkinson et al. [72] that wrote up an experience report using the Research in Practice Project Activity (RIPPPA), an activity to encourage researchers to get involved in MIMN studies. The authors agreed to place the paper in the *Artifact* category because it contains guidelines, methods, and experiences for conducting MIMN students. After the coding process, we quantified the coding frequencies which are discussed in the next section.

3.2 MIMN Results

We present the results of the MIMN literature review in two parts: findings from the influential MIMN studies [34, 55, 63, 116] (Section 3.3) and the 16 papers that met our literature review selection criteria (Section 3.4). We present the influential publications separately because the other works use them as guidance in their study designs and we discuss how they guide subsequent studies through their work. Section 3.5 concludes this section by bringing together the common challenges researchers faced when undertaking MIMN studies.

3.3 Early Influential MIMN Studies in CSE

As previously mentioned in Section 3.1, we reviewed four [34, 55, 63, 116] early influential MIMN studies in CSE discussing MIMN studies within CSE, serving as guidelines for future MIMN studies. One of the influential papers by Fincher et al. [34] reviews existing MIMN studies to present common characteristics across these studies and provides considerations for researchers wanting to conduct future MIMN studies. The work acknowledged that MIMN studies were emerging in the CSE discipline and wanted to support future studies by raising awareness of common challenges that may emerge during these studies. The paper provides considerations for coordinating a large number of researchers across different counties, which includes defining roles and responsibilities for team members. Other considerations include institutional characteristics, such as student population and enrollment, participant selection process, data cleanliness, and the analysis techniques to accurately address the research questions.

Like Fincher et al. [34], Whalley and Lister [116] also provided guidance on how to conduct a study across institutions and countries. They were motivated to help researchers contribute to the BRACElet group, a project that examines the relationship between reading and writing code in novice programmers. BRACElet is an ongoing study that focuses on novice programmers and has conducted workshops to form an educational collective designed to develop test questions used to assess CS1 students. The assessment questions do not focus on coding, but target students' reasoning skills during problem-solving while working with code. The paper describes the BRACElet study design, with the researchers acknowledging challenges in performing studies across institutions, such as bringing participants together with different abilities and backgrounds. This challenge made it difficult for the researchers to form findings that could be generalised across participants and institutions.

Fincher et al. [34] also included the McCracken working group [63] (MCGracken WG) since they observed that it was used as a model for subsequent MIMN studies. This McCracken WG study came out of concerns from educators about students' limited programming skills, and the ITiCSE WGs allowed the educators to explore this problem. The McCracken WG was an early ITiCSE WG that collected empirical evidence from 216 CS1 students across four institutions to determine their ability to program at the end of the course. The study had participants complete a 1.5-hour assessment with two sections: a practical assessment and multi-choice questions. The students had three short assignments to choose from for completing a practical assessment and a series of multiple-choice questions for measuring their understanding of programming concepts. The results from this study showed that students did not demonstrate the competency expected at the end of CS1 courses. A potential reason for these findings may be due to students' perception across the participating institutions that they did not have enough time to complete the assessment. From the researchers' perspective, they also observed that time pressure may have been a factor along with programming habits that impede students' ability to construct correct programs, such as mistaking a successfully compiled program with a solution that addresses the requirements. In this study, the researchers guide future work to address the challenges they experienced. For example, some students in the participating institutions had taken prior programming courses, which produced a variety in the sample size. The researchers suggested a background questionnaire to promote a more even sample of participants, which would help to identify data from participants with the specific level of programming experience for the study. Another challenge was ensuring the complexity of problems was uniform across the assessments presented in English and other languages. To encourage alignment, the researchers suggest generalizing the instrument and avoiding preexisting knowledge assumed from students residing in some areas of the world. The researchers also acknowledged that coordination was complex across the researchers residing in different countries, suggesting future groups have a coordinator to facilitate the researchers' work.

The McCracken WG inspired other computer science education researchers to evaluate further why CS1 students struggle with programming, such as Lister et al. [55] finding other explanations. The Lister et al. [55] study evaluated CS1 students' programming abilities by confirming their ability to perform tracing on routine programming tasks, which would demonstrate a basic understanding of the essential programming principles taught in CS1. The study confirmed through using short coding questions (MCQs) that students have a fragile understanding of the skills needed for problem solving. Like the McCracken WG, this study noted differences in students' abilities across institutions. Part of this might stem from how the activity was presented and graded, generating higher motivation to successfully complete it because it was compulsory at some institutions while others presented it as non-compulsory.

The papers described in this section acknowledge the difficulties in conducting studies across institutions, which includes conducting activities, coordinating data and communicating with researchers across institutions and countries. These papers provide guidelines for future work to help mitigate challenges that emerge in MIMN studies and support the papers discussed in the next section with MIMN studies in CSE conducted over the past ten years.

3.4 MIMN Studies in CSE (2013-2023)

We present the findings of the MIMN studies using two tables. The first table, Table 5, provides a brief description of the MIMN studies, organizing the papers in chronological order. These are the 17 papers from the past ten years, 2013-2023. The table summarizes the challenges researchers faced when conducting these studies. For example, Grissom et al. [39] conducted a study that surveyed faculty in the USA and Canada about their use of student-centered practices. This study had multiple challenges with a low response rate to the surveys, educators using different terminology to describe the practices, and the cultural differences from the institutions applying these practices. Because of the applied terminology and cultural differences across the participating institutions, the researchers limited the study results to one Canadian and 45 USA institutions.

The second table, Table 6, presents the studies' characteristics and, like Table 5, organizes the findings chronologically. We use both tables to discuss the results further, presenting the MIMN studies by their general characteristics (Section 3.4.1), their instruments (Section 3.4.2), their overall goals (Section 3.4.3), and their motivations (Section 3.4.4).

3.4.1 *General Characteristics.*

Table 6 presents additional information on the studies' characteristics, including the number of countries (Cos, median = 3), the institutions (Inst, median = 6.5), and the participants (Pcps, median = 357). The table displays the participants as students unless otherwise specified as educators. For example, Švábenský et al. [111] conducted a study involving 22 educators and 46 students. In this work, the researchers evaluate graph models that help educators visualize students' progression through cybersecurity exercises.

When evaluating the country involved in MIMN studies, we observed that three (18%) papers did not specify the origin countries. For example, the study by Porter et al. [77] investigates the deployment of Peer Instruction in introductory courses, where three countries participated in the study; however, the authors of this paper did not explicitly state the countries. We assumed the countries based on researchers' geolocations, but in the table, we define the countries as "Not specified" due to lack of certainty. Countries involved in MIMN studies include the USA, Ireland, and Great Britain, but we observed limited representation from Africa, Asia, and South America. These findings raise questions about the limited representation of CSE research from these regions. Further investigation might provide reasons for low participation. For example, the lack of representation could be due to language barriers or additional textual language translation necessary to get involved in studies. Understanding the limited participation could help develop strategies that encourage researchers and institutions from these regions to get involved in MIMN studies.

Table 6 also presents the source of the papers' proceedings, where the majority (n=13, 76%) are from ACM conferences. Three [11, 96, 109] (18%) of the papers are empirical studies generated from previous ITiCSE Working Groups (WG). For example, the work by Utting et al. [109] extends the McCracken WG study. Ian Utting, a contributor to the McCracken WG, built on the McCracken study by increasing the scaffolding and including a test harness to support students while solving a programming problem. Like the McCracken WG, the study design included an assessment test and Multiple Choice Questions (MCQs) to evaluate students' understanding of learning concepts. The study design included educators' expectations of students' performance to strengthen the findings. Compared to the McCracken WG, the Utting study improved the correlation between the educators' expectations and their students' performance. The results showed that the test harness supported students' performance, positively affecting students completing the activity.

Year	Paper by Title	Countries	Brief Description	Challenges
2013	Identifying Threshold Concepts: From Dead End to a New Direction [94]	Ireland & Great Britain	Describes a novel approach to identify threshold concepts.	Students' data collected was not helpful in identifying threshold concepts.
2013	A Fresh Look at Novice Programmers' Performance and Their Teachers' Expectations [109]	USA, Great Britain, Denmark, Israel, Finland, Poland and four more	Builds on the McCracken 2001 study by providing CS1 students with scaffolding and gathering teachers' expectation on students' performance.	Institutional restrictions made it difficult to collect student data; ITiCSE WG time constraints; Different backgrounds and abilities of the student participants.
2014	Benchmarking a Set of Exam Questions for Introductory Programming [92]	Australia, New Zealand	Examines two themes in CS1: students' performance and the different styles of exam questions.	Translating activity to different programming languages; Student performance across institutions resulted in excluded data.
2016	Novice Programmers and the Problem Description Effect [11]	USA, UK, China, Slovakia	Examines the effects of problem contextualization on novice programmer success in a typical CS1 exercise.	Volunteer bias; Different approaches to teaching courses with educators and instructional materials.
2016	A Multi-Institutional Study of Peer Instruction in Introductory Computing [77]	Not specified	Investigates peer instruction (PI) in introductory courses.	Different course contexts; Different demographics and education systems; Educators' experience with PI; Novelty effect with students.
2017	The Compound Nature of Novice Programming Assessments [58]	Not specified	Evaluates examination questions used to assess novice programming at the syntax level.	Not listed
2017	Insights on Gender Differences in CS1: A Multi-institutional, Multi-variate Study [87]	Ireland, Denmark	Compares the profiles of students enrolled in CS1 courses early in the courses based on their gender.	The study used a programming test, which was graded manually, resulting in the reporting of one institution's data.
2017	An Instrument to Assess Self-Efficacy in Introductory Algorithms Courses [17]	USA, Germany	Evaluates an instrument that assesses self-efficacy in the context of an algorithms course.	Most of the reporting came from one institution; Different approaches to teaching algorithms by the educators may have influenced results.
2017	How Student Centered is the Computer Science Classroom? A Survey of College Faculty [39]	USA, Canada	Surveys educators on their use of student-centered practices in the classroom.	Difficulty in getting teaching staff to respond to the survey.
2018	Programming: Predicting Student Success Early in CS1. A Re-validation and Replication Study [86]	Ireland and Denmark	Builds on the work into factors that predict student success in CS1 using the PreSS model.	None listed.
2018	An International Investigation into Student Concerns regarding Transition into Higher Education Computing [119]	Sweden, USA, Ghana, UK, Canada	Investigates issues that lead applicants to experience levels of concern when considering a transition into higher education.	The initial survey was designed for Scotland, making it unsuitable as a multi-national instrument; Low response rate to the survey.
2021	Challenges Faced by Teaching Assistants in Computer Science Education Across Europe [91]	Norway, Sweden, Czech Republic	Surveys TAs on the challenges they face tutoring CS courses.	Different interactions and responsibilities made it difficult to generalize findings; Different sample sizes between the three institutions influenced the comparisons.
2021	Visual Recipes for Slicing and Dicing data: Teaching Data Wrangling using Subgoal Graphics [102]	UK, Pakistan, USA, Egypt, Finland	Investigates subgoal labels as a scaffolding strategy for novices to decompose problems.	The multi-national study inflated the data variance; Participants were not at the desired learning level.
2022	Evaluating Two Approaches to Assessing Student Progress in Cybersecurity Exercises [111]	USA, Czech Republic	Compares two visual models for students to progress through cybersecurity assignments.	None listed.
2022	Experience Report: Running and Participating in a Multi-Institutional Research in Practice Project Activity (RIPPA) [72]	Two undefined countries	Describes experience conducting MIMN study, providing recommendations for the community.	Time constraints; Differences in ethics (IRB) approval.
2022	The Impact of COVID-19 on the CS Student Learning Experience [96]	UK, Canada, Japan, USA, Pakistan, Brazil, Switzerland	Investigates the impact of remote learning during COVID 19 on students learning experiences.	Time constraints prevented IRR calculations; The remote WG influenced the depth of the thematic analysis.
2022	PreSS: Predicting Student Success Early in CS1. A Pilot International Replication and Generalization Study [88]	USA, Ireland	Conducts an international replication and generalization study on PreSS.	Lack of diversity in countries, language, university level, and topics; Small data made it difficult to generalize; Bias in the institutional quality.

Table 5: MIMN Literature Review: Study Descriptions

Overall the general characteristics of MIMN studies show these studies have a substantial dataset (Pcps, Median=357). In addition, the participating countries (Cos, Median=3) have multiple institutions (Inst, Median=6.5) within the countries participating in the study.

3.4.2 Instruments. Part of our review evaluated the instruments used in the studies to determine how data was collected. The most commonly used instrument was *Surveys* (n=13, 45%). For example, Sheard et al. [92] developed a benchmark for use across institutions that measures student performance. This work was motivated by poor performance educators observed by CS1 students across multiple institutions. The study asked educators from multiple institutions to provide examination questions that measure CS1 students' performance. When evaluating these questions across institutions, the results identified "four simple questions in introductory programming courses at a wide range of institutions" [92, p. 113].

The next instrument commonly used in MIMN studies was *Programming Tests* (n=7, 24%) which evaluate students' understanding of programming concepts. The study by Bouvier et al. [11] focused on the contextualization of programming exercises for CS1 students. The use of programming tests in this study is not surprising since they wanted to measure students' understanding of programming concepts, and most (n=11, 65%) studies were centered on the students.

Instruments infrequently applied were *Interviews* (n=1, 3%), *Concept Mapping* (n=1, 3%), *Non-Programming Exercises* (n=1, 3%), *Reflection Essays* (n=1, 3%), and *Teacher Reflections* (n=1, 3%). An example study using one of these instruments is by Riese et al. [91], who applied reflection essays to collect teaching assistants' (TAs) perceptions of managing a high number of enrolled students in the classroom. The survey asked the TAs to give detailed feedback on the assessments and individual tutoring. Using qualitative analysis, the researchers identified five main challenges, such as defining and using best practices, which allowed them to discuss the ethical dilemmas for TAs and outline implications for future TA training. It is possible that these instruments were not commonly used due to the qualitative analysis typically required to report findings from the collected data. However, more work is required to draw a conclusion.

3.4.3 Overall Goals. When evaluating the overall goals of these papers, the foci align with three broad categories: *Student* (n=11, 64%), *Teacher* (n=3, 18%), and *Artifact* (n=3, 18%). Student-centered research was the most common, focusing on students' perceptions and learning. For example, Siegel et al. [96], another ITiCSE WG, conducted an empirical study that investigates the impact of remote learning on students' learning experiences during COVID-19. The study looked at different factors that influence the students' experiences, including the type of virtual learning, mental health, and study skills. Though the researchers leveraged work done by a previous ITiCSE WG [95] focusing on COVID-19, building on the background research and a multi-national study with educators to collect their experiences with online tools and technologies, the authors ran out of time to report their findings by the ITiCSE WG deadline. The second category, *Teacher*, contained papers investigating instructional strategies to improve teaching practices. An example of this is the previously mentioned Porter et al. [77] work examining Peer Instruction (PI) in CS1 course, providing "evidence that introductory computing instructors can successfully implement PI in their classrooms" [77, p. 358]

The final category, *Artifact*, contained three papers [72, 92, 94] with contributions on products or objects for practices and learning, such as assessment quality and critical self-reflection. One paper focusing on artifacts was by Parkinson et al. [72] that describes the Research in Practice Activity (RIPPA), an initiative designed to support researchers in conducting Computing Education Research (CER). The United Kingdom and Ireland Computing Education Research (UKICER) conference supports RIPPAs through a collaborative pathway at the conference, encouraging the research community to conduct MIMN studies [51]. To evaluate the RIPPA, the authors used critical self-reflection to identify ways to improve the activity, including examining extending support to research-practice collaboration for future studies. Considerations for future support include encouraging groups to start the research early and having the groups meet frequently to discuss study goals and outcomes. The RIPPA report demonstrates the research community's continual interest in improving support for MIMN studies where centering on the collaborative experiences can help promote higher-quality research.

3.4.4 Motivations for MIMN Studies. We found several motivations for MIMN studies such as to help understand teaching practices [11] and to generalize instruments [87, 92] for use across multiple institutions and countries. The Quille et al. [87] study examined gender early in CS1 courses to determine if there are any significant differences in background, programming, self-efficacy, and anxiety. The PreSS (Predict Student Success) model, presented to students as two web-based surveys, enabled student participation across multiple institutions in Ireland and Denmark. The collected data provided opportunities to generalize the findings so that institutions can understand students' self-efficacy, comfort, and anxiety by gender and help promote strategies for retention.

3.5 Considerations from Previous MIMN Studies

The literature review identified considerations for researchers when conducting MIMN studies. In this section, we present these considerations in bold, organizing them into three areas: Team Coordination (Section 3.5.1), Institutional Considerations (Section 3.5.2), and Study and Data Integrity (Section 3.5.3). The papers providing the considerations suggest they can promote a positive experience when conducting the study. We reflect on these considerations for our study, which we discuss in Section 7.2.

3.5.1 Team Coordination. Collaboration can be difficult for teams conducting an MIMN study because the research group is globally distributed [63]. Considerations for **team coordination** are necessary to synchronize the team in achieving the study's goals. Considerations include **starting early** on the project and having the group **meet frequently**. By starting early, the group can **formalize the participant list early** to help define the relationship within the group [72] for researchers to collaborate on tasks. During the group meetings, the researchers can **reiterate and discuss the**

Year	Paper	# Co	# Inst	# Pcps	Instruments	Keywords	Proceedings
2013	Shinners-Kennedy and Fincher [94]	2	3	32	Survey, Interviews, Concept-mapping	Threshold Concepts, Hindsight Bias, PCK	ICER
2013	Utting et al. [109]	10	12	418	Concept Tests, Programming Tests, Teacher Reflections	Programming, CS1, Assessment, Replication	ITiCSE WG
2014	Sheard et al. [92]	2	6	826, 17^E	Surveys, Concept Tests	Standards, Quality, Examination Papers, CS1, Introductory Programming, Assessment	ACE
2016	Bouvier et al. [11]	4	6	232	Programming Tests	Context, Novice Programmers, CS1	ITiCSE
2016	Porter et al. [77]	3	8	363	Surveys	Faculty Adoption, Clickers, Peer Instruction	SIGCSE
2017	Luxton-Reilly and Petersen [58]	-	3	9	Programming Tests	CS1, Novice Programming, Concepts, Syntax, Assessments, Exams, Questions	ACE
2017	Quille et al. [87]	2	11	693	Surveys, Psychological Questionnaires, Programming Tests	Computer Science Education, Gender, Female, Programming Self efficacy, Programming, CS1	ITiCSE
2017	Danielsiek et al. [17]	2	4	362, 130*	Surveys	Computer Science Education, Algorithms, Self Efficacy	ICER
2017	Grissom et al. [39]	2	46	684	Surveys	Instructional Practice, Evidence-based Instructional Practices, Student-Centered, Instructor-Centered, Active Learning	ACM TOCE
2018	Quille and Bergin [86]	2	11	692	Surveys, Psychological Questionnaire, Programming Tests	Computer Science Education, Programming, Success, CS1	ITiCSE
2018	Zarb et al. [119]	5	9	351	Surveys	Concerns, Transition, CS1, Retention, Higher Education	ITiCSE
2021	Riese et al. [91]	3	3	180	Reflection Essays	Teaching Assistants, TAs, Challenges	ITiCSE
2021	Sundin et al. [102]	5	3	288	Surveys & Non-programming Exercises, Programming Tests	Data Science, Data Wrangling, Programming Education, Visualization, Graphics, Subgoals	Koli Calling
2022	Parkinson et al. [72]	-	-	-	Surveys, Concept Tests	Experience Report, Multi-institutional, Spatial Skills, RIPPA	UKICER
2022	Švábenský et al. [111]	2	2	46, 22^E	Surveys	Cybersecurity Education, Command-line History, Educational Data	SIGCSE
2022	Siegel et al. [96]	7	23	304	Surveys	COVID-19, Coronavirus, Computing Education, Online Education, Student Perspective	ITiCSE WG
2022	Quille et al. [88]	2	3	472	Surveys, Psychological Questionnaire, Programming Tests	Computer Science Education, Programming, Machine Learning, Predicting Success, CS1	ITiCSE

E - Participants are educators

* - Measuring student participants pre-post course

Table 6: MIMN Literature Review Results: Study Characteristics

project outcomes and goals frequently to ensure the researchers align on the project's goals and tasks.

The coordination aspects MIMN researchers need to discuss are communication protocols, project timelines, and meeting times that accommodate group members within different time zones. The group also needs to discuss **data ownership**, deciding early in the project what to do with the data after the study. The group needs to decide how to release the data publicly, how other studies can use the data, and how, when, and where the data should be archived [34]. The team can also benefit from **continually checking for skills development opportunities** to help team members develop skills [72], such as research skills.

3.5.2 Institutional Considerations. It is well established that educational institutions are different. Differences include course curriculum, course delivery, and size of the courses. These differences, or **institutional characteristics**, can influence the collected data, potentially impacting how future research reproduces the study at their institutions [34]. To address these differences, the research group needs to consider the **selection of participants** across the institutions. The recruitment may vary due to the size of the cohort and their availability. The institutions may have different participation protocols, for example, student participants are required to complete the study or participation is voluntary [34].

Another institutional characteristic is **grades** [34] and comparing these grades across institutions cannot accurately represent the participants' performance. In addition, **comparing students' performance** using the collected data needs to be considered because the students across the participating institutions have different backgrounds and abilities. As a result, the study design has to include assurances to mitigate these differences influencing the results. Overall, to help collect data across the institutions for comparison and to ensure participants meet the study's requirements, the study design and instruments can "specify the level of prior programming experience or the specific programming knowledge that the students are assumed to have for each exercise" [63, p. 143].

Lastly, another consideration for institutional characteristics is the **ethics (IRB) approval** process, where the timing, requirements, and application differ. Fincher et al. [34] recommend that the first task is securing IRB approval so that the team can complete their project within the given timeline. Siegel et al. [96] experienced first-hand as an ITiCSE WG that approval of IRBs can delay a research project. For the Siegel WG, the delay caused a shorter time frame for the group to complete the project.

3.5.3 Study and Data Integrity. Another way to ensure success in an MIMN study is to have a robust study design for the researchers to apply at their institutions. Strengthening and evaluating the robustness of the study design can involve a **pilot program**. The McCracken WG [63] encourages using a pilot program to form solid instruments, analysis processes, and data formats. A pilot program allows the research group to ensure **consistent data collection**. Prior MIMN studies [34, 63] noted challenges in data collection that include data wrangling to align the contents and structure of the data files. Prior work [120] has also stressed the importance of standardizing qualitative data collection to give researchers more equal and clean comparisons across the institutions.

Though the researchers strive to collect consistent and appropriate types of data for the study, they also have to consider the **character of the data** because the data can be different across the institutions, potentially affecting the analysis. The researchers must select the relevant parts of the data for comparative analysis [34]. Consistent data collection can help mitigate issues surrounding **data cleanliness**, giving researchers concrete data management guidelines that protect the integrity and reliability of the final data set [34]. In addition, the study can apply multiple instruments, potentially generating a variety of data or an incomplete data set due to different factors, such as attrition, where participants do not finish the study's interventions and instruments.

In addition to deciding on the data to collect, researchers must also decide on the **choice of analysis techniques**. For example, observational and unstructured interviews require extensive interactions and communications between researchers during analysis, which can be difficult to coordinate across researchers at different institutions. In contrast, quantitative data analysis requires less inter-reliability once they agree on statistical tests for the data [34].

Related to the data collection and analysis process is **reliability**. Some approaches to collecting and analyzing data are vulnerable to inter-rater reliability issues, such as observational studies, but to mitigate reliability issues, researchers should adopt a "a detailed "script" describing the data collection process, and the use of explicit checks for inter-rater reliability wherever possible in the data collection / or analysis process" [34, p. 117].

Some MIMN studies include institutions that use different programming languages for instruction or use different textual languages in the learning environment. These differences generate another consideration for ensuring consistent data is the presentation of the study design within these institutions. **Translation** of the study design, which includes interventions and instruments, may be necessary to ensure the translation complexity aligns with the native [63]] programming languages used in the original study design. With multi-national institutions involved in the study, the researchers should consider removing localization that assumes knowledge from a particular place. This is sometimes called *culturally neutral* [63]. This term may be misleading because the study cannot impose a culturally neutral study when the institutions bring their values to the classroom.

4 2023 WORKING GROUP PARSONS PROBLEM STUDIES

The 2022 ITiCSE Parsons working group created and piloted several studies for Python based on gaps identified by an extensive literature review [27]. For example, while research has shown that students can usually solve Parsons problems significantly faster than writing the equivalent code with equivalent learning gains, these studies have been conducted at a single institution, in a single country, in a single programming language/environment, and on introductory computing concepts [26, 28, 30, 121]. Therefore, there is a need to replicate these studies at other institutions in various nations, with more advanced concepts, more programming languages, and newer types of Parsons problems.

In addition, there is evidence for and against using distractors, i.e., blocks that are not needed in a correct solution. Parsons and

$$\begin{tikzcd}
 & & & & \text{Condition A} \arrow[dr] & & \\
\text{Pretest (Optional)} \arrow[r] & \text{Study Information Page} \arrow[r] & \text{Presurvey} \arrow[r] & \text{Introduction to Problem Types} \arrow[ur] \arrow[dr] & & \text{Posttest} \arrow[r] & \text{Postsurvey (Optional)} \\
 & & & & \text{Condition B} \arrow[ur] & &
\end{tikzcd}$$

Figure 2: The Study Pipeline for A-B Design

Haden [73] used distractors in the first Parsons problems and expected them to help students learn to recognize common syntax and semantic errors. Denny, Luxton-Reilly, and Simon [18] found that distractors increased the difficulty of a Parsons problem, providing a distractor for every correct block overwhelmed students, and visually paired distractors were easier than randomly mixing in the distractors with the correct code. Distractors can also help students focus on details and reduce the ability for students to solve a Parsons problem through simple heuristics such as variable name dependencies [26]. Harms, Chen, and Kelleher [41] reported that distractors increased reported cognitive load, decreased success, and increased time on task. However, they tested semantic distractors, not syntactic distractors, and tested learning by having students solve a Parsons problem without any distractors. This did not test the ability of distractors to help students learn to recognize and fix errors. Distractors may provide desirable difficulties in that even if they slow initial learning, and they may promote long-term learning [9]. Distractors can also keep students in the Zone of Proximal Development (ZPD), where students are challenged but not frustrated [113].

There has also been research that provides evidence that solving Parsons problems can help students learn common patterns [115]. However, again, that is from a single institution in a single country.

The overall research questions that the 2022 working group focused on were: What is the effect on completion time and learning performance for 1) solving adaptive Parsons problems with distractors versus writing the equivalent code, 2) solving adaptive Parsons problems with distractors versus Parsons problems without distractors, 3) solving write code problems with a Parsons problem as scaffolding versus a write code problems without scaffolding, and 4) Were there significant differences by high vs low self-efficacy or self-evaluation? The 2022 working group was also interested in the effect of success rates of solving a set of Parsons problems on students' ability to write code for common algorithms and the number of errors during code writing. The 2022 working group created four studies: *p3pt*, *class-tog*, *class-exp*, and *python-swap*. The study *p3pt* tests the effect of solving adaptive Parsons problems with distractors versus writing the equivalent code. The study *class-tog* tests the effect of using a Parsons problem as scaffolding during a code writing problem versus no scaffolding. The study *class-exp* investigates the effect of solving Parsons problems with and without distractors on the ability of students to fix code with errors similar to the distractors and write code from scratch. Finally, *python-swap* tests the effect of solving three Parsons problems on students' ability to reproduce a common algorithm: swapping the value of two variables.

Three of the studies were between-subject studies with two conditions (*p3pt*, *class-exp*, and *class-tog*) that took from 50-70 minutes, while the other (*python-swap*) was a within-subject study that took 20-30 minutes. The first three studies were all intended to be run after students had covered the basics of Python (variables, strings, loops, conditionals, and lists) and before they learned how to write new classes in Python. We created *python-swap* to be a shorter study that could be run early in an introductory programming course.

The 2023 Parsons working group reviewed these studies and decided to create a new Python study, *p3dnd*, which also tested solving Parsons problems with and without distractors since some of the working group members planned to recruit students who had already completed an introductory programming course in Python. The *p3dnd* study was intended to be harder than *p3pt*. The 2023 working group members also created versions of some studies for other programming languages. They created *jspt* based on *p3pt* for JavaScript but also with the study materials in Spanish. In addition, they created *c-swap* based on *python-swap* and *cdnd* based on *p3dnd* for C.

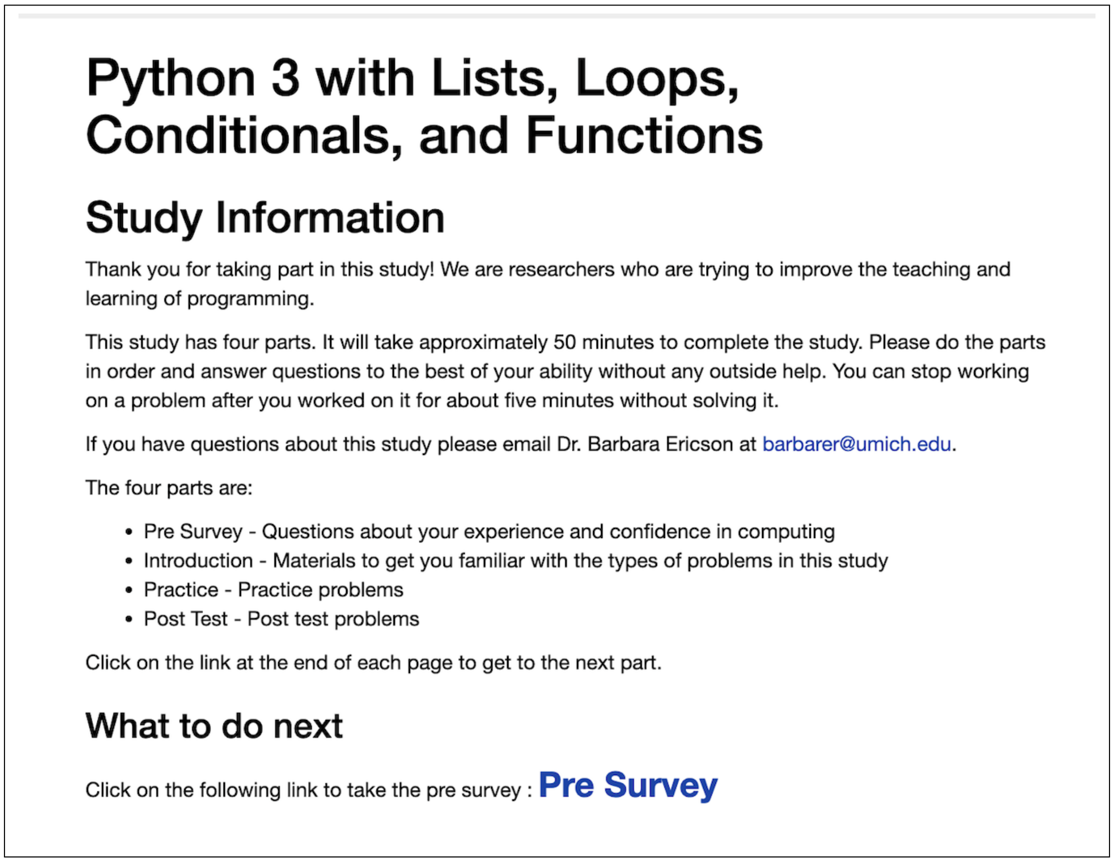
In addition, the 2023 Parsons working group also made a think-aloud version of *class-exp* called *classta* in which students were exposed to Parsons problems with distractors (WD), Parsons problems with no distractors (ND), and toggle problems (TP) which display a code writing problem but include the ability to pop-up the equivalent Parsons problem. This was still an A/B study with two conditions, where A was (ND, WD, ND, WD, TP) and B was (WD, ND, WD, ND, TP). The problems were in the same order in A and B, and they only varied by having distractors or not.

Finally, the 2023 Parsons working group also created a think-aloud version of *p3dnd* called *p3dndta* with six practice problems both with distractors (WD) and no distractors (ND). The A condition was (ND, WD, ND, WD, ND, WD), and the B condition was (WD, ND, WD, ND, WD, ND). Again, the problems were in the same order in A and B, with the only difference being whether they had distractors.

All of the studies included an information page about the study, a presurvey, an introduction to the problem types, a set of practice problems, and a posttest. In addition, there were two optional parts: a pretest and a postsurvey. The procedure for all between-subjects studies is shown in Figure 2.

4.1 Study Information Page

The information page gave an estimate of the time to complete the study, instructions on how long to work on a problem before giving up on it (five minutes), and explained the parts of the study as shown in Figure 3. Students clicked on the link at the end of each page to go to the next page. In between-subjects studies, students were randomly placed in either condition A or B based on generating a random number.


Python 3 with Lists, Loops, Conditionals, and Functions

Study Information

Thank you for taking part in this study! We are researchers who are trying to improve the teaching and learning of programming.

This study has four parts. It will take approximately 50 minutes to complete the study. Please do the parts in order and answer questions to the best of your ability without any outside help. You can stop working on a problem after you worked on it for about five minutes without solving it.

If you have questions about this study please email Dr. Barbara Ericson at barbarer@umich.edu.

The four parts are:

- Pre Survey - Questions about your experience and confidence in computing
- Introduction - Materials to get you familiar with the types of problems in this study
- Practice - Practice problems
- Post Test - Post test problems

Click on the link at the end of each page to get to the next part.

What to do next

Click on the following link to take the pre survey : Pre Survey

Figure 3: An example introduction about the study page. This one was for *p3pt*

4.2 Presurvey

The presurvey contained six Likert scale questions from a survey on self-efficacy for computing with evidence for reliability and validity [117]. Answers ranged from 1 (Strongly Disagree) to 5 (Strongly Agree).

(1) Generally I have felt secure about attempting computer programming problems.
(2) I am sure I could do advanced work in computer science.
(3) I am sure that I can learn programming.
(4) I think I could handle more difficult programming problems.
(5) I can get good grades in computer science.
(6) I have a lot of self-confidence when it comes to programming.

Rather than using a pretest to check that groups were not significantly different based on prior experience, we added questions to the presurvey that asked students to select the answer that best matched their familiarity and confidence about specified concepts. The concepts in the survey varied by study. While there are assessments of CS1 knowledge, such as SCS1 [71], that have evidence for validity and reliability, they are quite lengthy and cover more concepts than our studies. A recent study provided evidence that self-evaluation questions correlate with the score on SCS1 and the score on a code writing exam [21]. Students answered the self-evaluation questions using the following 5-point Likert scale.

(1) I am unfamiliar with this concept.
(2) I know what it means, but have not used it in a program.
(3) I have used this concept in a program, but am not confident about my ability to use it.
(4) I am confident in my ability to use this concept in simple programs.
(5) I am confident in my ability to use this concept in complex programs.

Having a set of questions about prior programming experience and knowledge in a MIMN study, as discussed in Section 3.5, can help account for potential differences between students from diverse institutions, such as backgrounds and abilities.

4.3 Introduction to Problem Types

Since most 2023 Parsons working group members do not usually use the Runestone ebook platform, we created a page to introduce students to the different types of problems they would have to solve in the studies. This introduction contained videos demonstrating how to solve Parsons problems and code-writing problems. It also included simple practice problems to test that students could solve each type of problem, as seen in Figure 4 and Figure 5. The 2022 Parsons problems working group piloted studies and found that all of the students successfully solved all of the practice Parsons problems. However, some students struggled to solve the practice code-writing problem even though it was very similar to the problem that was solved in the video. These students were not familiar with functions that took parameters or unit tests. Therefore, we modified our instructions to include that students should be familiar with unit tests and functions that take parameters before participating in the studies.

4.4 Optional Pretest

We developed an optional pretest that consisted of a timed exam with ten multiple-choice questions that measured basic knowledge of Python 3. In a timed exam, the students must start the exam by clicking the "Start" button. As depicted in Figure 6, the questions are shown one at a time. Students can select an answer but do not receive any feedback on their answers. Students navigate by clicking the "Next" or "Prev" button or instead the button for a particular question number. The exam shows the time left and will automatically stop when the time expires, and all answers will be saved. The multiple-choice questions covered strings, conditionals, functions, printing values, types, nested lists, a `for` loop with a range, a `for each` loop, a `while` loop, modulus, and `break` and `continue`. The questions have been used as a pretest to check students' knowledge of Python 3 in a programming course at the University of Michigan. See the appendix for all of the pretest questions.

We made the pretest optional to reduce the required time for the studies. To compare groups, we instead used the self-evaluation ratings on particular concepts. We recommended that if instructors wanted to use the pretest, they have students answer it on a different day than the study. If instructors used the optional pretest, they would have students start with an introduction to the timed pretest, which included a video to show how to start the exam, navigate between questions, flag a question to remind themselves to review it later and submit their answers. This page also included a practice timed exam with two simple multiple-choice questions, as shown in Figure 7. A link at the end of that page took students to the actual timed pretest.

4.5 Optional Postsurvey

The 2022 ITiCSE working group also developed an optional postsurvey that included questions on demographics, prior programming experience, ability to read and understand spoken English, and prior exposure to Parsons problems. We made this survey optional both to reduce the required time for the studies and because institutions in some countries are not allowed to ask demographic types of questions. The first seven questions allowed free text input and were:

(1) What is your age in years?
(2) What is your major or intended major, or program of study?
(3) What is your gender identity (woman, man, non-binary, etc, prefer not to say)?
(4) What year are you in your undergraduate education (1st, 2nd, 3rd, etc)?
(5) Please list any learning issues we should be aware of, such as Dyslexia, Autism, ADHD, etc or enter none.
(6) About how many hours have you been programming in Python?
(7) What language(s) do you speak at home?

Two questions asked the students to rate their ability to read and understand spoken English using a 5-point Likert scale where 1 = Poor, 2 = Below Average, 3 = Average, 4 = Above Average, and 5 = Excellent. We felt this was important since the study materials (text and videos) were originally in English.

The last question asked if the students had experience with Parsons problems before the study. They could select "Yes" or "No".

4.6 Study Details

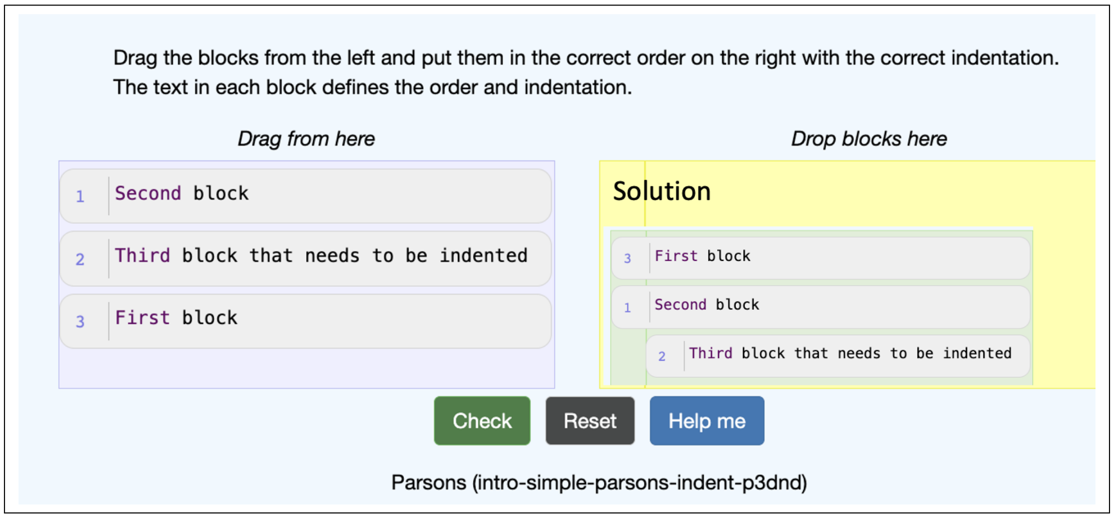
4.6.1 p3pt. This between-subjects study compares the learning performance and time to completion between solving adaptive Parsons problems with distractors versus writing the equivalent code. For example, Figure 8 shows the first practice problem as a Parsons problem on the left and a write code problem on the right. Students must be familiar with the following: Python 3 basics, including variables and modulus, strings (including `slice`), loops (`for each` and `for` with `range`), conditionals, lists, modulus, functions that take values, and unit tests. The ideal time for this study is after an introduction to lists and loops but before learners are proficient with lists and loops. The study takes about 50 minutes.

The self-evaluation concepts that students rated their level of familiarity for this study were:

(1) Loops/Iteration like `for n in nums:` and `for i in range(4):`
(2) Conditionals/Selection Statements like `if x < 3:`
(3) Functions like `def get_odd(nums):`
(4) Lists like `a = ["red", "green"]`

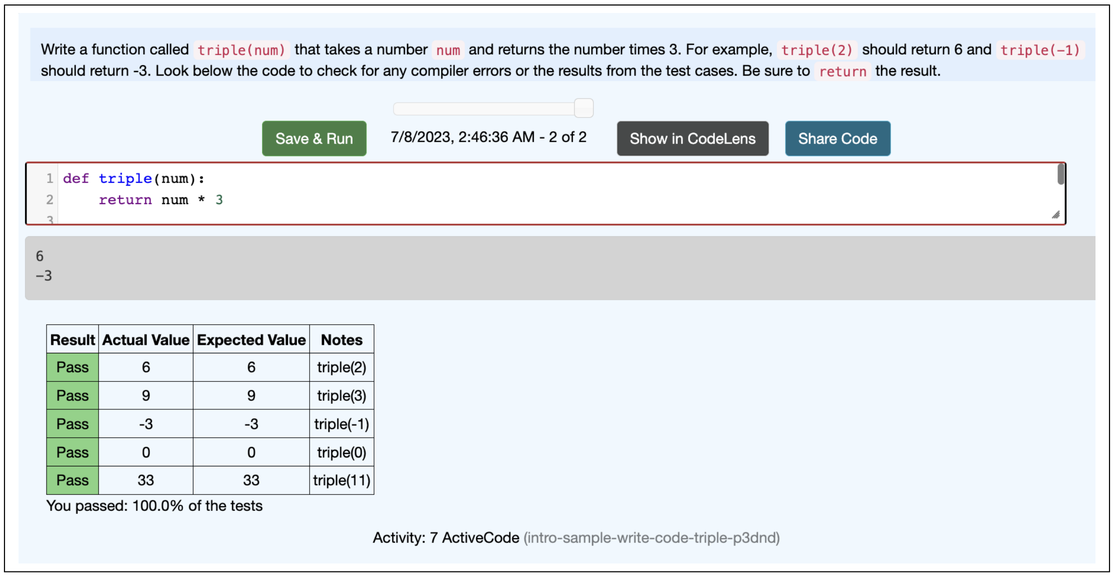
4.6.2 jspt. Building upon *p3pt*, we adapted the study to JavaScript to account for the difference in the programming language of instruction at one of the institutions. The conducted procedure was structured as follows: (1) First, we reviewed the Python items and code examples in the original version of the study (i.e., *p3pt*), ensuring that they were translated correctly into JavaScript syntax; (2) Next, one of the co-authors and a team of experienced instructors in CS1 verified that the overall perceived difficulty of the items did not get lost in translation; (3) Finally, we piloted the two versions of the experiment (i.e., *p3pt* in Python and JavaScript) to control as much as possible for ambiguity, objective specification of items, coding patterns, and comparable perceived difficulty between experiments. For example, Figure 9 shows the first practice problem involving string manipulation. Note the contrast to the first practice problem of *p3pt* (cf. Figure 8), where code indentation is required in Python, but in Javascript code blocks are defined in between curly braces (so indentation is not required). Since the study was conducted in a Spanish-speaking institution, the experiment items and platforms had to be translated due to the language barrier and potential accessibility concerns.

4.6.3 class-exp. This between-subjects study compares the learning performance and time to completion between solving adaptive Parsons problems with and without distractors, as shown in Figure 10. Students must be familiar with the following: Python 3 basics, including variables, strings, random, functions that take values, and unit tests. The ideal time for this study is before students have


Drag the blocks from the left and put them in the correct order on the right with the correct indentation. The text in each block defines the order and indentation.

Drag from here
1 Second block
2 Third block that needs to be indented
3 First block
Drop blocks here
Solution
3 First block
1 Second block
2 Third block that needs to be indented
Check
Reset
Help me
Parsons (intro-simple-parsons-indent-p3dnd)

Figure 4: Practice Parsons problem in the introduction to the problem types


Write a function called `triple(num)` that takes a number `num` and returns the number times 3. For example, `triple(2)` should return 6 and `triple(-1)` should return -3. Look below the code to check for any compiler errors or the results from the test cases. Be sure to `return` the result.

```python
def triple(num):
    return num * 3
```

```
6
-3
```

Result	Actual Value	Expected Value	Notes
Pass	6	6	triple(2)
Pass	9	9	triple(3)
Pass	-3	-3	triple(-1)
Pass	0	0	triple(0)
Pass	33	33	triple(11)

You passed: 100.0% of the tests

Activity: 7 ActiveCode (intro-sample-write-code-triple-p3dnd)

Figure 5: Practice write-code problem in the introduction to problem types

been introduced to writing new classes in Python. This study includes a short introduction to creating objects and writing new classes in Python, which comes after introducing the problem types and before the presurvey. This includes creating the `__init__` and `__str__` methods, creating new class objects, and adding additional methods to a class. The study takes about 60 minutes.

The self-evaluation concepts that students rated their level of familiarity for this study were:

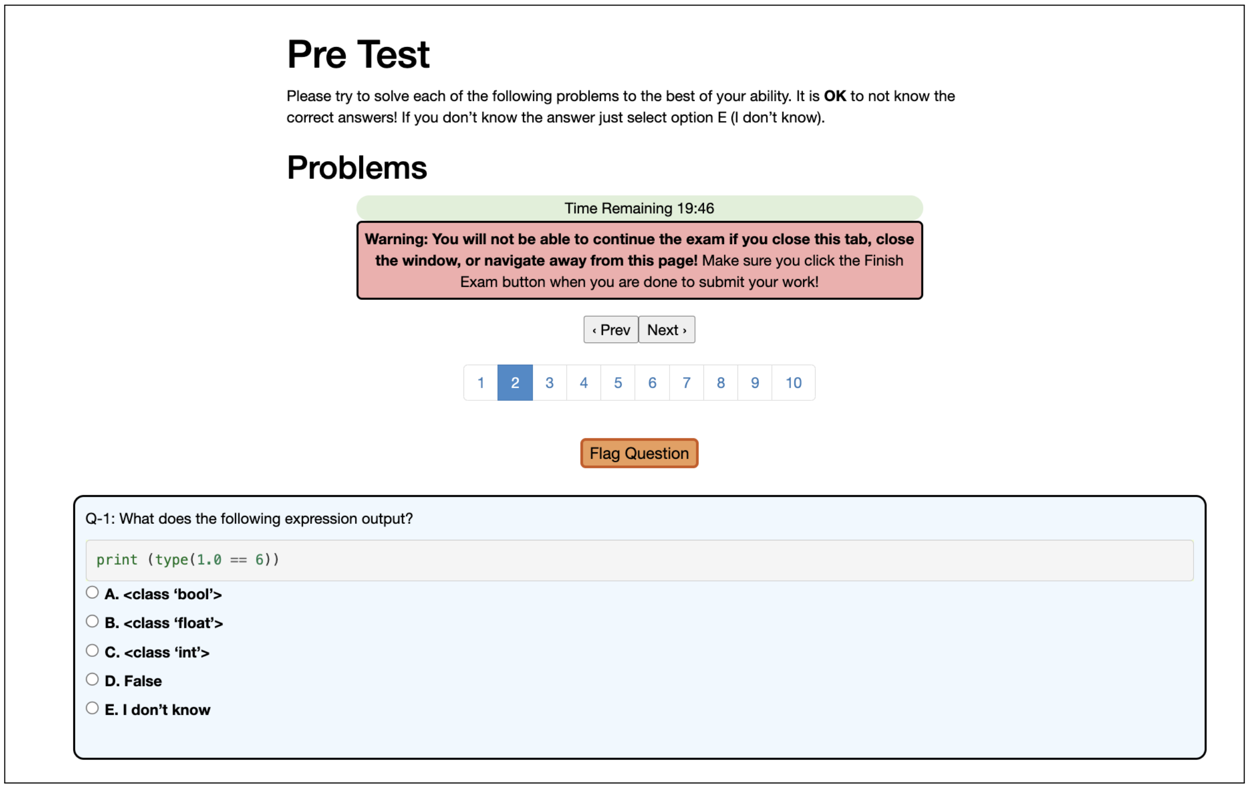
(1) Creating classes like `class Person:` and objects like `p = Person("Barb Ericson")`
(2) Methods like `__init__` and `__str__`
(3) The use of `self` in class
(4) Defining instance variables like `self.color = color`

4.6.4 *class-exp-ta.*

The *class-exp-ta* study is a version of the *class-exp* study designed for think-aloud observations. It exposes participants to solving Parsons problems both with distractors (WD) and no distractors (ND) as well as a toggle problem (TP) in which students are asked to solve a write code problem but can pop-up a Parsons problem as scaffolding [47]. The participants are placed randomly in conditions A or B. The A condition is WD, ND, WD, ND, TP, and the B condition is ND, WD, ND, WD, TP. The Parsons problems are in the same order in both A and B. The only difference is if they have distractors or not.

4.6.5 *class-tog.*

This between-subjects study compares the learning performance and time to completion between writing code with a Parsons problem as scaffolding versus writing code without


Pre Test

Please try to solve each of the following problems to the best of your ability. It is **OK** to not know the correct answers! If you don't know the answer just select option E (I don't know).

Problems

Time Remaining 19:46

Warning: You will not be able to continue the exam if you close this tab, close the window, or navigate away from this page! Make sure you click the Finish Exam button when you are done to submit your work!

Q-1: What does the following expression output?

```
print (type(1.0 == 6))
```

A. <class 'bool'>
B. <class 'float'>
C. <class 'int'>
D. False
E. I don't know

Figure 6: The pretest interface shows the second multiple-choice question in the pretest and the navigation buttons

a Parsons problem as scaffolding. See Figure 11 for an example. Students must be familiar with the following: Python 3 basics, including variables, strings, random, functions that take values, and unit tests. The ideal time for this study is before students have been introduced to writing new classes in Python. This study includes the same short introduction to creating objects and writing classes as class-exp. The study takes about 60 minutes. This study also includes the same self-evaluation questions as in *class-exp*.

4.6.6 python-swap. This study investigates how well students can learn to reproduce the code to swap the values of two variables after solving three Parsons problems. In the first Parsons problem, the blocks contain comment blocks describing the algorithm's steps, as seen in Figure 12. We refer to these blocks as *pseudocode comment blocks*. In the second Parsons problem, the blocks contain *pseudocode comments plus code*, as shown in Figure 13. In the third Parsons problem, the blocks contain *only code*, as seen in Figure 14.

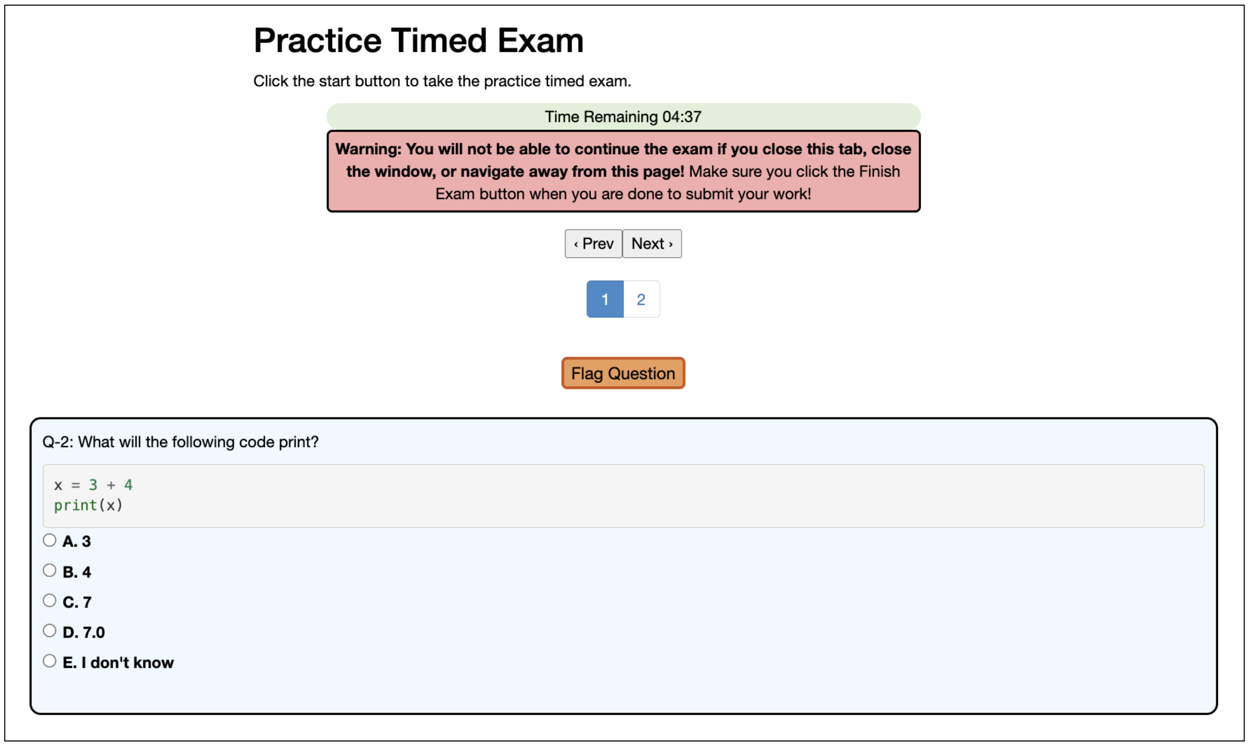
The self-evaluation concepts that students rated their level of familiarity for this study were:

(1) Setting the value of a variable like: `x = 4`
(2) Swapping the values of two variables so that `var1` has the original value of `var2` and `var2` has the original value of `var1`

The posttest had two write code problems where the variable initialization was provided, and students were asked to write the code to swap the values in the two variables as shown in Figure 15. The first used variable names of `x`, `y`, and `temp` just like the practice Parsons problems and the second problem used `a`, `b`, and `temp` in order to check for near transfer.

In order to gain further insight into any affordances provided by, or drawbacks created by, practice via Parsons problems, we conducted think-aloud interviews with students while they completed this study. Given this study did not involve the random assignment of students to conditions, we used this study as is, without any modifications for the think-aloud context.

4.6.7 c-swap. Building upon *python-swap*, we translated the study to C to be conducted at a broader set of institutions. The *c-swap* study had the same structure as *python-swap*, described in Section 4.6.6, that is, the same instructions, type of problems, pseudocode comments in the blocks, and variable names. The lines of code provided in the Parsons problems blocks were changed to C. For the first two Parsons problems, the blocks remained almost identical to *python-swap*, only with C syntax. For the third Parsons problem, the blocks contained the code for swapping two strings rather than integers, as shown in Figure 16. This change was done as the code for swapping strings is not directly transferable in C; thus, presenting the problem to students helps to highlight the additional steps to account for when working with different types and yet following the same logic. For the posttest, the first problem had the same structure as *python-swap*'s first posttest problem. The second posttest problem was also similar in structure; however, it tested for swapping strings like the third Parsons problem.



Practice Timed Exam

Click the start button to take the practice timed exam.

Time Remaining 04:37

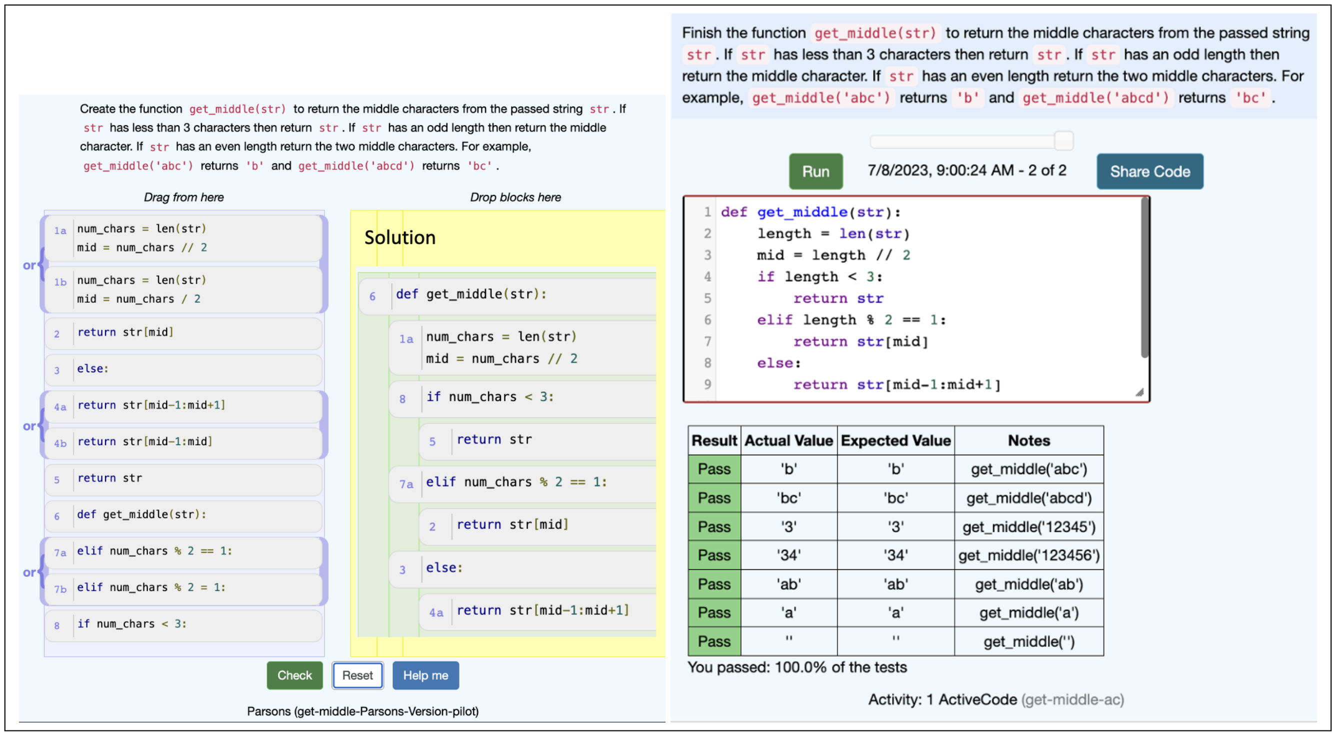
Warning: You will not be able to continue the exam if you close this tab, close the window, or navigate away from this page! Make sure you click the Finish Exam button when you are done to submit your work!

Q-2: What will the following code print?

```
x = 3 + 4
print(x)
```

- **A. 3**
- **B. 4**
- **C. 7**
- **D. 7.0**
- **E. I don't know**

Figure 7: The practice timed exam in the introduction to the timed pretest



Create the function `get_middle(str)` to return the middle characters from the passed string `str`. If `str` has less than 3 characters then return `str`. If `str` has an odd length then return the middle character. If `str` has an even length return the two middle characters. For example, `get_middle('abc')` returns `'b'` and `get_middle('abcd')` returns `'bc'`.

Drag from here

```
1a  num_chars = len(str)
    mid = num_chars // 2
or
1b  num_chars = len(str)
    mid = num_chars / 2
2   return str[mid]
3   else:
4a  return str[mid-1:mid+1]
or
4b  return str[mid-1:mid]
5   return str
6   def get_middle(str):
7a  elif num_chars % 2 == 1:
or
7b  elif num_chars % 2 = 1:
8   if num_chars < 3:
```

Drop blocks here

Solution

```
6   def get_middle(str):
1a      num_chars = len(str)
        mid = num_chars // 2
8       if num_chars < 3:
5           return str
7a      elif num_chars % 2 == 1:
2           return str[mid]
3       else:
4a          return str[mid-1:mid+1]
```

Parsons (get-middle-Parsons-Version-pilot)

Finish the function `get_middle(str)` to return the middle characters from the passed string `str`. If `str` has less than 3 characters then return `str`. If `str` has an odd length then return the middle character. If `str` has an even length return the two middle characters. For example, `get_middle('abc')` returns `'b'` and `get_middle('abcd')` returns `'bc'`.

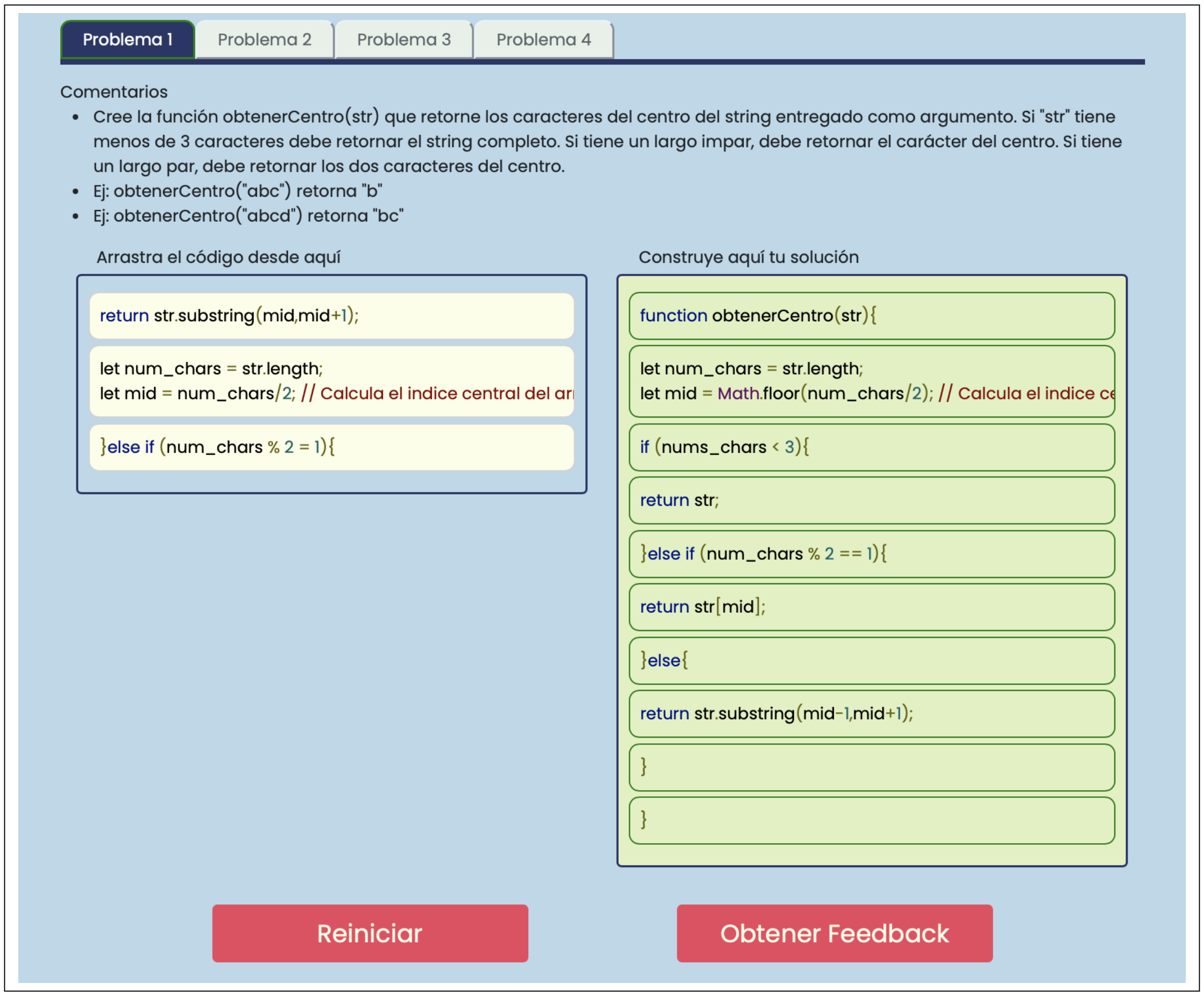
```
def get_middle(str):
    length = len(str)
    mid = length // 2
    if length < 3:
        return str
    elif length % 2 == 1:
        return str[mid]
    else:
        return str[mid-1:mid+1]
```

Result	Actual Value	Expected Value	Notes
Pass	'b'	'b'	get_middle('abc')
Pass	'bc'	'bc'	get_middle('abcd')
Pass	'3'	'3'	get_middle('12345')
Pass	'34'	'34'	get_middle('123456')
Pass	'ab'	'ab'	get_middle('ab')
Pass	'a'	'a'	get_middle('a')
Pass	''	''	get_middle('')

You passed: 100.0% of the tests

Activity: 1 ActiveCode (get-middle-ac)

Figure 8: First practice problem for *p3pt*, Parsons problem with distractors on the left and write code problem on the right



Comentarios

- Cree la función obtenerCentro(str) que retorne los caracteres del centro del string entregado como argumento. Si "str" tiene menos de 3 caracteres debe retornar el string completo. Si tiene un largo impar, debe retornar el carácter del centro. Si tiene un largo par, debe retornar los dos caracteres del centro.
- Ej: obtenerCentro("abc") retorna "b"
- Ej: obtenerCentro("abcd") retorna "bc"

```
return str.substring(mid,mid+1);
```

```
let num_chars = str.length;
let mid = num_chars/2; // Calcula el indice central del ar
```

```
}else if (num_chars % 2 = 1){
```

```
function obtenerCentro(str){
let num_chars = str.length;
let mid = Math.floor(num_chars/2); // Calcula el indice ce
if (nums_chars < 3){
return str;
}else if (num_chars % 2 == 1){
return str[mid];
}else{
return str.substring(mid-1,mid+1);
}
}
```

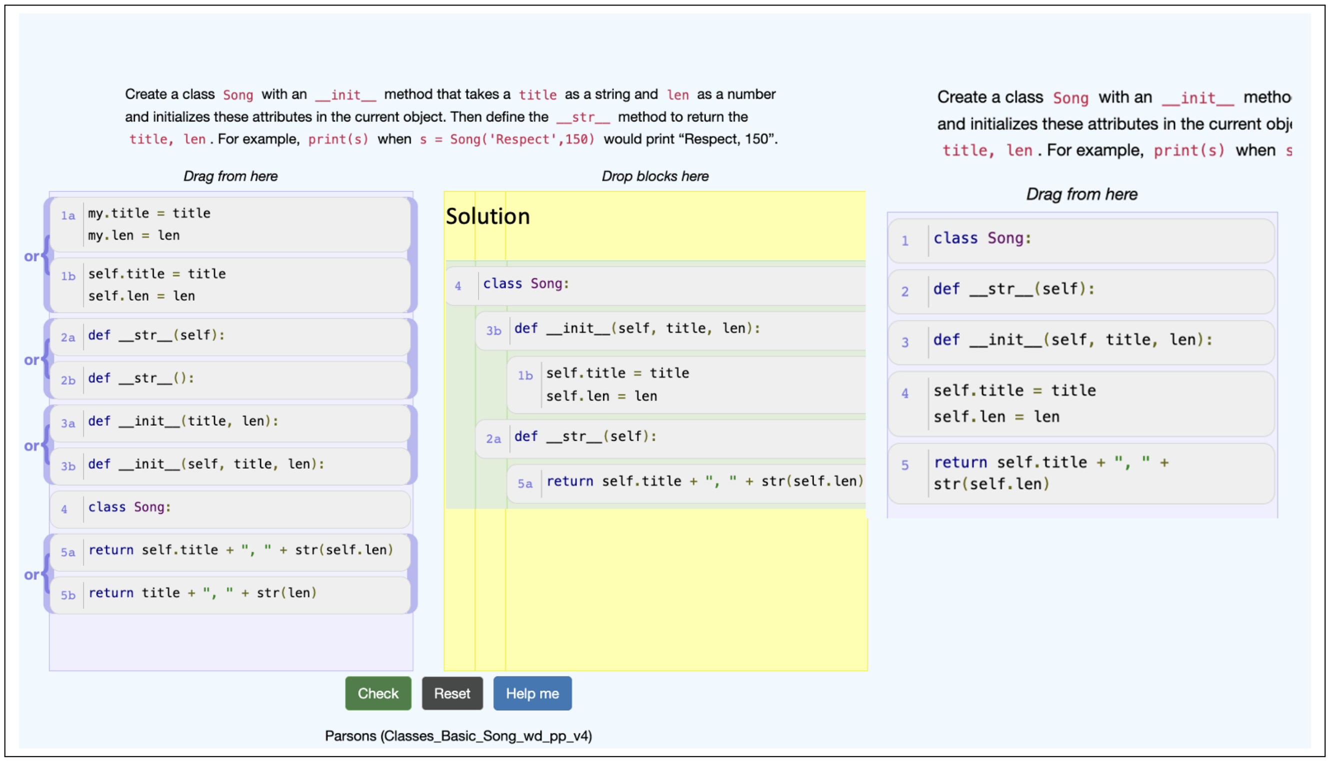
Figure 9: First practice problem for *j3pt* as a Parsons problem (in JavaScript)

The procedure to translate *python-swap* into C was similar to *jspt* presented in Section 4.6.2: (1) First, one co-author reviewed the Python code in the original version of the study, ensuring that they were translated correctly into C syntax; (2) Then, one of the co-authors and a team of experienced (and current) instructors verified that the overall perceived difficulty of the study did not change, even with the new added problem. Since students are already familiar with loops and how string variables behave differently than integer variables in C, the consensus was that the difficulty levels remain constant given the change of context.

4.6.8 p3dnd. Similar to *class-exp*, this study compares learning performance and time-to-complete between solving Parsons problems with distractors and those without. This study was created to address the need for a more complex set of algorithmic tasks and, as such, was comprised of problems from LeetCode[1] and CodingBat[2] that were deemed appropriate for CS1. Students should be familiar with the basics of Python3: loops, conditionals, built-in data structures (e.g., lists), and unit tests. The ideal time to run this study is after students are familiar with the basics of Python and are in the process of learning to construct solutions to problems that are of moderate complexity for CS1 students (e.g., the rainfall problem). The study was designed to take approximately 60 minutes for students to complete. It used the same self-evaluation questions as *p3pt*. The first practice problem is shown in Figure18 as a Parsons problem with distractors on the left and without on the right.

4.6.9 p3dnd-ta. The *p3dnd-ta* study was a redesign of the *p3dnd* study to conduct think-aloud observational studies. Given the original study compared students randomly assigned to one of two groups, this study removed that randomization such that, during a think-aloud, students would be exposed to questions with (WD) and without distractors (ND). Six questions from the *p3dnd* study were selected, and the questions were presented in alternating order

[1]https://leetcode.com/problemset/all/
[2]https://codingbat.com/python



Create a class `Song` with an `__init__` method that takes a `title` as a string and `len` as a number and initializes these attributes in the current object. Then define the `__str__` method to return the `title, len`. For example, `print(s)` when `s = Song('Respect',150)` would print "Respect, 150".

Drag from here

```
1a  my.title = title
    my.len = len
or
1b  self.title = title
    self.len = len

2a  def __str__(self):
or
2b  def __str__():

3a  def __init__(title, len):
or
3b  def __init__(self, title, len):

4   class Song:

5a  return self.title + ", " + str(self.len)
or
5b  return title + ", " + str(len)
```

Drop blocks here

Solution

```
4   class Song:
3b      def __init__(self, title, len):
1b          self.title = title
            self.len = len
2a      def __str__(self):
5a          return self.title + ", " + str(self.len)
```

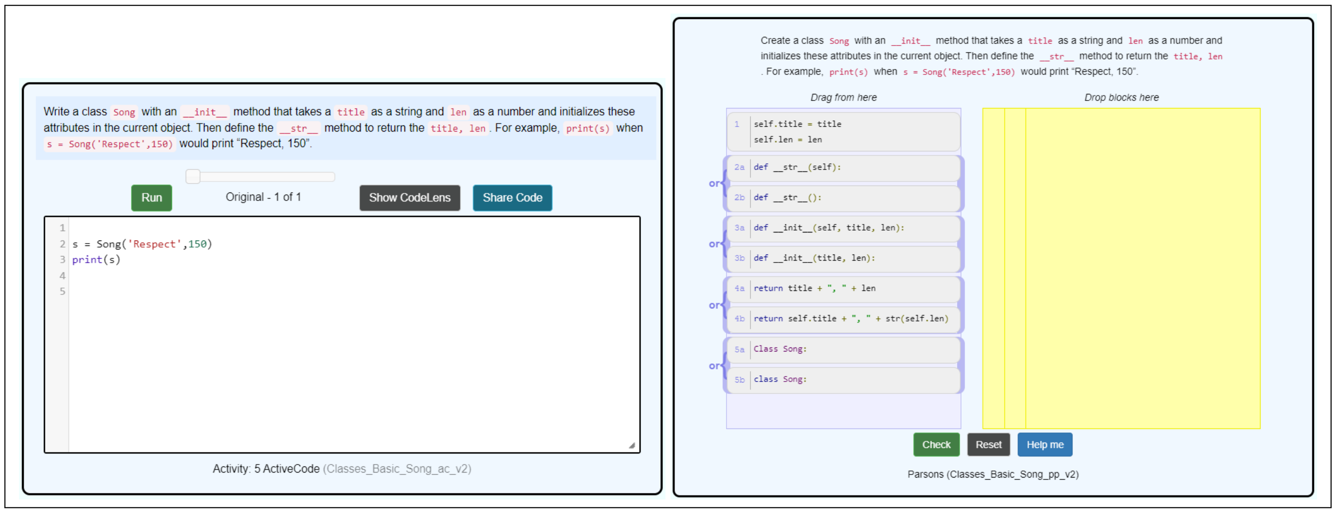
Parsons (Classes_Basic_Song_wd_pp_v4)

Create a class `Song` with an `__init__` metho
and initializes these attributes in the current obj
`title, len`. For example, `print(s)` when `s`

Drag from here

```
1  class Song:
2  def __str__(self):
3  def __init__(self, title, len):
4  self.title = title
   self.len = len
5  return self.title + ", " +
   str(self.len)
```

Figure 10: First practice problem for *class-exp* with a Parsons problem with distractors on the left, the solution in the middle, and the Parsons problem source without distractors on the right



Write a class `Song` with an `__init__` method that takes a `title` as a string and `len` as a number and initializes these attributes in the current object. Then define the `__str__` method to return the `title, len`. For example, `print(s)` when `s = Song('Respect',150)` would print "Respect, 150".

```

s = Song('Respect',150)
print(s)
```

Activity: 5 ActiveCode (Classes_Basic_Song_ac_v2)

Create a class `Song` with an `__init__` method that takes a `title` as a string and `len` as a number and initializes these attributes in the current object. Then define the `__str__` method to return the `title, len`. For example, `print(s)` when `s = Song('Respect',150)` would print "Respect, 150".

Drag from here

```
1   self.title = title
    self.len = len

2a  def __str__(self):
or
2b  def __str__():

3a  def __init__(self, title, len):
or
3b  def __init__(title, len):

4a  return title + ", " + len
or
4b  return self.title + ", " + str(self.len)

5a  Class Song:
or
5b  class Song:
```

Drop blocks here

Parsons (Classes_Basic_Song_pp_v2)

Figure 11: First practice problem in *class-tog* as a write code problem on the left and with the Parsons as scaffolding on the right

with respect to whether distractors were included in the question or not (e.g. WD, ND, ...).

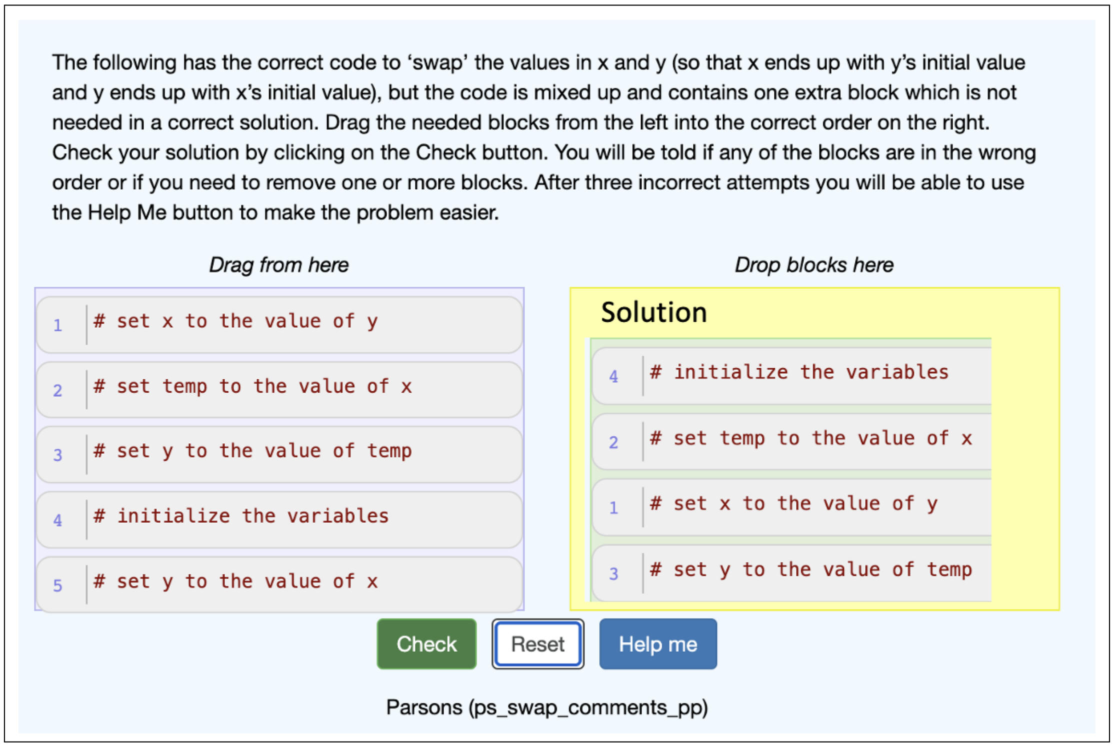
4.6.10 cdnd. Building upon *p3dnd*, we translated the study to C to be conducted at a broader set of institutions. The *cdnd* study had the same structure as *p3dnd*, described in Section 4.6.8, and the same knowledge requirements. The process of translation followed was the same as the one described in Section 4.6.7.

Name	Acronym	Country	Type	Size	Ownership
Ashesi University	ASH	Ghana	Private	2k	Private
Berea College	BEREA	USA	Private	1.6k	Private
Duke University	DUKE	USA	Research Intensive (R1)	6k	Private
Falmouth University	FALM	England	Regional	5k	Charity
Indian Institute of Technology Madras	IITM	India	Open	35k	Public
University of Chile	UCHL	Chile	Research Intensive (R1)	40k	Public
University of Illinois at Urbana-Champaign	UIUC	USA	Research Intensive (R1)	35k	Public
University of Michigan	UMICH	USA	Research Intensive (R1)	50k	Public
University of Strathclyde	USTR	Scotland	Research Intensive (R1)	26k	Public
University of Toronto	UofT	Canada	Research Intensive (R1)	97k	Public
Victoria University of Wellington	VUW	New Zealand	Open	20k	Public

Table 7: Participating Institutions

Course	Institution	Intake	Language (Spoken)	Language (Programmed)	Study Period
Introduction to Computing and Information Systems (CS0.5)	ASH	Selective	English	Python	May 2023
Software Design and Implementation (CS1)	BEREA	Selective	English	Python	April 2023
Introduction to Computer Science (CS1)	DUKE	Selective	English	Python	April 2023
Data Fundamentals (CS1.5)	FALM	Not Selective	English	Python	April 2023
Programming in Python	IITM	Qualifying Exam	English	Python	Feb - May 23
Introduction to Programming (CS1)	UCHL	Selective	Spanish	JavaScript	May - Jun 23
Introduction to Computing for Non-technical Majors (CS1)	UIUC	Selective	English	Python	Jul -Aug 23
Data-Oriented Programming (CS1.5)	UMICH	Selective	English	Python	Jan 2023
Software Development (Postgraduate Conversion)	USTR	Selective	English	Python | Java	June-Aug 23
Introduction to Computer Science II (CS2)	UofT	Selective	English	C	May - Jun 23
Programming for the Natural and Social Sciences	VUW	Not Selective	English	Python	Aug-Sep 23

Table 8: Courses in Which Studies Were Situated


The following has the correct code to 'swap' the values in x and y (so that x ends up with y's initial value and y ends up with x's initial value), but the code is mixed up and contains one extra block which is not needed in a correct solution. Drag the needed blocks from the left into the correct order on the right. Check your solution by clicking on the Check button. You will be told if any of the blocks are in the wrong order or if you need to remove one or more blocks. After three incorrect attempts you will be able to use the Help Me button to make the problem easier.

Drag from here

```
1  # set x to the value of y
2  # set temp to the value of x
3  # set y to the value of temp
4  # initialize the variables
5  # set y to the value of x
```

Drop blocks here

Solution

```
4  # initialize the variables
2  # set temp to the value of x
1  # set x to the value of y
3  # set y to the value of temp
```

Parsons (ps_swap_comments_pp)

Figure 12: First practice problem in *python-swap* with pseudocode comment blocks that explain the algorithm

5 2023 WORKING GROUP STUDY CONTEXTS

In this section, we describe the context of each of the 13 institutions that participated in the studies. We provide a description about the institutions, courses, and student demographics.

As part of the onboarding process to the Working Group, the organizers provided all members with a replication package. The replication package included the study procedure, study descriptions, study materials and sample IRBs. Each institution had to follow their own institutional rules in filing their own IRB. To assure that our data collection is consistent, our studies were developed to use one platform, Runestone Academy [31, 64], to conduct the studies, and this is the platform that was used with the exception of one institution where the language of instruction was not English. This institution used a locally developed system instead.

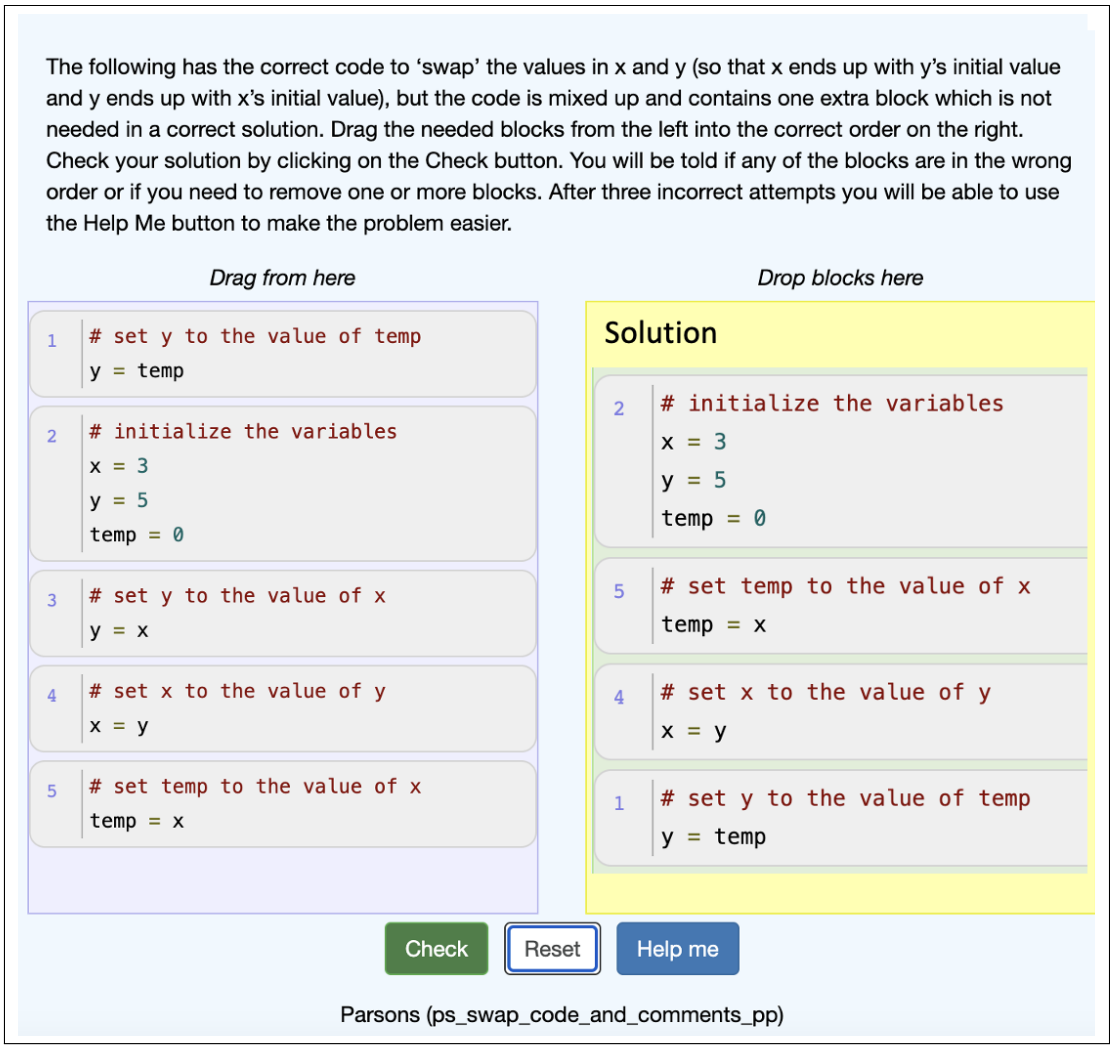
5.1 Ashesi University in Ghana

Ashesi University is a small English-speaking liberal arts university in Ghana. Ashesi draws students from across Africa and beyond, but international students for whom English was not a language of instruction during high school submit evidence of English language proficiency in order to be admitted. Over forty percent of Ashesi's students are on scholarship.

The studies were conducted as portions of a single homework assignment in all six sections of an in-person introductory information systems and computing course. This course is required for first-term freshmen students pursuing any of three majors: Business Administration, Computer Science, and Management Information Systems. Gender balance in the course is roughly evenly split between males and females. For the vast majority of students, this course is their first programming course, but all students in the studies had familiarity with Parsons problems prior to the studies because the *python-swap* and *p3pt* studies were delivered as a single graded homework assignment via the Runestone textbook they had used the entire term. Credit was given for participation if they spent at least at least 5 minutes on any problem that was not correctly solved. Of the 290 students in the six course sections, 268 submitted some portion of the work, and 212 consented to have their data included in the studies. The data that was analyzed includes only the data from these 212 students.

5.2 Berea College in the USA

Berea College is a small English-speaking liberal arts college in Kentucky that solely serves economically disadvantaged students. Berea College is also one of nine federally recognized work colleges, so all Berea College students work at least 10 hours per week for the institution. The computer and information science (CIS) major is one of the largest majors at the college with approximately 25% Female-identifying, 20% African-American, 45% other domestic, and 35% international. The *class-exp* study was conducted as a graded by participation only assignment in an in-person introductory computing (CS1) course which serves as the first required course in the CIS major, often following a CS 0.5 course. All students in the course had seen Parsons problems prior to the study because Parsons problems are utilized in their regular Runestone textbook, and the *class-exp* study was delivered via this textbook.



The following has the correct code to 'swap' the values in x and y (so that x ends up with y's initial value and y ends up with x's initial value), but the code is mixed up and contains one extra block which is not needed in a correct solution. Drag the needed blocks from the left into the correct order on the right. Check your solution by clicking on the Check button. You will be told if any of the blocks are in the wrong order or if you need to remove one or more blocks. After three incorrect attempts you will be able to use the Help Me button to make the problem easier.

Drag from here

```
1  # set y to the value of temp
   y = temp
```

```
2  # initialize the variables
   x = 3
   y = 5
   temp = 0
```

```
3  # set y to the value of x
   y = x
```

```
4  # set x to the value of y
   x = y
```

```
5  # set temp to the value of x
   temp = x
```

Drop blocks here

Solution

```
2  # initialize the variables
   x = 3
   y = 5
   temp = 0
```

```
5  # set temp to the value of x
   temp = x
```

```
4  # set x to the value of y
   x = y
```

```
1  # set y to the value of temp
   y = temp
```

Parsons (ps_swap_code_and_comments_pp)

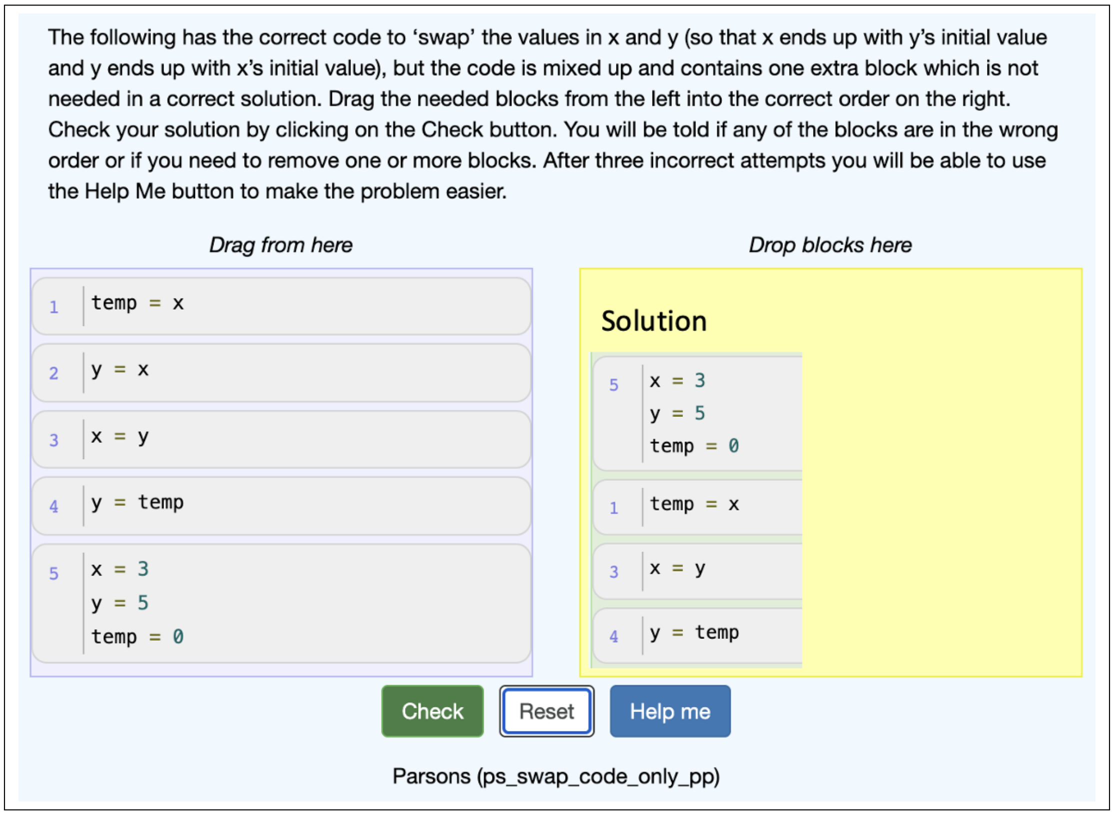
Figure 13: Second practice problem in python-swap with pseudocode comments plus code in each block

Of the 31 students in the course, only the data from the 25 who consented to have their data used in research was analyzed, and only the data from the 14 who completed the components was able to be used in the final analysis. A pool of 17 students were recruited for think aloud observation by two Berea College professors. Three of this pool were selected by meeting-time convenience for think aloud observational studies. Then three observational studies were conducted via Microsoft Teams. Although these three students volunteered for these observations, they were paid for their time through the college work program, so they earned $9.50 per hour for their time.

5.3 Duke University in the USA

Duke University is an English-speaking liberal arts private institution in Durham, North Carolina, USA. We ran a study in the course Introduction to Computer Science (CompSci 101), a beginning programming course in Python. 80% of the students in this course have never programmed before or have had little programming experience. This is the first programming course for majors, but the course is also taken by many non-majors. This course typically has 200-300 students each semester, mostly in the age range of 18-20 years old, with approximately 50% female students. As a beginner course, the course covers Python basics including variables, conditionals, repetition, lists, tuples, sets, sorting, lambda functions and dictionaries. The course has two lectures and one lab each week, both are taught in person, though labs can be completed online for those students who are absent. The course uses the online Runestone textbook *How To Think Like a Computer Scientist - Learning with Python: Interactive Edition*. Before each lecture, students are assigned reading from this textbook and they must answer quiz questions related to the reading before attending lecture.

In the Spring 2023 semester, CompSci 101 had 217 students enrolled in the course. We ran the study on *class-exp* in April 2023. Students in CompSci 101 have used class methods such as append



The following has the correct code to 'swap' the values in x and y (so that x ends up with y's initial value and y ends up with x's initial value), but the code is mixed up and contains one extra block which is not needed in a correct solution. Drag the needed blocks from the left into the correct order on the right. Check your solution by clicking on the Check button. You will be told if any of the blocks are in the wrong order or if you need to remove one or more blocks. After three incorrect attempts you will be able to use the Help Me button to make the problem easier.

Drag from here

```
1  temp = x

2  y = x

3  x = y

4  y = temp

5  x = 3
   y = 5
   temp = 0
```

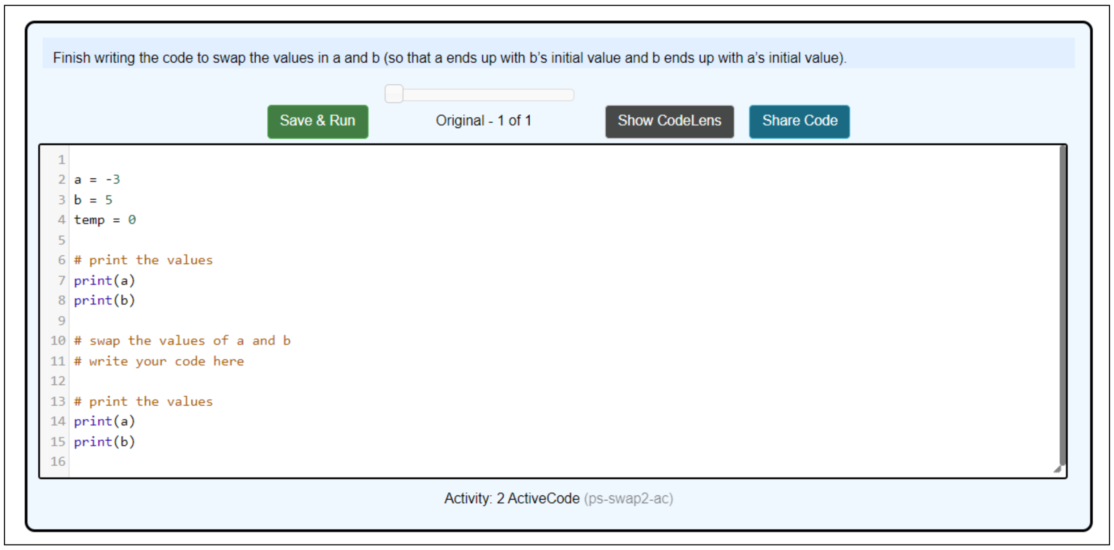
Drop blocks here

Solution

```
5  x = 3
   y = 5
   temp = 0

1  temp = x

3  x = y

4  y = temp
```

Parsons (ps_swap_code_only_pp)

Figure 14: Third practice problem for *python-swap* with just code in each block



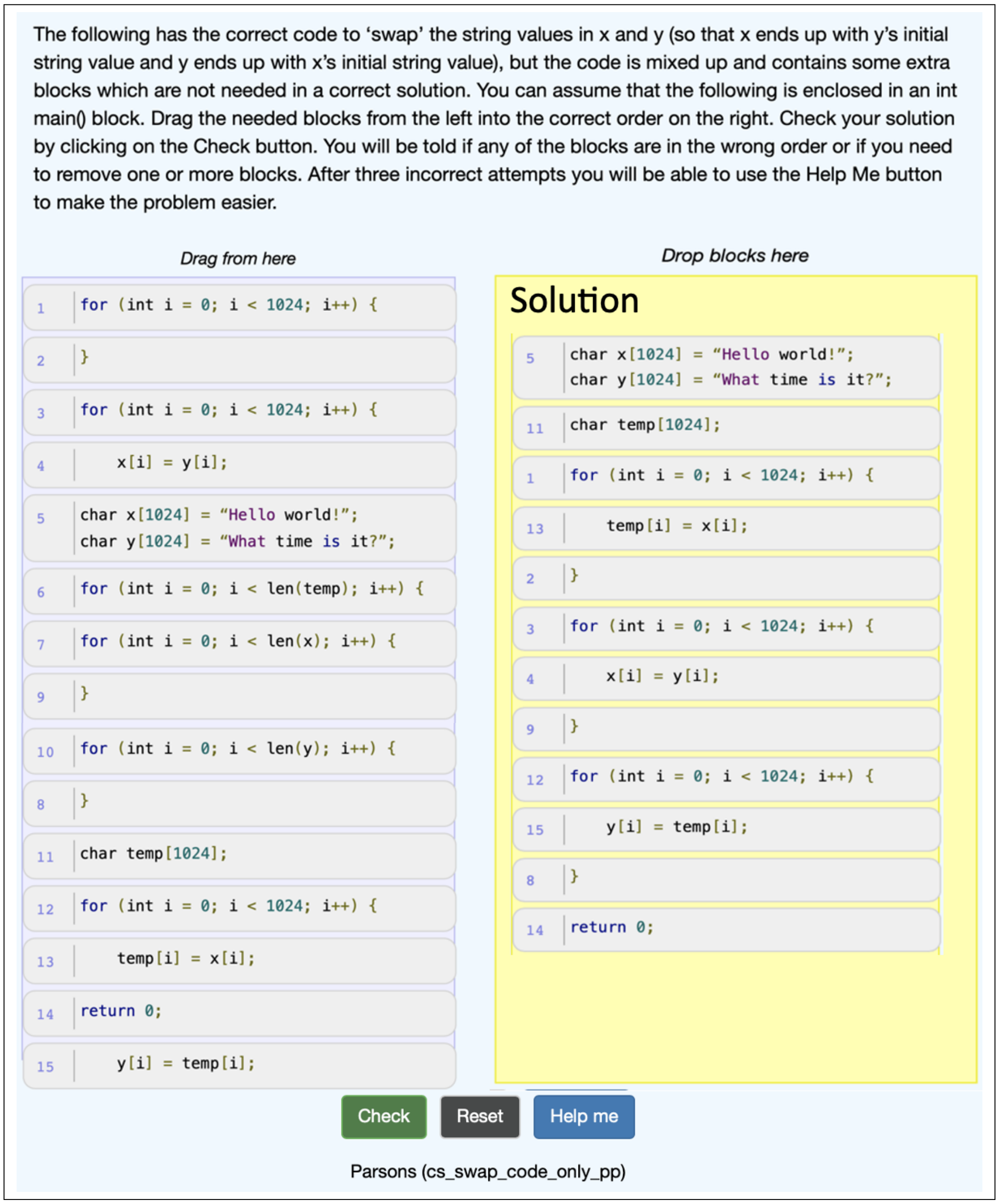
Finish writing the code to swap the values in a and b (so that a ends up with b's initial value and b ends up with a's initial value).

```

a = -3
b = 5
temp = 0

# print the values
print(a)
print(b)

# swap the values of a and b
# write your code here

# print the values
print(a)
print(b)

```

Activity: 2 ActiveCode (ps-swap2-ac)

Figure 15: The second write code problem in the posttest for *python-swap*

for lists, but the students have never seen a full class before attempting the study. The students have had some familiarity with Parsons problems as there are a few in the online textbook for the course. An IRB was applied for in January and approved in March 2023. The study was held as a complete lab, was graded (by participation), and was required to complete by all students. Each student was emailed an anonymous email and password to use in the study. They then logged into Runestone to complete the lab



The following has the correct code to 'swap' the string values in x and y (so that x ends up with y's initial string value and y ends up with x's initial string value), but the code is mixed up and contains some extra blocks which are not needed in a correct solution. You can assume that the following is enclosed in an int main() block. Drag the needed blocks from the left into the correct order on the right. Check your solution by clicking on the Check button. You will be told if any of the blocks are in the wrong order or if you need to remove one or more blocks. After three incorrect attempts you will be able to use the Help Me button to make the problem easier.

Drag from here

```
1   for (int i = 0; i < 1024; i++) {
2   }
3   for (int i = 0; i < 1024; i++) {
4       x[i] = y[i];
5   char x[1024] = “Hello world!”;
    char y[1024] = “What time is it?”;
6   for (int i = 0; i < len(temp); i++) {
7   for (int i = 0; i < len(x); i++) {
9   }
10  for (int i = 0; i < len(y); i++) {
8   }
11  char temp[1024];
12  for (int i = 0; i < 1024; i++) {
13      temp[i] = x[i];
14  return 0;
15      y[i] = temp[i];
```

Drop blocks here

Solution

```
5   char x[1024] = “Hello world!”;
    char y[1024] = “What time is it?”;
11  char temp[1024];
1   for (int i = 0; i < 1024; i++) {
13      temp[i] = x[i];
2   }
3   for (int i = 0; i < 1024; i++) {
4       x[i] = y[i];
9   }
12  for (int i = 0; i < 1024; i++) {
15      y[i] = temp[i];
8   }
14  return 0;
```

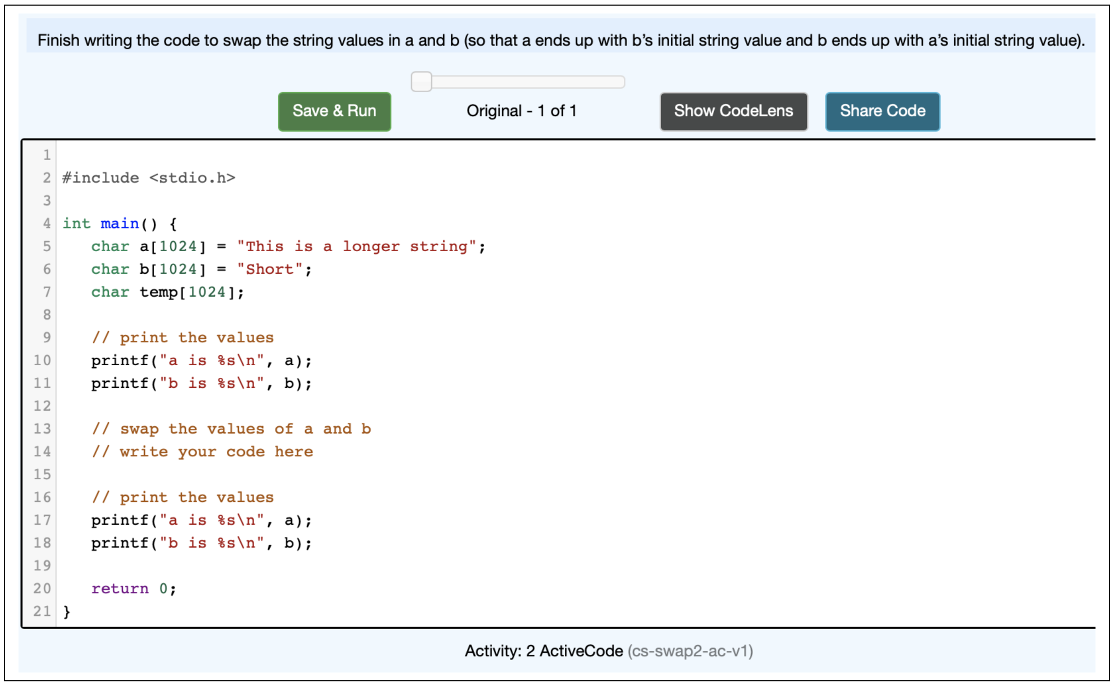
Parsons (cs_swap_code_only_pp)

Figure 16: Third practice problem for swapping the values of two variables with just code in each block in *c-swap*

online, and were instructed to take about 60 minutes for the lab, sometime during a four day period April 6-9. An email was sent out during the four day period to ask students to complete a short Qualtrix survey to consent or not to using their data in the study. 130 students consented to use their data, about 60% of the class.

5.4 Falmouth University in Cornwall, UK

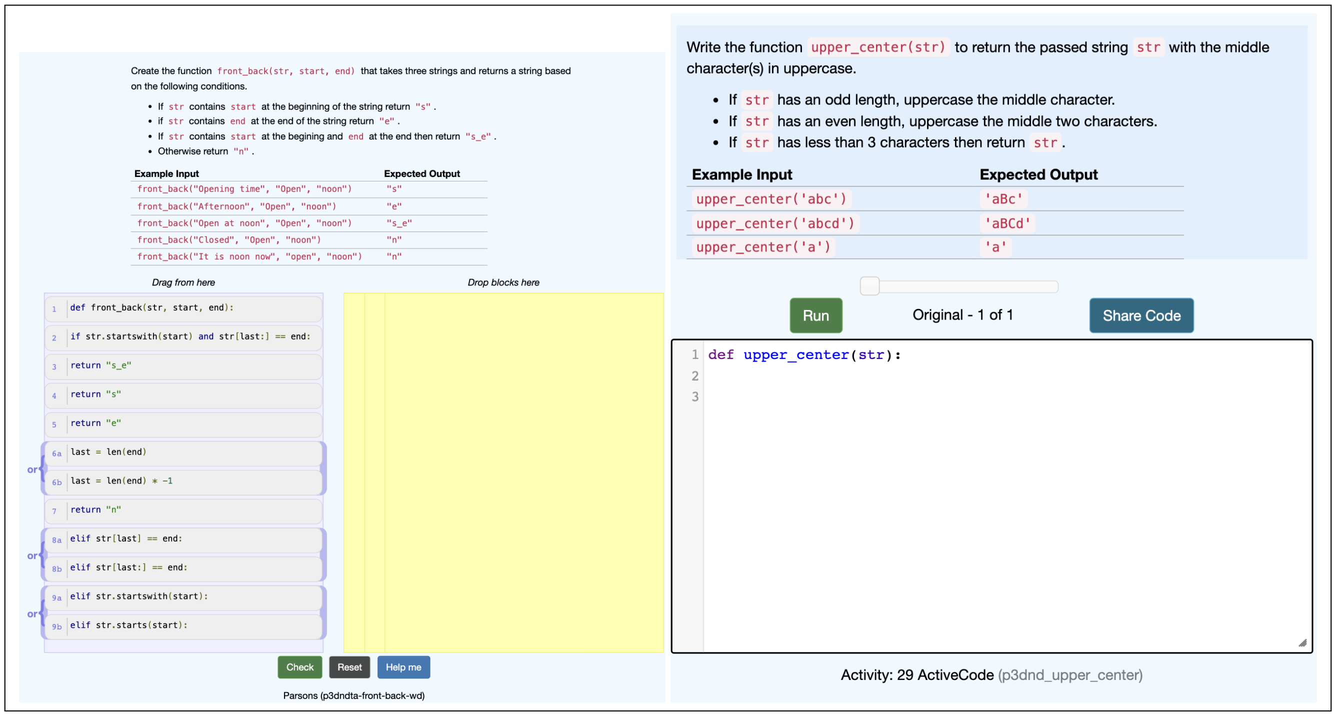
Falmouth University is an English-speaking institution in the United Kingdom, located in Cornwall, England. The studies were conducted in the Games Academy, which is a multi-disciplinary department of about 1000 students within the Faculty of Screen, Technology, and Performance. It offers many different courses to enable its students to come together in teams to make games. In addition to degrees in


Finish writing the code to swap the string values in a and b (so that a ends up with b's initial string value and b ends up with a's initial string value).

```c

#include <stdio.h>

int main() {
    char a[1024] = "This is a longer string";
    char b[1024] = "Short";
    char temp[1024];

    // print the values
    printf("a is %s\n", a);
    printf("b is %s\n", b);

    // swap the values of a and b
    // write your code here

    // print the values
    printf("a is %s\n", a);
    printf("b is %s\n", b);

    return 0;
}
```

Activity: 2 ActiveCode (cs-swap2-ac-v1)

Figure 17: The second write code problem in the posttest in *c-swap*


Create the function `front_back(str, start, end)` that takes three strings and returns a string based on the following conditions.

- If `str` contains `start` at the beginning of the string return `"s"` .
- if `str` contains `end` at the end of the string return `"e"` .
- If `str` contains `start` at the begining and `end` at the end then return `"s_e"` .
- Otherwise return `"n"` .

Example Input	Expected Output
`front_back("Opening time", "Open", "noon")`	`"s"`
`front_back("Afternoon", "Open", "noon")`	`"e"`
`front_back("Open at noon", "Open", "noon")`	`"s_e"`
`front_back("Closed", "Open", "noon")`	`"n"`
`front_back("It is noon now", "open", "noon")`	`"n"`

```
1   def front_back(str, start, end):
2   if str.startswith(start) and str[last:] == end:
3   return "s_e"
4   return "s"
5   return "e"
6a  last = len(end)
 or
6b  last = len(end) * -1
7   return "n"
8a  elif str[last] == end:
 or
8b  elif str[last:] == end:
9a  elif str.startswith(start):
 or
9b  elif str.starts(start):
```

Parsons (p3dndta-front-back-wd)

Write the function `upper_center(str)` to return the passed string `str` with the middle character(s) in uppercase.

- If `str` has an odd length, uppercase the middle character.
- If `str` has an even length, uppercase the middle two characters.
- If `str` has less than 3 characters then return `str` .

Example Input	Expected Output
`upper_center('abc')`	`'aBc'`
`upper_center('abcd')`	`'aBCd'`
`upper_center('a')`	`'a'`

```python
def upper_center(str):


```

Activity: 29 ActiveCode (p3dnd_upper_center)

Figure 18: First practice problem in *p3dnd* as a Parsons problem with distractors on the left and without on the right

Game Development and Game Programming, it also offers a range of degrees including Computer Science, Esports, Immersive Computing, and Robotics. The students are mostly domestic (88%), with a small number coming from the European Union (9%) or further afield (3%). The cohort mostly identifies as male (90%), with a minority identifying as female (8%) and non-binary (2%). Nearly one third

Course	Institution	Study Period	Enrolled Students	Participating Students	Voluntary	Studies
Introduction to Computing and Information Systems (CS0.5)	ASH	May 2023	290	212 | 212	Required	*python-swap*, *p3pt*
Software Design and Implementation (CS1)	BEREA	April 2023	31	25 | 2 | 1	Required	*class-exp*, *class-ta*, *p3pt-ta*
Introduction to Computer Science (CS1)	DUKE	April 2023	217	130	Required	*class-exp*
Data Fundamentals (CS1.5)	FALM	April 2023	85	32	Voluntary	*python-swap*, *p3pt*, *classexp*
Programming in Python	IITM	Feb - May 23	1632	50	Voluntary	*p3pt*
Introduction to Programming (CS1)	UCHL	May - Jun 23	35	35	Required	*jspt*
Introduction to Computing for Non-technical Majors (CS1)	UIUC	Jul -Aug 23	989	5	Voluntary	*python-swap*
Data-Oriented Programming (CS1.5)	UMICH	Jan 2023	191	155	Required	*class-exp*
Software Development (Postgraduate Conversion) and Introduction to Programming with Python (up-skilling)	USTR	June-Aug 23	86	6	Voluntary	*python-swap*
Introduction to Computer Science II (CS2)	UofT	May - Jun 23	150	51 | 67	Incentivised	*c-swap*, *p3-dnd*
Programming for the Natural and Social Sciences	VUW	Aug-Sep 23	169	7	Voluntary	*p3pt*

Table 9: Student Numbers and Studies

of the cohort declares a disability (29%) with a considerable number of these students declaring some form of neurodiversity. There is a low proportion of black, asian, and ethnic minority students in the Games Academy (6.5%).

The participants are students in their first year of study (if on our three-year programme) or second year (if on our four-year programme with an integrated foundation year). These students take six modules each academic year, which consist of 200 notional hours of study which are related through a shared set of intended learning outcomes. The *python-swap*, *p3pt*, and *class-exp* studies were situated in the 'Data Fundamentals' module, which were delivered between February and May in 2023. These modules help the students to learn to program in Python ahead of a syllabus focused on data analysis and academic report writing. The module typically enrolls around 80-100 students, half of which tend to have little to no prior programming experience. These students won't have encountered Parsons problems or open-source ebooks yet in their studies. The studies were integrated into the syllabus, which had the students complete the exercises on the Runestone platform during a series of timetabled synchronous one-hour, online distance-learning sessions led by two instructors and facilitated by Microsoft Teams. Participation was optional, with no associated grading, but strongly encouraged and presented in the same manner as a learning activity in any other workshop. 32 students consented to use their data.

5.5 Indian Institute of Technology Madras, India

Indian Institute of Technology Madras is a premier science and technology institution located in the city of Chennai, India. The English-speaking institute has recently started a BS Programme in Data Science and Applications that is delivered primarily online with in-person assessments. The curriculum is split into three levels - foundation, diploma (skills), and degree (specialization). Admission is open to anyone with K-12 education in any stream with an in-built qualification process to assess suitability. The current study is conducted with the students in an Introduction to Python course (one of the foundational courses) which covers conditionals, loops, functions, data structures, basics of file handling, and object-oriented programming. The medium of instruction is English and the students have two additional English courses as part of the programme to make them comfortable with the language of instruction. The course is offered three times a year, and the current set of studies focuses on the students from the January - April 2023 batch. The cohort of students targeted for the *p3pt* study were repeating the course as they failed to clear a mandatory programming exam in the previous offering of the same course (September - December, 2022). There were a total of 1632 students with a female to male ratio of 30:70 and an age-range of 17 to 68 years. More than 40% of the students in this group have not had a prior experience to programming language, however all of the students have cleared a course on Computational Thinking which is a prerequisite for the Python course. None of these students have prior exposure to Parsons problems even though their regular assessments have questions that involve identifying missing blocks of code or identifying lines of code that are having either syntax or logical errors or predicting output of a piece of code.

All the students were added to the Runestone platform by the course instructor and they were explained about the study as part of a synchronous session. The students who did not attend the live session were provided with a recording of the session. The students were provided a week (from the start of their attempt) to complete the activity, as their programming examinations were scheduled in the subsequent week. However this was not strictly enforced as the study was a voluntary activity and was recommended as the first activity to be done when they are revising Python course. A total of 152 students consented to the use of their data for this study.

We faced the following challenges during the execution of the study

- Since the course is delivered completely online, the biggest challenge was in conveying the information to the students. The information about the study was sent via asynchronous mechanisms (Emails and WhatsApp Notifications) and more often students either missed reading them or ignored them completely.
- The second biggest challenge was to allow students to familiarize with the Runestone platform. While the platform contained video and description about how to use it, the students were more often confused about the sequence of actions to be taken as part of the study. This required the instructors to setup multiple sessions to explain the flow of the pages in the Runestone platform for the study.
- We had configured another Runestone book to allow the students to learn and practice at their own pace before the programming exams. However, the requirement to access different URLs were a deterrent for students to take up the activity.
- As the participation to the study was voluntary, the students' engagement with the various sections within the Runestone notebook was not consistent. Many either skipped intermediate pages and directly attempted the post test or dropped off from the study after scanning through the initial activities.

We attempted to conduct a repeat of this study with the students from the May-August 2023 batch, however the asynchronous communication channel challenge resulted in only 7 students (out of 1639 contacted) joining the initial interaction session. Though we tried to re-schedule the session in the following weeks, the participation trends were similar and finally we had to drop the study completely.

5.6 Victoria University of Wellington in New Zealand

The Victoria University of Wellington (VUW) is English-speaking institution, with the studies commencing in July 2023 (Semester 2) as non-compulsory activities. The course delivery is in-person and using Python. This institution typically have traditional student cohorts in first-year programming courses. Recruitment was done by a representative from the research group, speaking of the study in the lecture a week before the start of the study. The same researcher sent out three emails to encourage participation: two emails were sent before the start of the study and one reminder during the two-week period the study was open for participation.

At VUW, we conducted the study in one first-year course for Natural and Social Sciences majors, teaching programming fundamentals that perform basic operations on data sets, such as processing, transforming, analyzing, and presenting data. This course is designed for students with no background in programming. For this course, we had the student participate in the *p3pt* study. This course had 169 enrolled students, with seven (4%) students consenting to their data used in the study. We did not determine the reasons for the low student participation rate and we did not follow-up with them to determine the cause. Further work is required to understand the low participation from this institution.

5.7 University of Chile in Chile

We conducted *jspt* with a sample of 35 non-professional students (aged between 24 to 55 years old, 70%-30% male-female gender distribution) enrolled in an online bootcamp, who took an introductory course on computational thinking and programming. These students did not have any sort of formal background in STEM-related fields, particularly in computer science.

In this group, students were exposed to Parsons problems as an explicit scaffolding strategy of instruction. However, these Parsons problems were not adaptive. The study took place in the form of a hands-on practical session assisted virtually by a team of trained teaching assistants; therefore, participation was required (although not graded). Consequently, the running experiment was conducted as a required assignment during a lab session after the notions of unit testing, conditionals, iteration (for and while loops), lists, and strings were all covered in lectures. Due to the difference in context setting, besides collecting quantitative measures of performance (such as the number of tries or average time spent in producing a correct answer), we conducted exit interviews aiming to better understand how Parsons problems could effectively provide scaffolding to (very) novice programmers when exposed to writing code.

Given that in the University of Chile there is no formal requirement of mastering English as a foreign language at the undergraduate level, the study was run in Spanish. Resonating with the challenges of conducting MIMN studies previously identified by McCracken et al. [63], several procedures were followed to ensure that the studies were correctly translated into the target language.

5.8 University of Illinois at Urbana-Champaign in the USA

The University of Illinois at Urbana-Champaign (UIUC) is an English-speaking institution located primarily in Urbana, Illinois. The student population for these studies is from an introductory Python course that is specifically for students from non-technical majors who typically have limited prior experience with programming. The class covers the basics of programming in Python in addition to file manipulation and an introduction to building classes. Beyond Python, it covers the basics of HTML and several topics in Microsoft Excel. It has historically consisted of primarily freshmen and sophomores (18-20) and has a roughly even split between men and women. Students are familiar with Parsons problems as they are used on both formative and summative assessments in the course. We will conducted think-aloud interviews with five students to provide a qualitative lens in answering the research questions associated with each of these studies. Students were recruited via email from past semesters in the course. Participation in these interviews was completely voluntary and students are compensated at a rate of $15 dollars an hour. Interviews are to be conducted online in a recorded Zoom session wherein participants were asked to share their screen while they complete the problems.

5.9 University of Michigan in the USA

The University of Michigan in the USA is an English-speaking research-intensive institution in Ann Arbor, Michigan, USA. We ran an early version of *class-exp* with four practice problems and four post-test problems in a second required programming course for School of Information majors in the winter semester of 2023, which runs from January to April. This course had 191 students and is usually about 40% female but does not have a high percentage of people from minoritzed groups. It covers Python basics, object-oriented programming basics, regular expressions, unit tests, debugging, and working with data from files, websites, APIs and databases. Students are familiar with Parsons problems as they are used in interactive readings and as active learning assignments in lecture. We ran the study during lecture in the first month of the course and 155 students completed the study. Students received points for attempting the study, they did not have to get the problems correct to earn the points. Based on the results from this study, which did not find any significant difference between conditions on learning performance, we added another practice problem and post-test problem to *class-exp*.

5.10 University of Toronto in Canada

The University of Toronto (UofT) is an English-speaking institution with three campuses located in Toronto, Canada. Our studies were conducted in one of the three campuses in a CS2 course during the Summer 2023 semester. The course was delivered in-person and using the C language. CS2 is a mandatory course for students wishing to pursue a computer science program, however, some students from other departments (management, neuroscience, statistics) can take it as an elective. In order to be enrolled in the course, students had to successfully complete CS1, which is delivered in Python. During the Summer semester, 70% of students were in the computer science program, and 75% of the students were in their first year of studies. Students ranged from 18 to 22 years old, where 60% identified as men and 24% identified as women. UofT is known for its diverse and multi-cultural population, where only a third of students are domestic. Around 5% of students mentioned that their levels of understanding spoken and written English is below average.

In this course, students learn about C syntax, the memory model, pointers, linked lists, abstraction, graphs, and recursion. At the beginning of the semester, students were informed about two bonus marks opportunities that would come up during the semester as online activities. The completion of each study rewarded students with one bonus mark for their midterm test. To recruit students for each study, the course instructor made an announcement in the course's discussion forum. Students were given one full week to complete each study. The first study, *c-swap*, was conducted after the third week of the course after students got introduced to how strings work in C. The second study, *c3-dnd*, was conducted after week 6 of the course when students have seen strings, conditionals, memory model, loops, functions, structs and compound data types in C. For the first and second studies, we collected 51 and 67 responses respectively.

5.11 University of Strathclyde in Scotland

The University of Strathclyde is an English-speaking institution in Glasgow, Scotland, UK. We ran non-compulsory think aloud observations with *python-swap* that acted as an assessment alternative in an introductory Python programming module offered fully online and asynchronously by the Department of Computer and Information Sciences as part of the University's Upskilling Programme, which was delivered from January to August. This module had 35 learners, and it covered Python basics, iteration, conditions, unit testing, and basic data types and structures, but no object-orientation. Learners were not familiar with Parsons problems. Demographically, the cohort included learners aged from 26 to 50, with a gender balance split of 65% males and 35% females, and a split of 13% international and 87% home learners. We also ran *python-swap* as non-compulsory think aloud observations by recruiting volunteers from the Department's MSc in Software Development conversion course, which ran (on-campus, synchronously) from September to August. This course had 51 students, and it covered both Python and Java: basics, iteration, conditions, unit testing, inheritance, library classes, and APIs, polymorphism, and basic data types and structures. Students were not familiar with Parsons problems. Demographically, the cohort included students aged from 22 to 60, with a gender balance split of 67% males and 33% females, and a split of 35% international and 65% home students. For both cohorts, the majority of students joined the courses with no prior knowledge of programming, and all international students had a command of the English language equivalent to IELTS 6.0 or higher, both spoken and written, as this is an entry requirement for our courses. All think aloud observations were conducted over Zoom on a one-to-one basis, and participants were recruited by emailing the relevant students using their cohort-specific emailing lists.

6 RESULTS

This section describes the results from our studies. Section 6.1 presents the findings of our think alouds while Section 6.2 describes our findings running the studies in computing courses at many multi-national institutions.

6.1 Think Aloud Observations

We conducted think aloud observations at four institutions: Berea College, the University of Illinois, the University of Strathclyde,

Interview ID	Generally I have felt secure about attempting computer programming problems.	I am sure I could do advanced work in computer science.	I am sure that I can learn programming.	I think I could handle more difficult programming problems.	I can get good grades in computer science.	I have a lot of self-confidence when it comes to programming.
class-ta 1	1	4	4	5	4	4
class-ta 2	5	5	5	5	5	3
p3dnd-ta	2	4	4	3	5	3
python-swap-1	4	2	4	4	4	3
python-swap-2	5	4	5	4	5	4
python-swap-3	5	3	5	4	5	3
python-swap-4	3	2	4	3	4	2
python-swap-5	4	2	4	3	4	4
python-swap-6	3	3	5	3	4	4
python-swap-7	4	3	5	3	4	2
python-swap-8	1	1	4	4	2	2
python-swap-9	4	2	5	4	4	4
python-swap-10	4	3	5	4	4	4
python-swap-11	4	4	4	N/A	4	4
jspt-1	1	2	3	3	2	2
jspt-2	4	5	5	4	5	5
jspt-3	3	4	4	4	4	4
jspt-4	3	4	4	4	4	4
jspt-5	4	4	5	4	4	4
jspt-6	2	2	2	2	2	2
jspt-7	2	2	4	2	4	3
jspt-8	3	2	4	2	4	2
jspt-9	4	4	5	5	5	4
jspt-10	2	1	5	2	2	1
jspt-11	4	4	5	5	5	5
jspt-12	3	3	4	4	4	4

Table 10: Responses to the general pre-survey questions for all think-aloud studies. Responses were collected on a 5-point Likert scale (1) strongly disagree to (5) strongly agree.

and the University of Chile. In the following descriptions of the think aloud observations we do not specify the institution in order to protect the anonymity of the participants. However, we describe the participant in order to provide the context, if the institution's ethics (IRB) approval allowed us to share this information. Each think aloud was a video-conferencing session that we recorded and transcribed for analysis.

In this section, we describe noteworthy points from the think aloud observations to better understand the student experience, perceptions, and thinking processes. We also explain how the Parsons problems and the assessment tool's user interface either helped the participants improve their understanding of the problem or generated challenges for them when solving a problem. By reporting on the think aloud observations, we add a qualitative dimension to the associated quantitative studies, enabling us to go deeper into how the Parsons problems support the learning process. For the remainder of this section, we present the results of the Parsons problem studies. Section 6.1.1 presents the think aloud results from the *python-swap* study. Section 6.1.2 describes the *class-ta* study results, while Section 6.1.3 presents the *p3dndta* study. We conclude with Section 6.1.4, discussing the results of the *jspt* study. We provide a detailed description of the think aloud studies in Section 4.6.

6.1.1 Think Aloud Observations: python-swap. This study had students learn about the swap algorithm by solving three Parsons problems 1) the first with just pseudocode comments that described the algorithm, 2) the second with pseudocode comments and code, and 3) the third with just code. A total of 11 students participated in think aloud observations.

One of the participants (i.e. python-swap-6) exhibited difficulties completing the "Introduction to Problem Types" section for

Student	Setting the value of a variable like: x = 4	Swapping the values of two variables so that var1 has the original value of var2 and var2 has the original value of var1
python-swap-1	4	3
python-swap-2	4	4
python-swap-3	4	3
python-swap-4	4	4
python-swap-5	4	2
python-swap-6	4	4
python-swap-7	4	4
python-swap-8	3	3
python-swap-9	4	3
python-swap-10	4	4
python-swap-11	4	4

Table 11: Responses to questions on the *python-swap* presurvey asking students to rate their familiarity with the concepts of setting variables and swapping. Responses were collected on a four-point Likert scale (1) strongly disagree to (4) strongly agree.

introducing the Parsons problems and code writing problems, as described in Section 4.3. In particular, this participant began strong in completing the first problem type, which involved arranging blocks without indentation, in a single attempt. However, when beginning the second problem type, which involved arranging and indenting blocks, the participant faced several challenges which might be attributed to two main factors: lack of experience with the UI of the Runestone platform and/or a lack of familiarity with the concept of indentation in Python. Regarding the first factor, the participant's first attempt involved adding in only one (i.e. "First block") of the required three blocks using indentation (not required for this block), and clicked the "Check" button. Regarding the second factor, upon reading the feedback after the first failed attempt the participant used all three required blocks in the correct order,

	Parsons problem 1	Parsons problem 2	Parsons problem 3	Write Code 1	Write Code 2
python-swap-1	5	2	1	1	1
python-swap-2	4	1	1	1	1
python-swap-3	1	1	1	1	1
python-swap-4	5	2	2	5	1
python-swap-5	3	1	1	1	1
python-swap-6	9	4	1	8	1
python-swap-7	1	1	1	1	1
python-swap-8	2	1	1	1	1
python-swap-9	6	2	2	2	1
python-swap-10	1	1	1	1	1
python-swap-11	2	1	1	1	1

Table 12: The problems in the order students were presented them during the think aloud interviews for *python-swap* and the number of attempt on each problem. One trend that emerged was that, for students with a higher number of attempts on the first Parsons problem, the number of attempts on the subsequent two decreased, and these students were very successful on subsequent code-writing tasks.

but failed to indent the final block, and most likely ignored the feedback on how to fix the issue with indentation. This was followed by a series of failed attempts, with the participant using either two of the three required blocks and indentation in one attempt or all three required blocks and indentation, but with the wrong block ordering and/or wrong indentation, in four attempts. In particular, during these five failed attempts: the interviewer prompted the participant to use the "Help me" button but upon reading the generated feedback the participant experienced another failed attempt, which was followed by the interviewer's further advice to read the generated feedback carefully and also think about what "indentation" meant. Upon receiving this advice, the participant started to realise that the third block had to be indented. However, they placed the third block at the top, with the other two blocks intended below. Despite these initial difficulties with this problem type, the participant completed the third problem type, which also involved indentation, in a single attempt. Although this may be an indication that the second problem type was successful in teaching the participant the concept of indentation and/or the indentation element of the UI, the degree of difficulty faced by the participant in overcoming this initial challenge was still large. There was a video that demonstrated how to solve a Parsons problem that did showed how to indent the lines. However, that video didn't show the type of feedback that is displayed when just the indentation is wrong.

Before completing the think aloud observations, all participants indicated that they were comfortable with setting a variable equal to a value. However, there was mixed stated familiarity with the concept of swapping variables. We observed three common themes from the participants: confusion over the role of the `temp` variable, an increase in understanding of the swap algorithm, and difficulty organizing the pseudocode comment blocks. We describe each of the three themes further.

Confusion over the role of the `temp` *variable:* In completing the practice activities, three participants indicated confusion over the role the `temp` variable has in the swap algorithm. For example, participant **python-swap-1** claimed the problem could be solved with the following two Python statements:

```
x = y
y = x
```

After five attempts on the first Parsons problem (Table 12), the participant completed it and used the solution from this problem to solve the rest of the Parsons problems since they were all on the same page. Unfortunately, the participant remained confused over the role of the `temp` variable. During the first code-writing activity, the participant attempted to solve it without using the `temp` variable, but that did not work. The participant next used the CodeLens tracing tool and realized that their mental model of the solution was incorrect. However, they were unable to recall the exact solution from the Parsons problems. Instead they created a solution that used two `temp` variables:

```
temp1 = x
temp2 = y
y = temp1
x = temp1
```

Another participant, **python-swap-4**, also struggled to recall the Parsons problem solution while writing the code. After the think aloud, the interviewer asked the participant **python-swap-4** to reflect on the Parsons problem and code-writing activities. Below is a portion of the participant's self-reflection, stating:

> **python-swap-4** - I just felt really annoyed because I wanted to do it how the Parsons problems were because I feel like that's what the whole point was, was that I learned it in Parsons and then replicate that solution. But because it wasn't making sense to me before and then I forgot everything that I had done.

For participant **python-swap-4**, the issue appeared to be that they could not resolve their initial misconception that swapping could be performed without using a `temp` variable. However, it is noteworthy that they resolved this misconception in the code-writing activity by recalling the `temp` variable from the Parsons problems.

Participant **python-swap-6** also struggled to solve Parsons problem 1 due to their initial misconception that swapping could be performed by replacing the value of `x` by `y` (`x = y`) before the use of the `temp` variable. The participant demonstrated a similar misconception in the first write code problem by trying to solve it in a single line without using the `temp` variable, i.e. `x = y`. After the interviewer observed several incorrect attempts by the participant, the interviewer prompted them to recall the previous Parsons problem activities and to compare the number of lines used in the previous solutions.

> **interviewer** - ... try to recall how many steps you had in the previous page in terms of doing the actual swap...Do you remember how many further steps you had? At the moment, you have only one line.
> **python-swap-6** - Okay.

Participant **python-swap-8** indicated initial confusion with the `temp` variable in the first Parsons problem activity with pseudocode comment blocks. During the think aloud, the participant vocalized their confusion with the `temp` variable since it was not declared in the problem description. As a result, the participant attempted a three-block solution (shown below), excluding the two blocks with the `temp` variable.

```
# initialize the variables
# set x to the value of y
# set y to the value of x
```

The participant stated:

> **python-swap-8** - . . . Because there is nowhere that mentions *temp* . . . I am trying to figure out the odd one out almost . . . I just need three pieces of the five pieces of code.

The interviewer and participant discussed the problem, which helped the participant realize the `temp` variable is in two lines of the code, demonstrating to the participant that their initial mental model of the solution could not lead to a correct solution. The discussion also highlighted the presence of the `temp` variable in the first line, "Initialize the variables". However, the use of this variable continued to confuse the participant. After the discussion, the participant constructed an incorrect solution using four lines of code: the last line `y = x`, instead of the correct statement `y = temp`. However, upon completion, the participant identified the issue and adjusted the last line of code to produce a correct solution in their second attempt.

Positive influence with understanding the swap algorithm: Participants involved in this think-aloud study reacted positively to learning the swap algorithm by solving Parsons problems before writing code. Participants indicated that the Parsons problems supported their understanding of the algorithm and felt they would not have been able to solve it on their first attempt successfully without it. The following three quotes came from think alouds where the participants perceived the Parsons problems supported their understanding of the swap algorithm.

> **python-swap-1** - All right, I'd definitely say the Parsons, if you put me directly into the code, I probably wouldn't have been able to figure it out. I think the Parsons kind of gave me the general format and how it's supposed to be completed.

> **python-swap-7** - I mean that I think I was aware of that coming in, I mean the process of swapping two values... Had you just given me this last bit at the start [the code writing problems]. I would have got it wrong at least once then had to work out what I'd done wrong. But by doing the blocks [Parsons problems] before hand has made it a lot clearer in my head.

> **python-swap-8** - I would say the second one on the previous page [i.e. Parsons Problem 2 - pseudocode comment blocks plus code] was a good introduction to them, this one here [i.e. Write-code Problems].

Some participants expanded on how the Parsons problems supported their understanding. Below are three excerpts from think alouds demonstrating how Parsons problems helped them.

> **python-swap-5** – It's just, it [swapping] feels condensed, almost like, like the whole Parsons problems. It's now just one step with the entire code, if that makes sense. Yeah, the Parsons problem exists from line 11 to 13 [in the code writing problem].

> **python-swap-8** - But this one [i.e. Write-code Problem 1] was extremely valuable without having the code in front of me. If I'd had the previous piece of code [i.e. Parsons Problems] in front of me, I wouldn't have thought it through to the same extent as what I did there [i.e. Write-code Problems]. I have more understanding by almost having to just figure it out myself, but . . . you've had some previous insight into it from what I did in the previous page [i.e. Parsons Problems]. But it's not that I could recall that, but helped me thinking: Okay, there's a *temp* variable, and this kind of thing. So yeah.

> **python-swap-9** - So, I'm comparing writing the code to the last [i.e. Write-code Problems] to the previous exercises [i.e. Parsons Problems]. I was clicking and dragging. And I think that if you are just approaching that problem [i.e. swapping] for the first time, it's easier to click and drag [compared] to written code because it's like multiple choice in it. But if you are writing code for the first time it's probably more to work out to know what's involved.

Their statements suggest that the Parsons problems were sufficiently effective at helping the participants understand the swap algorithm. In particular, participant **python-swap-5** internalized the process taught via the Parsons problems as a single "chunk" and was able to transfer their mental model of the solution to the code-writing activities.

Difficulty organizing the pseudocode comment blocks: We observed that participants needed additional support with organizing the pseudocode comment blocks. For example, **python-swap-2** initially struggled with the problem with only pseudocode comment blocks, resulting in the participants getting the problem wrong several times. When encountering a Parsons problem activity with code, one participant stated *"this is the way I would have done it. . .*

showing code rather than the description". When asked to reflect on their work completing the activities, a participant stated:

> **python-swap-2** - I think having code written out [code blocks] rather than just the descriptions [pseudocode comment blocks] is a lot easier for me at least, like, um, when it's just the descriptions, it's kind of hard to follow in my head. But then if I have code, whether it's with the description or with out the description, seeing code is a lot easier

Participant **python-swap-2** is an engineering major who claimed they previously taken an advanced CS course, yet still enrolled in the CS1 course involved in this study. The sentiment about pseudocode comment blocks was also echoed by other participants, which we highlight in the following two think-aloud excerpts:

> **python-swap-5** - Yeah, it was very hard to like put words or put the word turn the words into like a code that I was like, used to like working with. So kind of coding by words was very hard for me.

> **python-swap-4** - Yeah, it was just kind of weird because I'm like if it just showed me the code, I would have been able to do it right away. But seeing the higher level descriptors I've never done Parsons problems like that before so I think I just need to adjust to it. Like I'm used to more like this where it's like, it might have a comment but it's showing you what the code looks like.

The participants' opinions on pseudocode comment blocks may be because it is the study's first practice Parsons problem. We observed participants preferring subsequent activities, such as participant **python-swap-10** expressing the second problem with pseudocode comments plus code was more approachable in comparison to the first, stating:

> **python-swap-10** - . . . I think I find the version with the comments actually a little bit harder. I found it's a bit more abstract. . . Also affected by the fact that is the first time I see that problem in this exercise [i.e swapping].

Other participants also preferred the second Parsons problem containing comments and code. Below are three excerpts from *python-swap* participants expressing their preferences.

> **python-swap-8** - . . . This one here [i.e. Parsons Problem 2 - pseduocode comments plus code blocks]. The first one [i.e. Parsons Problem 1 - pseudocode comment blocks] for me is much harder to figure out. . . So, I think the detail within this one [i.e. Parsons Problem 2 - pseduocode comments plus code blocks]. I find it easier to comprehend when there's values assigned to variables rather than just the name of the variable.

> **python-swap-9** - . . . Code and comments is easiest. . . the second version [i.e. Parsons Problem 2 - pseduocode comments plus code blocks], . . . as you can actually see the values. It's easy to relate it.

> **python-swap-11** - . . . For example, I do not have comments on paper [this participant used pen-and-paper to produce a solution to Parsons Problem 1 - pseudocode comment blocks]. If I had the comments on paper, maybe it would have been a little easier.

The results from the *python-swap* study provided evidence that solving Parsons problems with distractors can help students overcome common misconceptions and learn common algorithms, like swapping variables. However, we need to investigate further how the presentation order of the three types of Parsons problems impacts their learning and understanding of these concepts.

6.1.2 Think Aloud Observations: class-ta . This study had students solve Parsons problems with and without distractors and then write and fix code with similar errors to the distractors. Two think-aloud observations were performed with students from Berea College using the *class-ta* study. Due to the low number of participants we follow the approach of student biographies and narrative interview overview used by Haynes-Magyar and Ericson [43] when reporting the results of these two think-aloud observational studies.

Participant class-ta-1 Biography: The participant **class-ta-1** is a 19-year-old rising sophomore male student majoring in computer science. Although English is not the language that he speaks at home, he rated his ability to read and understand spoken English as good. He had completed a CS1 course in Python at the time of the think aloud, where his CS1 used Runestone with Parson problems activities. As a result, the participant was highly familiar with Runestone's UI and with Parsons problems.

class-ta-1 Results: This student was highly engaged and made numerous statements about components he found helpful in Runestone during his CS1 course. His previous experience with Runestone provided him with a high level of understanding of the platform. He could explain the UI in-depth as if he were explaining to a peer. For example, he said,

> **class-ta-1** - Here we have a video, like from YouTube, so if we want we can just watch it and get like a bit a better understanding of the concept so... Right now we don't have to watch it like so we can just go to the next section, but if we want to we we can just click it... So basically those videos are I believe embedded from YouTube to run.

The interviewer encouraged him to engage with the activity as if he were working independently. However, vocalizing his actions during the think aloud may have interfered with his focus on the intended scaffolding of the study's Introduction to Problem Types and Creating Classes sections.

On the first Parsons problem, he initially failed to notice that he had chosen both paired distractors. After seven attempts at solving the Parsons problem, he correctly solved it, saying:

> **class-ta-1** - So, basically what I did was first like I run the the code blocks, and I was able to see my mistakes. Then I read the section that tell me which things I make wrong, and I was able to notify my mistake. And, then basically I just fixed it.

With the second Parsons problems activity, the participant correctly solved it after two attempts. With the third and fourth Parsons problems, he successfully solved them in one attempt. Afterwards, he stated:

class-ta-1 - On the first and second problems, I was trying to do everything at the same time. I was just trying to plug like you know everything at once. But when when it's come to the third and the last,... it's really helpful to do each things like step by step the state of trying to do everything at once.

Despite his increased success in solving the Parsons problems, participant **class-ta-1** was unable to solve the code-writing activities. During these code-writing activities, he expressed confusion over the `self` keyword and conflated the use of `self` with whether or not a type conversion would be needed.

Participant class-ta-2 Biography: Participant **class-ta-2** is a 21 male rising junior majoring in Computer Science. English is one of two languages he speaks at home. Before the think aloud, he reported high confidence in his Python programming ability. However, after the think aloud, he acknowledged that he recently took a CS2 course in C++, so he had not worked with Python for at least six months. The participant reported that the time away from Python made it challenging to remember the programming syntax. His CS1 and CS2 courses utilized Runestone textbooks, so he was highly familiar with the UI and with Parsons problems. However, he had not previously seen a toggle problem that included code writing, which contained a feature that pops up a Parsons problem as scaffolding.

class-ta-2 Results: Overall, the participant progressed quickly through the lesson pages presented in the study and with relative ease. His first point of error came when selecting between a correct block (`Class Song:`) and its visually paired distractor (`class Song:`). After choosing the distractor and before testing his solution, he reflected on his decision, stating:

class-ta-2 - Newer programmers like me at least don't pay a lot of attention on which letter is a capital letter and so on so forth... Generally, since the IDEs, like you know, PyCharm and other programs like that... when you write something it just fixes it for you. ... I'm pretty confident that it is a capital 'C'.

After he finished arranging his choice of fragments to form a program, he tested his solution and exclaimed:

class-ta-2 - It says that this is wrong... So if I change it with this, oh, oh! So, the 'c' is Okay. I will never forget that ever again! All right, so the 'C' is upper case... I will never forget that ever again!

The participant's exclamation provides an example of a student being startled into a period of reflection when selecting between a correct block and a distractor block containing a common error made by novices. Additionally, the participant had a highly positive reaction to the result of the interaction, generating self-reflection that suggests an effective distractor may have encouraged his recollection of the correct syntax.

The next two Parsons problems went smoothly for him. Unfortunately, after spending several minutes on the FortuneTeller Practice Problem, he could not solve it before moving on to the posttest. He first realized he was supposed to make a class and began creating an initializer. Without finishing the line of code containing the initializer, he asked, *"Uh, is it OK for me to just tell my friends to be a little bit quieter for a second?"* The interviewer responded, *"They're not bothering me. But if it if it's more helpful to you to do that, that's fine."* He replies, *"Yeah, it's just they're being a little loud. I don't know if you can hear from the microphone, but I can hear them."* He got up, opened his door and spoke into the hallway. This distraction potentially contributed to the challenges he next faced. The interviewer encouraged him to take his time to arrange his thoughts. He next re-read the problem statement while highlighting it with his cursor. Then, before he could type, his calculator popped up for some reason, and he quickly closed it. Before executing his coded solution, he said, *"this might be wrong"*. The response from Code Coach was blank. However, the test response to his solution generated the error message *"Error: Maximum Call Stack Exceeded"*. The following code segment came from the participant's solution.

```
class FortuneTeller:
  def __(self, f):
    self.f = f
  def tell_fortune(self, f):
    self.f = FortuneTeller(["You will get an A", ...])
    ...
```

The participant proceeded to make several changes in this work using CodeLens on his iterative solutions. However, despite the interviewer encouraging the participant to reflect on the problem's name, `toggle`, he moved on to the posttest without realizing he could pop-up a Parsons problem as scaffolding. This interaction highlighted a potential problem with the UI presenting the keyword `toggle` for additional support. This keyword might not be sufficiently clear to the students.

On the first problem on the Post Test, he struggled to build a correct `__str__` method because he had added two parameters, but was not using them. He requested assistance from the researcher and with some help, he then was able to articulate differences between parameters and instance variables. After this, he solved all of the remaining post test problems without seeking further assistance.

He then indicated that he wanted to return to the FortuneTeller problem to attempt it again. This time, the interviewer helped him use the toggle problem type, which supported him to solve the problem independently.

6.1.3 *Think Aloud Observation: p3dndta* .

The *p3dndta* study is a version of *p3pt* without distractors that is specifically designed for think-aloud observational studies. See Section 4 for the study design.

p3dndta-1 Participant Biography: We conducted the study with participant **p3dndta-1**, a 19-year-old female rising sophomore majoring in computer science. English is her second language, and she rated her ability to read and understand spoken English as very good. Her CS1 course utilized a Runestone textbook, so she had experience with the platform's UI and Parsons problems. However, she had no prior background in using adaptive Parsons problems, combining pseudocode comment blocks and removing distractors. She disclosed during the study that she has high-functioning anxiety.

p3dndta-1 Results: Overall, the participant progressed through most of the study quickly, with the first challenge occurring in the

Student	Creating classes like class Person: and objects like p = Person("Barb Ericson")	Methods like __init__ and __str__	The use of self in class	Defining instance variables like self.color = color
class-ta-1	4	4	5	5
class-ta-2	5	5	5	5

Table 13: Responses to the pre-survey questions specific to the class-ta study. Responses were collected on a 5 point Likert scale (1) strongly disagree to (5) strongly agree

Student	Loops/Iteration like for n in nums: and for i in range(4):	Conditionals/Selection Statements like if x < 3:	Functions like def get_odd(nums):	Lists like a = [1, 2, 3]
p3dndta-1	3	4	4	3
jspt-1	2	3	2	2
jspt-2	3	3	3	4
jspt-3	4	5	4	4
jspt-4	3	3	2	3
jspt-5	4	5	4	5
jspt-6	2	3	2	2
jspt-7	3	3	2	3
jspt-8	2	2	2	2
jspt-9	4	5	4	4
jspt-10	2	2	3	2
jspt-11	4	5	4	4
jspt-12	4	4	4	4

Table 14: Responses to the pre-survey questions specific to the *p3dnd* and *jspt* studies. Responses were collected on a 5-point Likert scale (1) strongly disagree to (5) strongly agree

Introduction Section and continuing to the Problem Types Section. The Problem Types section contained a code-writing activity asking the participant to program the function `triple(num)` that takes a number variable, `num`, and returns that number times three. She succeeded after five attempts. Upon completion, she reflected:

> **p3dndta-1** - The Code Coach is very helpful because it points out some possible solutions for students.

The participant solved all six Parson problems correctly within two and thirteen attempts. She seemed delighted by the *"Help Me"* button on the adaptive Parsons problems, stating:

> **p3dndta-1** - I learned a lot from the combined blocks. It was truly, truly helpful... I really appreciated the help me (button)!

The participant attempted to solve the first code-writing activity, asking her to write a function called `is_descending(num)`. The function returns `True` if the numbers in the `nums` list are sorted in descending order; otherwise, the function returns `False`.

For the code-writing activity, the participant implemented the line `for num[i] in len(num)` to iterate the loop on line 3 of her solution. Unfortunately, her solution generated an error, *"SyntaxError: bad input on line 4"*. Because the error referenced the proceeding line (line 4), the message did not provide sufficient help to support her to solve the activity successfully. She did not attempt additional code-writing activities.

6.1.4 Think Aloud Observations: jspt . The *jspt* study used in the think aloud is a variant of the Python *p3pt* study written in JavaScript instead. The description of the JavaScript *jspt* study design is in Section 4.6.2.

About 25% of the study participants (4 out of 12) completed all of the study sections. The results from the participants' data showed they had a favorable view of the Parsons problems. For example, they valued that the initial practice Parsons problems helped them plan and design a solution. The Parsons problems allowed the participants to consider the function's purpose and unit test cases. Likewise, the drag-and-drop feature of Parsons problems allowed most of the participants to understand better the code-writing activity they attempted to solve despite their lack of awareness that the Parsons problems and code-writing activities were isomorphic.

Some participants did not complete all the problems in the study. The most frequent reasons were:

- Lack of time. For example, participant **jspt-3** stated *"the activity was too long and I didn't manage to work in the last exercise"*.
- Fragile knowledge of lists and arrays. For example, participant **jspt-8** stated, *"I didn't understand how to choose between the two blocks of code: both seemed exactly the same to me"*.
- Anxiety. For example, participant **jspt-6** stated, *"I felt quite overwhelmed by the end, so I decided to drop out and let it go... This is so frustrating and it only shows me that I need to work"*.

The sentiments expressed by the three participants demonstrate factors that contributed to them not completing the majority of the code-writing activities, even though they were isomorphic to the Parsons problems. Furthermore, one participant pointed out that

Student	Parsons problem 1	Write-code 1	Parsons problem 2	Write-code 2	Parsons problem 3	Write-code 3	Parsons problem 4	Write-code 4
jspt-1	Attempted	Attempted	17	Attempted	24	Did not attempt	Did not attempt	Did not attempt
jspt-2	3	Solved	Attempted	Solved	4	Solved	Did not attempt	Did not attempt
jspt-3	1	Solved	5	Attempted	4	Solved	6	Did not attempt
jspt-4	5	Attempted	Attempted	Did not attempt	Attempted	Solved with Assistance	Did not attempt	Did not attempt
jspt-5	3	Solved	2	Solved	2	Solved	2	Solved
jspt-6	Attempted	Did not attempt	7	Did not attempt	11	Did not attempt	Did not attempt	Did not attempt
jspt-7	2	Solved	8	Did not attempt	10	Solved with Assistance	Attempted	Did not attempt
jspt-8	Attempted	Did not attempt	8	Did not attempt	Attempted	Did not attempt	Attempted	Did not attempt
jspt-9	4	Solved	1	Solved	3	Solved	Attempted	Did not attempt
jspt-10	1	Solved	2	Did not attempt	6	Solved with Assistance	5	Did not attempt
jspt-11	2	Solved	2	Solved	5	Solved	Did not attempt	Attempted
jspt-12	4	Solved	3	Solved	7	Solved	Attempted	Attempted

Table 15: Student coding attempts and successes in *jspt*

they felt less anxious while solving Parsons problems and were disinterested in completing the code-writing activities.

Finally, it is worth pointing out that none of the participants—even those who completed the study, reported high self-efficacy, and a medium-to-high pre-knowledge of the concepts covered—realized that problems in both parts of the study were isomorphic. This implies that study participants have not mastered yet, or at least still have some work to do, in developing a mastery of abstraction.

6.2 Quantitative Experimental Studies

In this section we describe the Parsons experimental studies that we ran. We ran several different studies at several institutions.

6.2.1 python-swap. Quantitative studies were conducted at Falmouth and Ashesi Universities. Figure 19 shows each cohorts' respective familiarity with programming a variable swap and their self-efficacy with computer programming. Both cohorts reported they were familiar with the notion of swapping variables. Falmouth University students reported being more familiar perhaps due to being in a CS1.5 type of course vs the Ashesi CS0.5 type. The cohorts had broadly similar distributions of programming self-efficacy, with Ashesi University students skewing towards more confident they could successfully complete the task, though this was not a statistically significant difference ($t = -.56, p = 0.57$).

Figure 20 illustrates the correlations between these two attitudinal variables with the number of attempts required to reach success. As might be anticipated, the strongest correlation was between their reported self-efficacy and their score on the final code writing problem ($r = 0.41, p = .024$). There was a notable negative relationship between their self-reported familiarity with swap and the number of Parsons problem-solving attempts, which was consistent across both contexts. The more familiar, the fewer attempts needed ($r = -0.37, p < 0.5$). This suggests that those students who have encountered swap before were able to apply their experience readily to solve the Parsons problem with fewer attempts than peers who were less familiar. However, this familiarity with the swap exercise was not correlated with the number of code-writing attempts, with a magnitude close to zero ($r = 0.1$) in both contexts ($p = 0.38$). Crucially, there was no correlation between those students succeeding on the final code-writing task and their familiarity with the value-swapping problem ($r = .054, p = 0.77$). This implies that the Parsons problem practice was able to close the experience gap between those who were familiar with the algorithm and those who were not.

Analysis of time-series data relating to each task (starting with `intro-simple-parsons-no-indent`, the introductory tasks, proceeding to `ps_swap_comments_pp`, the practice with parsons problems, and concluding with `ps-swap2-ac`, the final code writing exercise) illustrates favorable retention. Figure 21 shows the completion rates of the two cohorts. More than 80% of participants successfully completed the final code-writing exercises.

A small proportion of students were disengaged by the introductory material. This may indicate similar challenges as those encountered with the interface and the user experience in the qualitative studies. There was a noticeable drop in the proportion of students completing the pseudocode comment blocks Parsons problem that presented the steps in the algorithm without any code. Figures 22 and 23 show the mean attempts and mean times needed to complete these tasks. Examining these offers a potential explanation as there is a noticeable increase in the time required to complete the pseudocode comment blocks Parsons problem, requiring an average more than six attempts in both cohorts. This is consistent with the qualitative observations in the think-aloud studies. Students need to read through and parse the pseudocode comment blocks themselves. They also needed to think through and experiment with the sequence of operations. There was also a modest drop in successful completion rates for the final two puzzles. Though, those completing the first post-test code writing task tended to also complete the second. This too is illustrated in Figures 22 and 23. One aspect of this seems to be the cross-referencing prior material, suggesting the scaffolding was useful. Though, in the think aloud observations, some participants took time to recognise that the problems they are tasked with were the same as the ones they had just solved previously. With respect to the difference between the first post-test write-code problem and the second, students tended to complete the second code writing problem designed to verify near transfer in less time than the firsts, though some of the think-aloud observations suggest continuing obstacles with syntax.

6.2.2 class-exp. Quantitative studies were conducted at DePaul University (N=42), Duke University (N=130), Berea College (N=14), Falmouth University (N=20), and the University of Michigan (N=155), with a total of 361 participants contributing to this study. The students at University of Michigan were shown 4 questions (hereafter referred to as *class-exp-4q*) as part of this study whereas all other institutions had 5 questions (referred as *class-exp-5q*).

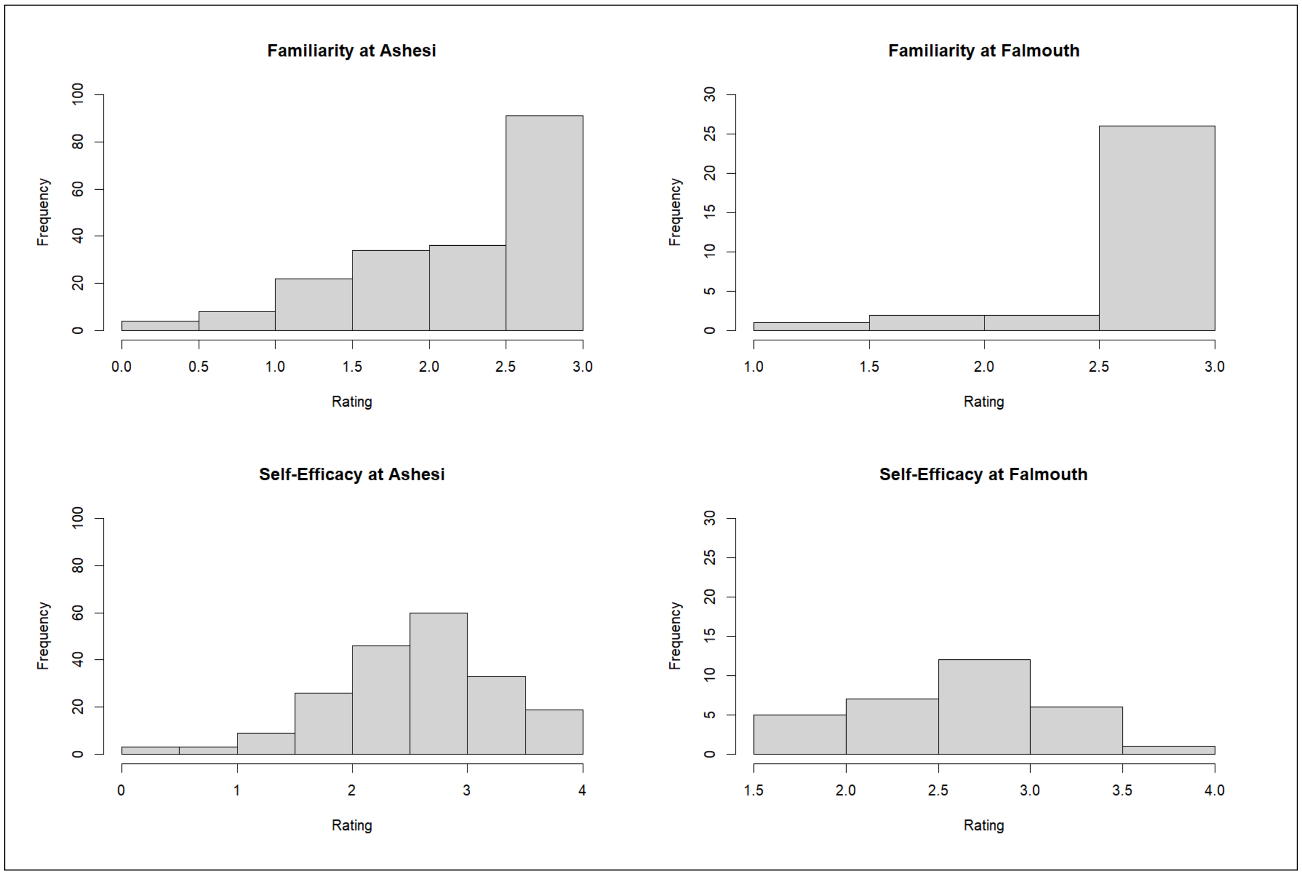


Figure 19: Students' stated familiarity with variable swap and programming self-efficacy

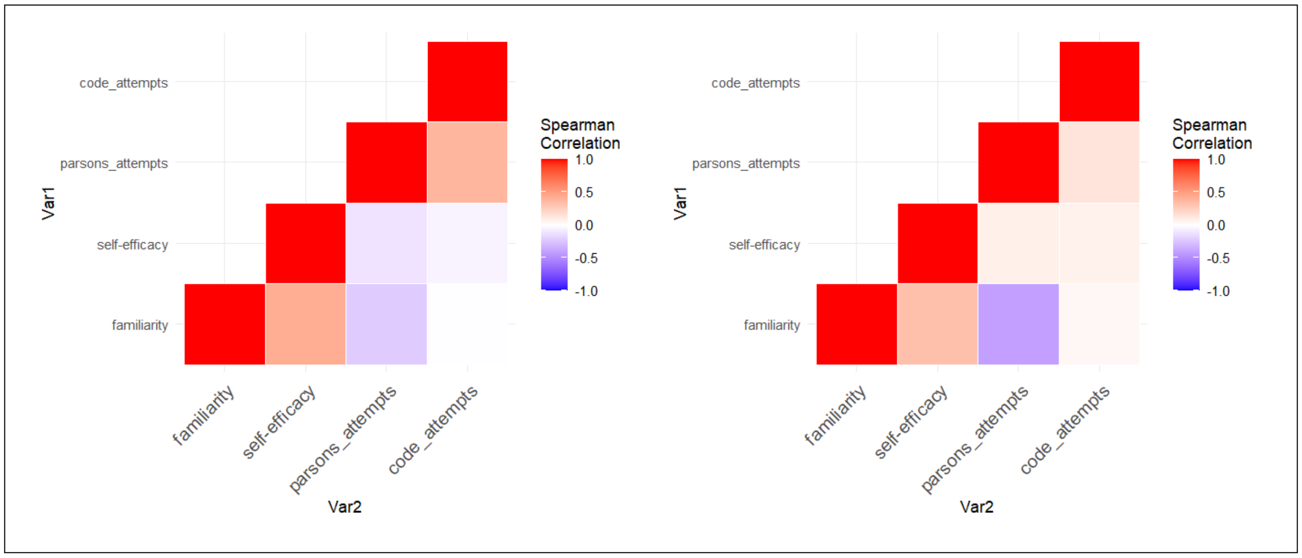


Figure 20: Correlations between stated familiarity, self-efficacy, and attempts (Left: Ashesi, Right: Falmouth)

We used Mann-Whitney U Tests to check if the conditions (with and without distractors) were comparable by students' distributions of self-efficacy and pre-knowledge: for self-efficacy, $U = 16638.5$,

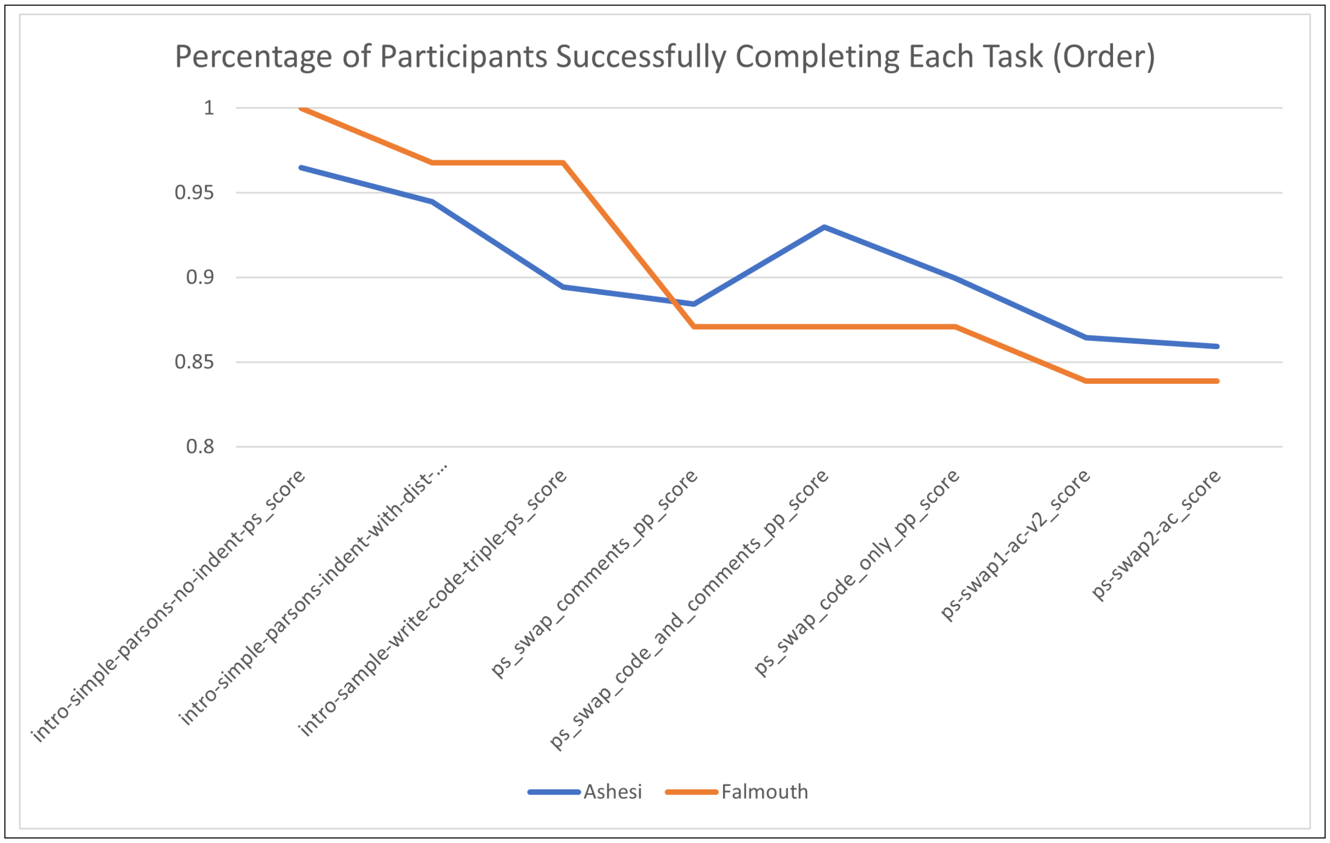


Figure 21: Drop-off rate of students not completing each problem successfully

p-value = 0.70; for pre-existing knowledge U = 16638.5, p-value = 0.92. Therefore, the conditions are comparable.

We first compare the post test scores of the students with and without distractors in the *class-exp-5q* for identifying improvements in the learning performances. The difference in the averages (with distractors: 335/500, 67%; without distractors: 311/500, 62%) were not statistically significant (p-value = 0.56 shown by a Mann-Whitney U-test). For the *class-exp-4q* group too, the average post-test score differences (with distractors: 304/400, 76%; without distractors: 276/400, 69%) were statistically not significant (p-value = 0.45).

We had also categorized the students into two groups (high and low) based on their self-efficacy and pre-existing knowledge levels to further analyze performance differences. There were 153 students in the high self-efficacy group(μ: 19.12, σ: 2.56) and 208 students in the low self-efficacy group (μ: 11.52, σ: 3.90) while there were 155 students with high pre-existing knowledge level(μ: 10.81, σ: 2.95) and 206 students with low pre-existing knowledge (μ: 2.67, σ: 1.91). The self-efficacy scale used in this study was found to be internally reliable with a Cronbach's α of 0.877.

In this study, we found a decline in participation in the post-test, as shown in Tables 16 and 17. It is observed that the students who were provided with the distractors were generally more likely to attempt problems on the post-test, although none were statistically significant, as shown in Table 18. Further dividing the students based on their self-efficacy and pre-existing knowledge of the concepts under examination, we found students with lower self-efficacy or lesser pre-existing knowledge demonstrated a significantly higher likelihood of attempting all problems on the post-test, particularly the write-code problems, when exposed to distractors during practice, as shown in Table 18.

We were also interested in understanding if students would finish the post-test problems at different speeds, if shown distractors while practicing. There was a broad trend in those shown distractors completing the problems faster. However, we did not find a significant difference in time taken to complete the post-test, except for with the *Movie* post-test problem, in which students who were shown the distractors completed the problem a minute faster (on an average) than those who were not shown them (shown in Table 19).

Another characteristic that we were interested in comparing was the differences in number of syntax errors made across conditions. Those that were shown distractors (*class-exp-5q*: 18 average errors, *class-exp-4q*: 13 average errors) made significantly less errors in the post-test, compared to those that were not shown distractors (*class-exp-5q*: 23 average errors, *class-exp-4q*: 19 average errors), with a Mann-Whitney U-test (*class-exp-5q*: p-value = 0.03, *class-exp-4q*: p-value = 0.003).

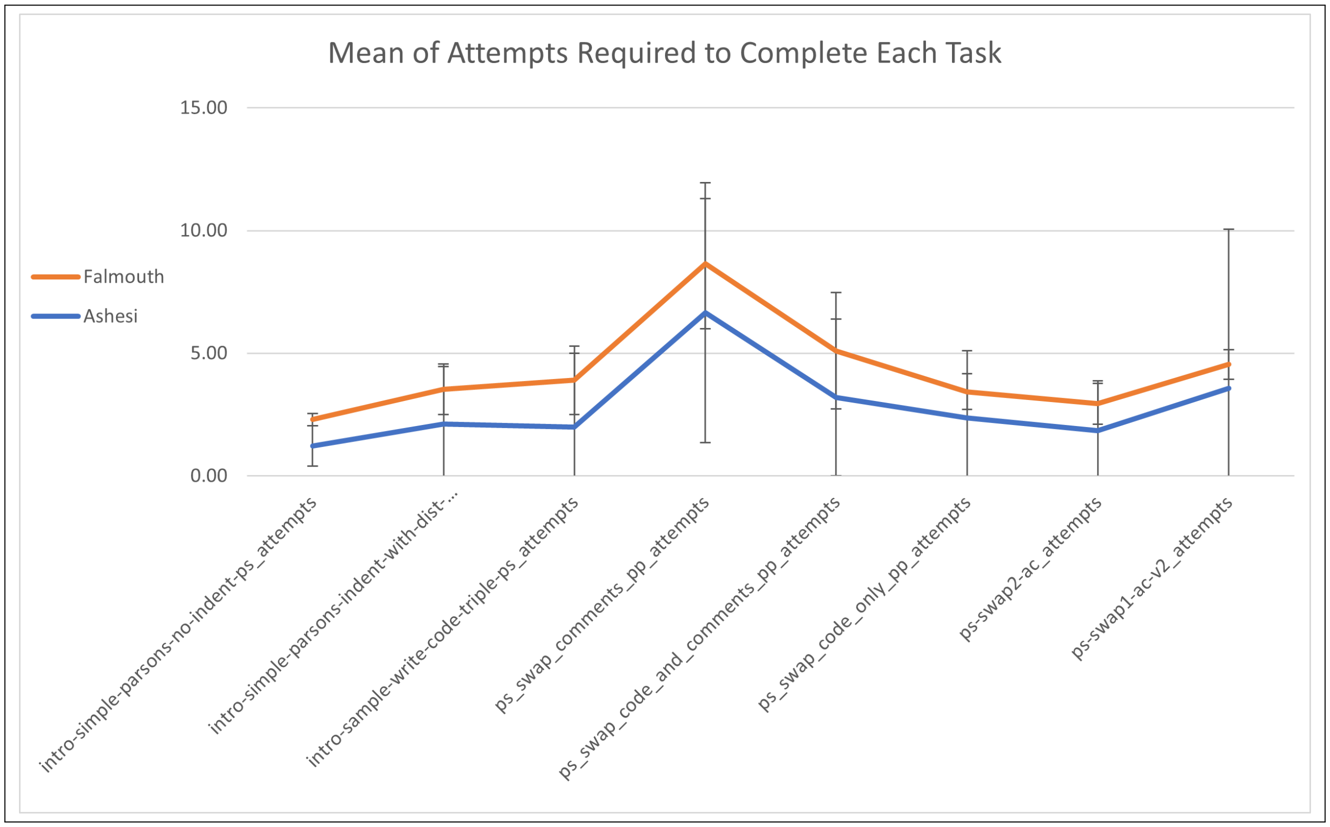


Figure 22: Mean number of attempts utilized by students successfully completing each task

Problem	Problem Type	# of Participants Attempted
Q1	Fix Code	173
Q2	Fix Code	142
Q3	Write Code	149
Q4	Write Code	139
Q5	Fix Code	118

Table 16: Number of participants attempting each question of the posttest for all contexts combined, except University of Michigan.

Problem	Problem Type	# of Participants Attempted
Q1	Fix Code	131
Q3	Write Code	116
Q4	Write Code	112
Q5	Fix Code	108

Table 17: Number of participants attempting each question of the posttest for University of Michigan.

6.2.3 p3pt. Quantitative studies of *p3pt* were conducted at Ashesi University (N=168), DePaul University (N=39), Falmouth University (N=10), Victoria University of Wellington (n=7), and IIT Madras (N=152), with a total of 369 participants contributing to this study.

To understand how Parsons problems or write-code practice questions may impact students with varying CS self-efficacy levels and pre-existing knowledge, we separated the students into groups: those with high (N: 175, μ: 18.77, σ: 2.42) and low (N: 226, μ: 11.74, σ: 3.2) self-efficacy, and those with high (N: 133, μ: 13.99, σ: 1.60) and low (N: 268, μ: 9.01, σ: 1.92) pre-existing knowledge. The self-efficacy scale used in this study was found to be internally reliable with a Cronbach's α of 0.870. We used Mann-Whitney U Tests to check if the conditions (with and without distractors) were comparable by students distributions of self-efficacy and pre-knowledge: for self-efficacy, $U = 17862.0$, p-value = 0.72; for pre-existing knowledge $U = 6384.5$, p-value = 0.80. Therefore, the conditions are comparable.

We found no significant difference in comparing students' performance in the post-test by their practice condition using a Mann-Whitney U-test ($p = 0.60$). Students in the Parsons problems practice condition scored on average 107/400, and students in the write-code condition scored on average 111/400. We also found no significant difference in condition of whether students attempted all or any of the post-test problems, as shown in Table 21. However, students with Parsons problem practice questions, on average, took significantly more attempts until getting the correct answer compared

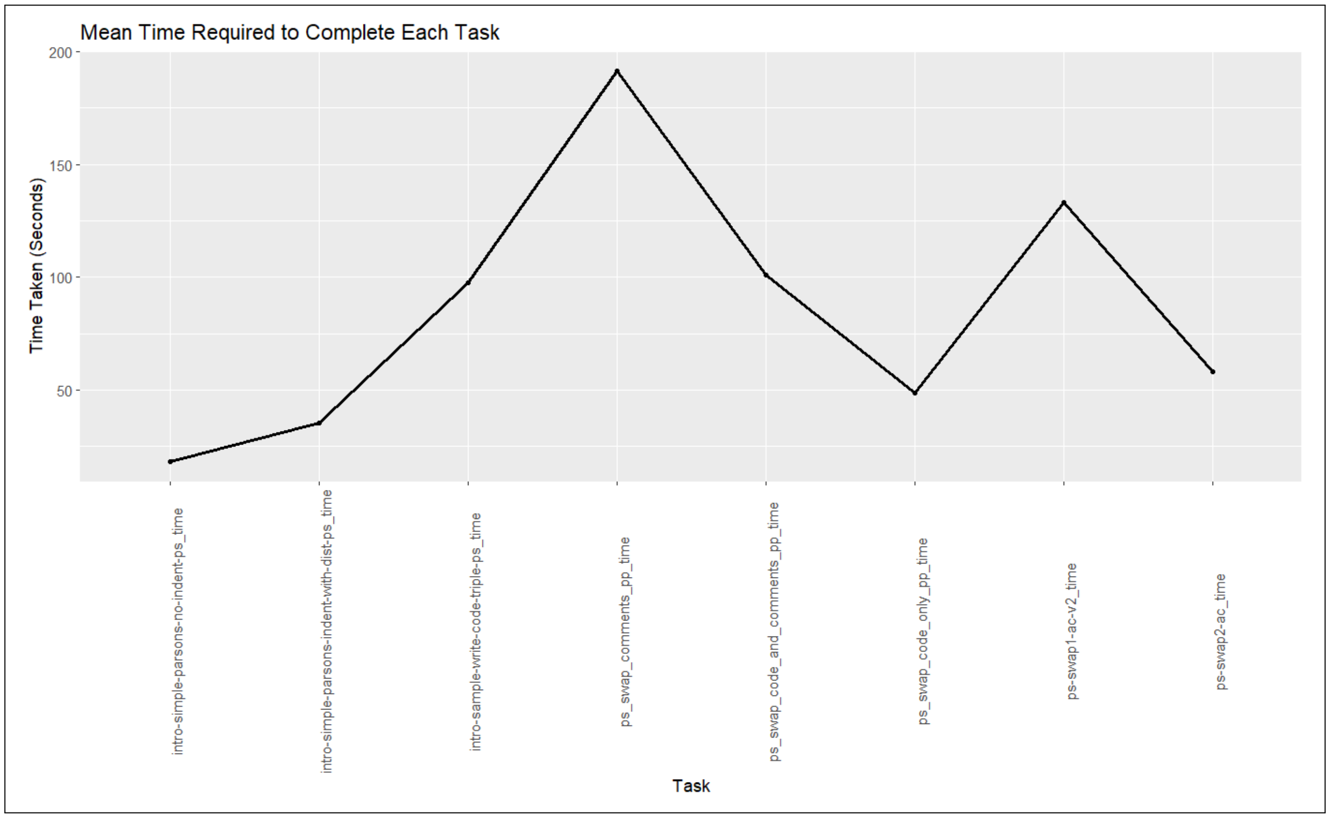


Figure 23: Mean time to a correct solution at Ashesi University (in seconds)

the students with write-code practice problems, as shown in Table 22. This was also significant in students with low self-efficacy.

To answer part of our research question for this study, we compared completion times for the practice questions between those with Parsons problems, and those with write-code problems. We found no significant difference between the average total time to completion of students with Parsons problems (1783 seconds) to the average total time to completion of students with write-code problems (1600 seconds) with an independent t-test value of p=0.58.

We also investigated if students finished the post-test problems at different speeds, if shown Parsons problems or write-code problems while practicing a concept. There was a broad trend in those shown write-code questions completing the problems faster. This was found to not be significant in the post-test altogether, as well as separately for each post-test question, excluding the first question, as shown in Table 20.

6.2.4 jspt. For the group analyzed in this study, the self-efficacy scale was found to be internally reliable with a Cronbach's α of 0.91. Consistent with the group splits initially reported by Wiggins et al. [117], we divided the study sample ($N = 31, Med = 3.5$) into two groups: those with a higher self-efficacy score than the median ($N_{high} = 16$) and those with a lower score ($N_{low} = 15$). We analyzed two metrics: (1) the number of Parsons problems that were solved completely and (2) the mean time to completion, for the students who successfully completed all four problems presented in the study.

On the one hand, Figure 25 depicts the number of completed problems, according to the two subgroups analyzed in the study. Given the reduced sample size and the fact that data did not satisfy the normality assumption (verified through applying Shapiro-Wilk tests), we opted to run non-parametric tests for the analysis; in this case, Mann-Whitney for statistical significance and Cliff's delta for effect size. We did not observe a statistically significant difference in the number of correctly completed problems ($U = 161.5, p = .086$) between students in the high self-efficacy group ($Med = 4$) and those in the low self-efficacy group ($Med = 2$). We observed a medium effect size of $\delta = .346$.

Considering only the participants who correctly completed all four Parsons problems, we did not observe a statistically significant difference in the mean time to completion ($t(9.3055) = 0.9938, p = .3455$) between students who declared high self-efficacy ($N_{high} = 9(56.25\%), M = 437.76, SD = 158.89$) and those who declared low self-efficacy ($N_{low} = 5(33.3\%), M = 519.66, SD = 141.19$). There was a medium effect size with $d = .534$.

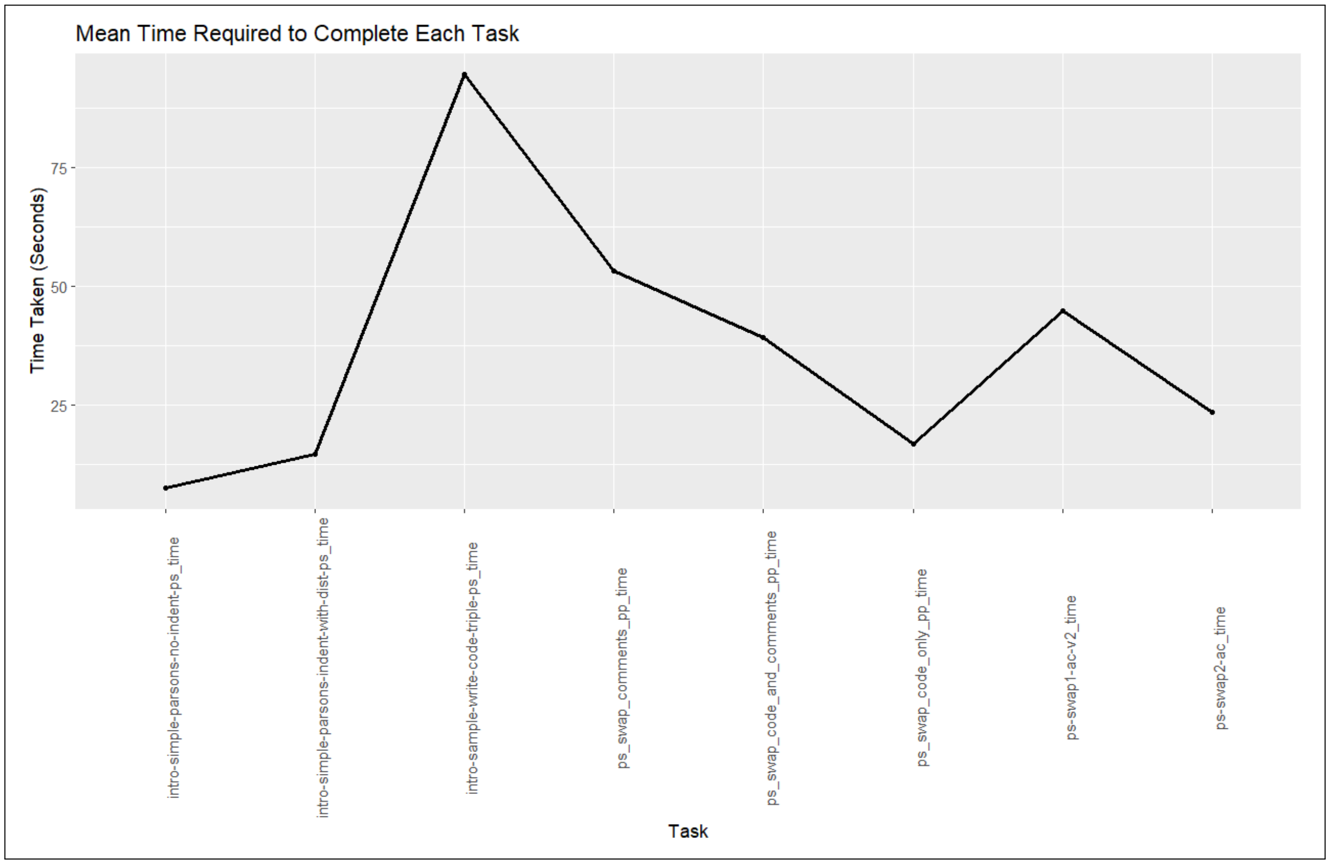


Figure 24: Mean time to correct solution at Falmouth University (in seconds)

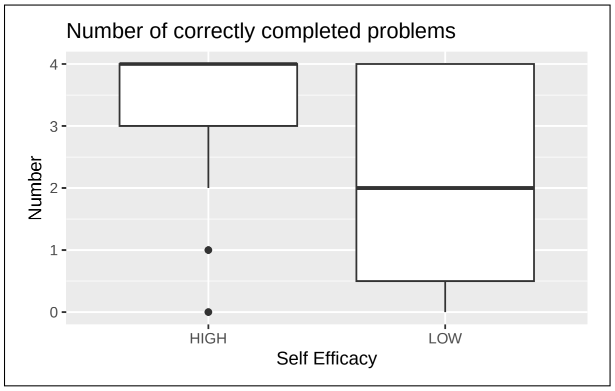


Figure 25: Number of correctly solved problems (*jspt*).

7 DISCUSSION

7.1 Parsons Problems Recent Directions

Since 2022, Parsons problems have continued to be investigated most heavily in the context of learning to program and most frequently at a single institution. In the past year, we have seen an increase in the use and study of cognitive load theory with respect to Parsons problems. Research has increased on different types of Parsons problems, including newer variants such as adaptive Parsons, faded Parsons, and micro Parsons. In micro Parsons problems the learner puts fragments into a single statement such as the symbols in a regular expression or the keywords in a SQL query [118].

As in the past, the majority of articles on Parsons problems were published by researchers working at institutions in the USA. The only other countries for the 2022 and 2023 publications we found had just one or two papers. The ACM Digital Library in the past year did not see any expansion into new countries or regions beyond those from which there had already been Parsons problem publications.

7.2 Addressing Considerations for MIMN Studies

This discussion section is viewed through the lens of critical self-reflection, as advocated by Brookfield [14]. Brookfield proposes four lenses for critical self-reflection: autobiographical, students' eyes, other practitioners' experiences, and theoretical literature. Predominately, the results generated in this study are viewed through the first autobiographical lens by the working group members. The conducted think-aloud studies are viewed through the second lens, that of students' eyes which enables their voices to be heard and

Group/Problems	wd	nd	χ^2	*p*-value
All Students				
All Problems Attempted	0.63	0.55	2.22	0.14
All Fix Code Attempted	0.65	0.57	2.00	0.16
All Write Code Attempted	0.73	0.65	1.98	0.16
Students: low self-efficacy				
All Problems Attempted	0.55	0.37	3.83	**0.05**
All Fix Code Attempted	0.55	0.41	2.25	0.13
All Write Code Attempted	0.68	0.46	6.15	**0.013**
Students: high self-efficacy				
All Problems Attempted	0.70	0.68	0.0	1.0
ll Fix Code Attempted	0.72	0.69	0.075	0.78
All Write Code Attempted	0.76	0.80	0.23	0.63
Students: low pre-knowledge				
All Problems Attempted	0.66	0.51	4.12	**0.04**
All Fix Code Attempted	0.66	0.54	2.71	0.10
All Write Code Attempted	0.77	0.62	4.49	**0.03**
Students: high pre-knowledge				
All Problems Attempted	0.60	0.61	0.0	1.0
All Fix Code Attempted	0.63	0.61	0.004	0.95
All Write Code Attempted	0.68	0.70	0.014	0.90

Table 18: Percentage of students attempting each group of questions, separated by whether the students were shown distractors (wd) or not (nd), alongside a χ^2 test of statistical significance and its respective *p*-value.

Group/Problem	wd	nd	*p*-value
Total Time Taken	1002s	1127s	0.44
Q1 Time Taken	154s	143s	0.78
Q2 Tank Time Taken	200s	218s	0.61
Q3 Time Taken	232s	305s	**0.016**
Q4 Time Taken	158s	183s	0.41
Q5 Time Taken	192s	201s	0.65

Table 19: Time taken on each post-test question of *class-exp-5q* alongside the results of an independent *t*-test comparing students time taken after completing practice questions with distractors (wd) and without distractors (nd).

Group/Problem	wc	pp	*p*-value
Total Time Taken	1270s	1882s	0.068
Q1 Time Taken	292s	566s	**0.017**
Q2 Time Taken	307s	445s	0.16
Q3 Time Taken	229s	350s	0.15
Q4 Time Taken	435s	431s	0.98

Table 20: Time taken on each post-test question of *p3pt* alongside the results of an independent *t*-test comparing students time taken after completing practice questions with write-code questions (wd) and with Parsons problems (pp).

Group/Problems	wc	pp	χ^2	*p*-value
All Students				
All Problems Attempted	0.25	0.22	0.38	0.53
Any Problem Attempted	0.47	0.51	0.55	0.45
All Fix Code Attempted	0.35	0.37	0.03	0.87
All Write Code Attempted	0.28	0.26	0.12	0.73
Students: low self-efficacy				
All Problems Attempted	0.21	0.21	0.0	1.0
Any Problem Attempted	0.41	0.56	3.48	0.06
All Fix Code Attempted	0.31	0.41	1.87	0.17
All Write Code Attempted	0.23	0.27	0.15	0.69
Students: high self-efficacy				
All Problems Attempted	0.29	0.23	0.71	0.40
Any Problem Attempted	0.52	0.48	0.28	0.59
All Fix Code Attempted	0.40	0.33	0.73	0.39
All Write Code Attempted	0.32	0.25	0.97	0.32
Students: low pre-knowledge				
All Problems Attempted	0.30	0.26	0.38	0.53
Any Problem Attempted	0.52	0.54	0.056	0.81
All Fix Code Attempted	0.42	0.39	0.12	0.73
All Write Code Attempted	0.33	0.30	0.12	0.73
Students: high pre-knowledge				
All Problems Attempted	0.16	0.14	0.02	0.89
Any Problem Attempted	0.39	0.47	0.45	0.49
All Fix Code Attempted	0.24	0.32	0.76	0.38
All Write Code Attempted	0.19	0.17	0.0047	0.95

Table 21: Percentage of students attempting each group of questions, separated by whether the students practiced using write-code problems (wc) or Parsons problems (pp), alongside a χ^2 test of statistical significance and its respective *p*-value.

Group/Problems	wc	pp	*p*-value
All Students			
Mean Attempts to Correct	6.32	9.82	0.21
Students with low self-efficacy			
Mean Attempts to Correct	3.55	8.61	**0.018**
Students with high self-efficacy			
Mean Attempts to Correct	8.69	10.66	0.68
Students with low pre-knowledge			
Mean Attempts to Correct	7.15	11.15	0.45
Students with high pre-knowledge			
Mean Attempts to Correct	4.77	6.91	0.31

Table 22: Number of attempts students took until passing all unit tests, separated by whether the students practiced using write-code problems (wc) or Parsons problems (pp), alongside a Mann-Whitney U-test of statistical significance and its respective *p*-value.

articulated regarding their participation in this study. While the third lens, other practitioners' experience, indicate the work and finding of other computing education researchers conducting research in this field. The fourth lens of theoretical literature enables the group to critically reflect upon what has previously occurred, discoveries and challenges to be addressed.

Applying the autobiographical lens of critical self-reflection, we reflect on our efforts in performing a MIMN study. Our self-reflection is guided by previous MIMN studies sharing their recommendations and considerations for conducting these studies (See Section 3.5). We also share our experiences to encourage more MIMN studies in CSE and raise awareness that organizations like ITiCSE can support future work. In this section, we divide our discussion into three parts: Setting MIMN studies up for success (Section 7.2.1), addressing outliers in the studies (Section 7.2.2), and supporting MIMN studies by the community (Section 7.2.3).

7.2.1 Setting MIMN Studies up for Success. Firstly, we want to share the guidelines that set the group up for success, starting with **Team Coordination**, where team members were provided with a digital onboarding support package to each participating researcher at the start of our study to apply at their institution. The package included:

- A common set of Parsons problem studies (Section 4) previously validated through a pilot program. The validated assessment and instruments were designed to collect more accurate data and promote higher **Reliability** for our study results.
- A generic ethics (IRB) approval that researchers can use as a template for their institution's application process to complete the process faster.

To address **Consistent Data Collection**, **Cleanliness**, and **Character of Data**, most of the institutions used Runestone Academy to conduct the Parsons problem studies. In contrast, one institution adopted another platform due to language, which we further discuss in Section 7.2.2. Our Parsons problems studies collected data through Runestone Academy, so we did not have to account for **Grades** across the institutions. Using the same scripts and statistical tests, the quantitative **Analysis Techniques** were aligned across the institutions. For the think-alouds, the facilitators agreed upon a protocol, meeting regularly to discuss the progress and validity. Per the ethics (IRB) approval, each institution was responsible for their study data, following their respective **Data Ownership** policy set by their institution and enabling us to share student data anonymously.

The efforts to conduct this study were possible due to our **Team Coordination**. The working group co-leaders onboarded the institutions **early** and hosted two weekly **meetings** to accommodate the researchers' timezones. The co-leaders ensured we made progress aligned with the **Project's Goals**. Reflecting on our success, we can see the planning, organization, and communication that supported researchers in conducting the Parsons problems studies as the various institutions.

7.2.2 Addressing Outliers in the Studies. A benefit to performing MIMN studies is that it provides diversity in the data to help identify global trends. However, the variety in **Institutional Characteristics** also brings obstacles when applying a homogeneous study across institutions.

Institutional Characteristics include the variety of students' abilities across the participating institutions. We addressed students' abilities by providing a pretest to collect their programming backgrounds. The pretest enabled us to calibrate the results for **Comparing Students' Performance**. In addition, to support student participants with their understanding Parsons problems and encourage data consistency, we provided them with videos on how to use the assessment tool, giving them a hands-on opportunity to practice using the assessment tool before engaging with the treatment. However, institutional characteristics such as course timelines and offerings were a challenge for data analysis, which we discussed further in Section 8. Each institution also had different ways of **Selecting Participants** by making the activity compulsory and non-compulsory. We found that this difference had an impact on volunteers' behavior. For example, the volunteer participants tend to withdraw from studies when more work is required, such as completing the writing code treatment. This student behavior suggests we use shorter activities to support students completing the activity, such as the distractors vs. no distractors activities (See Section 4.6.8) or the python swap activity (See Section4.6.6).

Another outlier was the instructional language used in the course. One of the institutions, whose language of instruction is Spanish, had to translate and validate the Parsons problem studies and used a different tool that provided instructions in Spanish. The translation process was structured as follows: (1) First, one of the co-authors manually translated all prompts, instructions, and test items into Spanish; (2) Next, the translated materials were individually and independently reviewed by three bilingual English-Spanish application domain experts, as a way to assess their understandability and accuracy; (3) Finally, we conducted a small-scale pilot with a sample of five domain experts—instructors and former teaching assistants experienced in teaching the CS1 class at the University—asking them to complete both versions of the experiment (i.e., the original in English and the one translated into Spanish), and then comment on the interpretability, perceived difficulty, and overall assessment of the translation. The goal was to control as much as possible for potential misunderstanding, ambiguity, wording issues, and objective specification of each item.

Similarly, two institutions needed to translate the Parsons problem studies from Python to C and JavaScript (JS). We addressed the issues and considerations in Section 4.

Another challenge was the institutions' course offerings, where the timing of the semesters was not aligned. With **Ethics (IRB) Approval**, we encountered a similar situation as a previous ITiCSE WG [96], where time constraints and different institutions' ethical regulations and policies shorten the timeframe to conduct the studies. We addressed the time constraints by some group members previously working on related studies with existing approvals. For North American and European institutions, the study timing aligned with the ITiCSE WG time constraints, while institutions in Oceania conducted their studies after the ITiCSE conference since courses began at the end of July 2023, and some did not get ethics approval until then.

7.2.3 Future MIMN Considerations and Recommendations. Reflecting on our successes and challenges in conducting a MIMN study, we recognize that completing one can be difficult. The results from our MIMN literature review strengthen our experience that MIMN studies can be hard to conduct, with 17 studies reported in the last ten years. In this section, we suggest how the CSE community can support researchers conducting these types of studies in the future.

Currently, there is support from the community to encourage MIMN studies. The previously mentioned RIPPA paper [72] describes clear guidelines for helping teams to succeed, along with a web presence for more visibility.[3] On the website, RIPPA explains that the MIMN study has a capstone workshop, bringing the researchers together to formalize the work and share with the community, which is currently supported by two UK-based conferences, Computing Education Practice (CEP) and United Kingdom and Ireland Computing Education Researcher (UKICER).

Given the different institutional schedules, our semesters did not align, so not all studies could be conducted simultaneously. Researchers conducting MIMN studies should consider this concern well in advance. Moreover, they should evaluate each institution's schedules and timelines and the courses' schedules and coordinate when each study can be deployed.

In this paper, we share the instruments and their descriptions not only to inform readers but also, to motivate replication. In the future, we would like to prepare a replication package that includes materials from our onboarding package, analysis scripts, and study guidelines. Setting up the studies required guidance from the co-chairs of the working group. Thus, an alternative must be provided to set up the interventions without the co-authors' involvement. The co-authors should also coordinate data ownership protocols after the study is conducted. Even though these are specified in the IRB and can depend on various regulations, the data collected in MIMN studies can be used for follow-up studies.

Our MIMN literature review found cases in which an instrument designed for a specific context is unsuitable for others [119]. Given the nature of MIMN studies, the instruments and design choices must be carefully reviewed to identify (and hopefully fix) potential issues in using them across nations.

7.3 Discussion of Study Results

The 2023 ITiCSE working group conducted some small think-aloud observational studies and several larger quantitative studies. The think-aloud observational studies were conducted with *python-swap*, *class-ta*, and *p3dndta*. The quantitative studies were conducted with *p3pt*, *jspt*, *class-exp*, and *python-swap*.

7.3.1 *Discussion of Think-Alouds.*

As an overall takeaway from the think-alouds, we observed no confusion from students when they completed the surveys and practice pages. This suggests that the interface, and the practice with those interfaces, was effective in teaching students how to interact with both the Parsons problems and write-code problems. This not only contributed to the quality of the results achieved during the think-aloud observations but also contributes to the reliability of the quantitative results.

python-swap: Students found the practice with the Parsons problems to be effective in teaching them how to write code to swap values between variables. It is worth noting that this finding holds true for both of the institutions where *python-swap* think-alouds were conducted. Prior research had already found that Parsons problems are typically enjoyed by students, can serve as effective worked examples [47], and can help students learn common patterns [115].

Students who struggled with the algorithm were typically confused about why they needed a temporary variable. These students mistakenly believed that they could simply set *x = y and y = x*. Many held onto that misconception even though the Parsons problem solution used a temporary variable. One student struggled to solve the first Parsons problem, but then used the solution to the first Parsons problems to solve the other two. However, that student still wrote the incorrect solution of *x = y and y = x* in the first post write-code problem. It was only after that code executed that the student realized that they had a misconception. They used the codelens feature to step through the code and finally realized why they needed a temporary variable. However, they used two temporary variables in their write-code solution since they didn't quite recall the Parsons problem solution.

This suggests that there are some contexts in which Parsons problems alone may be insufficient scaffolding to dislodge certain misconceptions and additional measures may be needed. For example, we could let students run the code first that attempts to simply set the value of *x to y and y to x*. That way they would be primed to learn a new approach.

Finally, many students expressed difficulties when organizing pseudocode comment blocks and preferred blocks that included just code or code and comments. However, these difficulties come with the caveat that it was the first of the practice Parsons problems these students encountered. As such, it is difficult to determine how much of the difficulty students faced is attributable to the form of the problem or simply that it was their first time working through the solution. Further work may be needed to determine if these are desirable difficulties that contribute to students' success on code writing activities. We could try reversing the current order of the Parsons problems. Instead of pseudocode comments first, comments plus code, and then code only we could try code only first, then pseudocode comments plus code, and finally pseudocode comments only.

p3dndta and class-ta: Each of these studies compared practice with and without distractors. These studies were analyzed using a narrative form given the small number of participants (n=3). Students participating in these interviews, though also expressing a positive sentiment towards Parsons problems, had difficulty completing the code writing tasks after practice with the Parsons activities. It is worth noting that these activities covered topics more complex than *python-swap* which may somewhat explain the students difficulties. As for the presence of distractors, students in all interviews contended with them while solving those problems and often included them in their solutions. However, given the small sample size and the baseline difficulty of the topics at hand, it is not clear from the interviews if those problems that included distractors were substantively more difficult. In one case, a student did vocalize after selecting a distractor that they "would never forget that [correct syntax] again" suggesting that examples of incorrect code may be effective at teaching students to notice common errors.

[3]https://rippa.co.uk

7.3.2 Discussion of Quantitative Studies. We conducted several studies at more than one institution.

python-swap. This study was conducted at two institutions with a total of x participants. It tested if students could write code to swap the values of two variables after solving several Parsons problems. Most students (80%) could successfully write code to swap the values of two variables after solving three Parsons problems - one with only pseudocode comments that explain the steps of the algorithm, one with the same comments and code, and one with just code. This finding strengthens the evidence that solving Parsons problems can help students learn to reproduce common algorithms.

However, there was a noticeable drop in the percentage of students who completed the first Parsons problem - the one with only pseudocode comments. Students required more attempts to solve this problem than subsequent problems; typically, taking a mean of six attempts. This matches the findings from the think-aloud observations that some students find it much harder to solve a Parsons problem with just pseudocode comments rather than code or comments and code. This indicates that more studies should be run to test the result from reversing this order, i.e. starting with code only, then code with comments, and finally just comments.

The code writing exercises were typically completed in fewer attempts and in less time than the Parsons problems. The second write-code problem results also suggest some degree of near-transfer, with students extending their reasoning beyond just copying the code from the previous Parsons problem.

p3pt. This study was conducted at four institutions with a total of 369 participants. It compared solving Parsons problems to writing the equivalent code. There was no significant difference in learning performance by condition which replicates prior findings. However, there was also no significant difference in practice time, which is different from previous research that found that students can often complete Parsons problems significantly faster than writing the equivalent code, unless a Parsons problem solution is unusual [25, 28]. More work needs to be done to test the learning efficiency of solving Parsons problems versus writing the equivalent code.

class-exp. This study was conducted at five institutions with a total of 361 participants. It compared solving Parsons problems with distractors versus no distractors. While those in the distractor condition had a higher average score on the posttest than those in the no distractor condition, the difference was not statistically significant. However, students in the Parsons problems with distractors condition had significantly less errors while writing code for the posttest questions than those in the Parsons problem without distractors condition. This supports the hypothesis that solving Parsons problems with distractors can help students recognize and avoid common errors. In addition, students with low self-efficacy and low pre-existing knowledge were more likely to attempt the posttest problems if they were in the condition with distractors.

jspt. Like *python-swap*, the qualitative and quantitative results suggest that students found value in the Parsons Problems as they helped them, to some extent, in completing isomorphic write-code assignments. However, in most cases, participants did not recognize the isomorphic relation between the problem types. A considerable number of students faced difficulties completing the tasks, citing lack of time and a prevalent misconception with respect to foundational concepts regarding lists/arrays and abstraction. It is worth pointing out that the participants that took part in this experiment were very novice programmers, without a strong background in STEM-related fields. Nevertheless, the study findings give us a first idea of how to adapt Parsons Problems in adult education, particularly when they come from diverse backgrounds.

8 LIMITATIONS

There are limitations and threats to validity to implementing a MIMN collaborative study as an ITiCSE working group. Due to the ITiCSE Working Group timing constraints of accepting the working group proposal in January and adding working group Members through March 2023, course schedules in Oceania and South America institutions were not aligned with the timing to report findings for the conference's publication deadline. As a result, these institutions had to collect and analyze data post-conference. This Working Group would have benefited from starting the preparation earlier to receive approvals to the IRBs in time to perform studies in their scheduled courses.

There are likely limitations to our Parsons problems and MIMN literature reviews. Firstly, both literature reviews used the specific libraries to identify papers, potentially excluding works published in other venues. Another limitation relates to content validity for the MIMN literature review since our search reliability depends on the papers labeling their intention as a MIMN study. It is also possible researchers applied other terms to describe a MIMN study or place a priority on study's context across countries and institutions to feature in their paper. We cannot say our review identified all MIMN papers; however, multiple co-authors evaluated the papers' contents to ensure they met the selection criteria.

9 CONCLUSION

The 2023 ITiCSE Parsons problems working group leveraged the work of the 2022 ITiCSE Parsons problem working group, which designed several studies in Python, created 'study-in-a-box' materials, and piloted two of the sets of 'study-in-a-box' materials. The current timeline for ITiCSE working groups makes it difficult to design, pilot, and also conduct research studies at various institutions and nations in the allotted time. To better match their institutional context, some of the 2023 ITiCSE working group members translated studies to other programming languages (JavaScript and C) and another to a natural language (Spanish). To reduce the typical problems with MIMN studies concerning differing user interfaces, data collection, cleaning, and analysis, we originally intended all the studies to be conducted on the Runestone Academy platform. However, one of the institutions utilized a local platform due to potential language barriers and accessibility concerns that could have introduced unwanted biases in the data collection. We found that it was more expedient for some of the institutions to run think-aloud observational studies than A/B experimental studies, which was also recommended by the MIMN literature review. Still, some of our institutions were able to conduct quantitative studies which provide evidence for the benefits of solving Parsons problems with distractors and for learning common algorithms. The think-aloud observations also provided suggestions for ways to improve the study materials. With our work, we hope to motivate further work in Parsons problems MIMN studies and contribute to previous work in MIMN research by sharing our experiences and recommendations.

10 ACKNOWLEDGEMENTS

The 2023 ITiCSE Parsons problems working group would like to acknowledge support from the Fulbright U.S. Scholar Program and sabbatical support from Berea College, both of which facilitated the cultural connections that made the IRB and data collection possible at Ashesi University in Ghana. In addition we thank Craig Miller of DePaul University for piloting studies and contributing data to the class-exp study.

REFERENCES

[1] Robert K Atkinson, Sharon J Derry, Alexander Renkl, and Donald Wortham. 2000. Learning from examples: Instructional principles from the worked examples research. *Review of educational research* 70, 2 (2000), 181–214.

[2] Albert Bandura. 1997. *Self-efficacy: The exercise of control.* Worth Publishers, New York, NY.

[3] Brett A. Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J. Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, Janice L. Pearce, and James Prather. 2019. Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education* (Aberdeen, Scotland Uk) *(ITiCSE-WGR '19)*. ACM, New York, NY, USA, 177–210. https://doi.org/10.1145/3344429.3372508

[4] Sarah Beecham, John Noll, and Tony Clear. 2017. Do We Teach the Right Thing? A Comparison of GSE Education and Practice. In *2017 IEEE 12th International Conference on Global Software Engineering (ICGSE)*. IEEE, Buenos Aires, Argentina, 11–20. https://doi.org/10.1109/ICGSE.2017.8

[5] Klara Benda, Amy Bruckman, and Mark Guzdial. 2012. When life and learning do not fit: Challenges of workload and communication in introductory computer science online. *ACM Transactions on Computing Education (TOCE)* 12, 4 (2012), 1–38.

[6] Jeff Bender, Bingpu Zhao, Alex Dziena, and Gail Kaiser. 2022. Learning Computational Thinking Efficiently How Parsons Programming Puzzles within Scratch Might Help. In *Proceedings of the Twenty-Fourth Australasian Computing Education Conference*. ACM, Online, 66–75.

[7] Sylvia Beyer, Kristina Rynes, Julie Perrault, Kelly Hay, and Susan Haller. 2003. Gender differences in computer science students. In *Proceedings of the 34th SIGCSE technical symposium on Computer science education*. ACM, Reno, NV, USA, 49–53.

[8] Elizabeth Ligon Bjork, Jeri L Little, and Benjamin C Storm. 2014. Multiple-choice testing as a desirable difficulty in the classroom. *Journal of Applied Research in Memory and Cognition* 3, 3 (2014), 165–170.

[9] Robert A Bjork. 2017. Creating desirable difficulties to enhance learning. In *Best of the Best: Progress (Best of the Best series)*. Crown House Publishing, Carmarthen, UK.

[10] A. Booth, A. Sutton, and D. Papaioannou. 2016. *Systematic Approaches to a Successful Literature Review.* Sage, London. https://eprints.whiterose.ac.uk/105755/ © 2016 Andrew Booth, Anthea Sutton and Diana Papaioannou.

[11] Dennis Bouvier, Ellie Lovellette, John Matta, Bedour Alshaigy, Brett A. Becker, Michelle Craig, Jana Jackova, Robert McCartney, Kate Sanders, and Mark Zarb. 2016. Novice Programmers and the Problem Description Effect. In *Proceedings of the 2016 ITiCSE Working Group Reports* (Arequipa, Peru) *(ITiCSE '16)*. Association for Computing Machinery, New York, NY, USA, 103–118. https://doi.org/10.1145/3024906.3024912

[12] John D Bransford, Ann L Brown, and Rodney R Cocking. 2000. *How people learn.* Vol. 11. National academy press, Washington DC, USA.

[13] Pearl Brereton, Barbara A. Kitchenham, David Budgen, Mark Turner, and Mohamed Khalil. 2007. Lessons from applying the systematic literature review process within the software engineering domain. *Journal of Systems and Software* 80, 4 (2007), 571–583. https://doi.org/10.1016/j.jss.2006.07.009 Software Performance.

[14] Stephen D Brookfield. 2017. *Becoming a critically reflective teacher* (2 ed.). Jossey-Bass, London, England.

[15] Peter Brusilovsky, Barbara J Ericson, Cay S Horstmann, Christian Servin, Frank Vahid, and Craig Zilles. 2023. *The Future of Computing Education Materials.* Technical Report. ACM.

[16] Aparna Chirumamilla and Guttorm Sindre. 2019. E-Assessment in Programming Courses: Towards a Digital Ecosystem Supporting Diverse Needs?. In *Digital Transformation for a Sustainable Society in the 21st Century: 18th IFIP WG 6.11 Conference on e-Business, e-Services, and e-Society, I3E 2019, Trondheim, Norway, September 18–20, 2019, Proceedings 18.* Springer, Trondheim, Norway, 585–596.

[17] Holger Danielsiek, Laura Toma, and Jan Vahrenhold. 2017. An Instrument to Assess Self-Efficacy in Introductory Algorithms Courses. In *Proceedings of the 2017 ACM Conference on International Computing Education Research* (Tacoma, Washington, USA) *(ICER '17)*. Association for Computing Machinery, New York, NY, USA, 217–225. https://doi.org/10.1145/3105726.3106171

[18] Paul Denny, Andrew Luxton-Reilly, and Beth Simon. 2008. Evaluating a New Exam Question: Parsons Problems. In *Proceedings of the Fourth International Workshop on Computing Education Research* (Sydney, Australia) *(ICER '08)*. Association for Computing Machinery, New York, NY, USA, 113–124. https://doi.org/10.1145/1404520.1404532

[19] Paul Denny, James Prather, Brett A Becker, Zachary Albrecht, Dastyni Loksa, and Raymond Pettit. 2019. A Closer Look at Metacognitive Scaffolding: Solving Test Cases Before Programming. In *Proceedings of the 19th Koli Calling International Conference on Computing Education Research.* ACM, New York, NY, USA, 1–10.

[20] Yuemeng Du, Andrew Luxton-Reilly, and Paul Denny. 2020. A Review of Research on Parsons Problems. In *Proceedings of the Twenty-Second Australasian Computing Education Conference* (Melbourne, VIC, Australia) *(ACE'20)*. Association for Computing Machinery, New York, NY, USA, 195–202. https://doi.org/10.1145/3373165.3373187

[21] Rodrigo Duran, Jan-Mikael Rybicki, Juha Sorva, and Arto Hellas. 2019. Exploring the Value of Student Self-Evaluation in Introductory Programming. In *Proceedings of the 2019 ACM Conference on International Computing Education Research* (Toronto ON, Canada) *(ICER '19)*. Association for Computing Machinery, New York, NY, USA, 121–130. https://doi.org/10.1145/3291279.3339407

[22] Carol S Dweck. 1986. Motivational processes affecting learning. *American psychologist* 41, 10 (1986), 1040.

[23] Jacquelynne Eccles. 2009. Who am I and what am I going to do with my life? Personal and collective identities as motivators of action. *Educational psychologist* 44, 2 (2009), 78–89.

[24] Elsa Eiriksdottir and Richard Catrambone. 2011. Procedural instructions, principles, and examples: How to structure instructions for procedural tasks to enhance performance, learning, and transfer. *Human factors* 53, 6 (2011), 749–770.

[25] Barbara Ericson and Carl Haynes-Magyar. 2022. Adaptive Parsons Problems as Active Learning Activities During Lecture. In *Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education Vol. 1* (Dublin, Ireland) *(ITiCSE '22)*. Association for Computing Machinery, New York, NY, USA, 290–296. https://doi.org/10.1145/3502718.3524808

[26] Barbara Ericson, Austin McCall, and Kathryn Cunningham. 2019. Investigating the affect and effect of adaptive parsons problems. In *Proceedings of the 19th Koli Calling International Conference on Computing Education Research.* ACM, New York, NY, USA, 1–10.

[27] Barbara J. Ericson, Paul Denny, James Prather, Rodrigo Duran, Arto Hellas, Juho Leinonen, Craig S. Miller, Briana B. Morrison, Janice L. Pearce, and Susan H. Rodger. 2022. Parsons Problems and Beyond: Systematic Literature Review and Empirical Study Designs. In *Proceedings of the 2022 Working Group Reports on Innovation and Technology in Computer Science Education* (Dublin, Ireland) *(ITiCSE-WGR '22)*. Association for Computing Machinery, New York, NY, USA, 191–234. https://doi.org/10.1145/3571785.3574127

[28] Barbara J Ericson, James D Foley, and Jochen Rick. 2018. Evaluating the efficiency and effectiveness of adaptive parsons problems. In *Proceedings of the 2018 ACM Conference on International Computing Education Research.* ACM, New York, NY, USA, 60–68.

[29] Barbara J. Ericson, Mark J. Guzdial, and Briana B. Morrison. 2015. Analysis of Interactive Features Designed to Enhance Learning in an Ebook. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research* (Omaha, Nebraska, USA) *(ICER '15)*. Association for Computing Machinery, New York, NY, USA, 169–178. https://doi.org/10.1145/2787622.2787731

[30] Barbara J Ericson, Lauren E Margulieux, and Jochen Rick. 2017. Solving parsons problems versus fixing and writing code. In *Koli Calling '17: Proceedings of the 17th Koli Calling International Conference on Computing Education Research.* ACM, New York, NY, USA, 1–10.

[31] Barbara J Ericson and Bradley N Miller. 2020. Runestone: A Platform for Free, Online, and Interactive Ebooks. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education.* ACM, New York, NY, USA, 1012–1018.

[32] K Anders Ericsson, Ralf T Krampe, and Clemens Tesch-Römer. 1993. The role of deliberate practice in the acquisition of expert performance. *Psychological review* 100, 3 (1993), 363.

[33] José Figueiredo and Francisco José García-Peñalvo. 2022. Strategies to increase success in learning programming. In *2022 International Symposium on Computers in Education (SIIE)*. IEEE, New York, NY, 1–6.

[34] Sally Fincher, Raymond Lister, Tony Clear, Anthony Robins, Josh Tenenberg, and Marian Petre. 2005. Multi-Institutional, Multi-National Studies in CSEd Research: Some Design Considerations and Trade-Offs. In *Proceedings of the First International Workshop on Computing Education Research* (Seattle, WA,

USA) *(ICER '05)*. Association for Computing Machinery, New York, NY, USA, 111–121. https://doi.org/10.1145/1089786.1089797

[35] Flynn Fromont, Hiruna Jayamanne, and Paul Denny. 2023. Exploring the Difficulty of Faded Parsons Problems for Programming Education. In *Proceedings of the 25th Australasian Computing Education Conference*. Association for Computing Machinery, New York, NY, USA, 113–122.

[36] Rita Garcia. 2021. Evaluating Parsons Problems as a Design-Based Intervention. In *2021 IEEE Frontiers in Education Conference (FIE)*. IEEE, IEEE, New York, NY, 1–9.

[37] Rita Garcia, Katrina Falkner, and Rebecca Vivian. 2018. Scaffolding the Design Process Using Parsons Problems. In *Proceedings of the 18th Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '18)*. Association for Computing Machinery, New York, NY, USA, Article 26, 2 pages. https://doi.org/10.1145/3279720.3279746

[38] Fernand Gobet and Herbert A Simon. 1996. Recall of random and distorted chess positions: Implications for the theory of expertise. *Memory & cognition* 24, 4 (1996), 493–503.

[39] Scott Grissom, Renée Mccauley, and Laurie Murphy. 2017. How Student Centered is the Computer Science Classroom? A Survey of College Faculty. *ACM Trans. Comput. Educ.* 18, 1, Article 5 (nov 2017), 27 pages. https://doi.org/10.1145/3143200

[40] Shuchi Grover, Brian Broll, and Derek Babb. 2023. Cybersecurity Education in the Age of AI: Integrating AI Learning into Cybersecurity High School Curricula. In *Proceedings of the fifty-fourth ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2023),*. ACM, New York, NY, 980–986.

[41] Kyle James Harms, Jason Chen, and Caitlin L. Kelleher. 2016. Distractors in Parsons Problems Decrease Learning Efficiency for Young Novice Programmers. In *Proceedings of the 2016 ACM Conference on International Computing Education Research* (Melbourne, VIC, Australia) *(ICER '16)*. Association for Computing Machinery, New York, NY, USA, 241–250. https://doi.org/10.1145/2960310.2960314

[42] Devamardeep Hayatpur, Tehilla Helfenbaum, Haijun Xia, Wolfgang Stuerzlinger, and Paul Gries. 2023. Structuring Collaboration in Programming Through Personal-Spaces. In *Extended Abstracts of the 2023 CHI Conference on Human Factors in Computing Systems* (Hamburg, Germany) *(CHI EA '23)*. Association for Computing Machinery, New York, NY, USA, Article 263, 7 pages.

[43] Carl Haynes-Magyar and Barbara Ericson. 2022. The Impact of Solving Adaptive Parsons Problems with Common and Uncommon Solutions. In *Proceedings of the 22nd Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '22)*. Association for Computing Machinery, New York, NY, USA, Article 23, 14 pages. https://doi.org/10.1145/3564721.3564736

[44] Juha Helminen, Petri Ihantola, Ville Karavirta, and Lauri Malmi. 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*. Association for Computing Machinery, New York, NY, USA, 119–126.

[45] Roya Hosseini, Kamil Akhuseyinoglu, Peter Brusilovsky, Lauri Malmi, Kerttu Pollari-Malmi, Christian Schunn, and Teemu Sirkia. 2020. Improving Engagement in Program Construction Examples for Learning Python Programming. *International Journal of Artificial Intelligence in Education* 30 (2020), 299–336.

[46] Roya Hosseini, Kamil Akhuseyinoglu, Andrew Petersen, Christian D Schunn, and Peter Brusilovsky. 2018. PCEX: interactive program construction examples for learning programming. In *Proceedings of the 18th Koli Calling International Conference on Computing Education Research*. Association for Computing Machinery, New York, NY, USA, 1–9.

[47] Xinying Hou, Barbara Jane Ericson, and Xu Wang. 2022. Using Adaptive Parsons Problems to Scaffold Write-Code Problems. In *Proceedings of the 2022 ACM Conference on International Computing Education Research V. 1*. Association for Computing Machinery, New York, NY, USA, 15–26.

[48] Xinying Hou, Barbara Jane Ericson, and Xu Wang. 2023. Parsons Problems to Scaffold Code Writing: Impact on Performance and Problem-Solving Efficiency. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 2*. ACM, New York, NY, 665–665.

[49] Slava Kalyuga, Paul Ayres, Paul Chandler, and John Sweller. 2003. The Expertise Reversal Effect. *Educational Psychologist* 38, 1 (2003), 23–31. https://doi.org/10.1207/S15326985EP3801_4

[50] Sandra Katz, David Allbritton, John Aronis, Christine Wilson, and Mary Lou Soffa. 2006. Gender, achievement, and persistence in an undergraduate computer science program. *ACM SIGMIS Database: the DATABASE for Advances in Information Systems* 37, 4 (2006), 42–57.

[51] United Kingdom and Ireland Computing Education Research (UKICER) Conference. 2023. *UKICER 2023: Call for Participation*. UK ACM SIGCSE. Retrieved July 8, 2023 from https://www.ukicer.com/participation.html

[52] Paivi Kinnunen and Beth Simon. 2010. Experiencing programming assignments in CS1: the emotional toll. In *Proceedings of the Sixth international workshop on Computing education research*. Association for Computing Machinery, New York, NY, USA, 77–86.

[53] Päivi Kinnunen and Beth Simon. 2011. CS majors' self-efficacy perceptions in CS1: results in light of social cognitive theory. In *Proceedings of the seventh international workshop on Computing education research*. Association for Computing Machinery, New York, NY, USA, 19–26.

[54] Jo-Anne LeFevre and Peter Dixon. 1986. Do written instructions need examples? *Cognition and Instruction* 3, 1 (1986), 1–30.

[55] Raymond Lister, Elizabeth S. Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, Beth Simon, and Lynda Thomas. 2004. A Multi-National Study of Reading and Tracing Skills in Novice Programmers. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education* (Leeds, United Kingdom) *(ITiCSE-WGR '04)*. Association for Computing Machinery, New York, NY, USA, 119–150. https://doi.org/10.1145/1044550.1041673

[56] Nelson Lojo and Armando Fox. 2022. Teaching Test-Writing as a Variably-Scaffolded Programming Pattern. In *Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education Vol. 1*. ACM, New York, NY, 498–504.

[57] Dastyni Loksa, Lauren Margulieux, Brett A. Becker, Michelle Craig, Paul Denny, Raymond Pettit, and James Prather. 2022. Metacognition and Self-Regulation in Programming Education: Theories and Exemplars of Use. *ACM Trans. Comput. Educ.* 22, 4 (dec 2022), 1–31. https://doi.org/10.1145/3487050

[58] Andrew Luxton-Reilly and Andrew Petersen. 2017. The Compound Nature of Novice Programming Assessments. In *Proceedings of the Nineteenth Australasian Computing Education Conference* (Geelong, VIC, Australia) *(ACE '17)*. Association for Computing Machinery, New York, NY, USA, 26–35. https://doi.org/10.1145/3013499.3013500

[59] Andrew Luxton-Reilly, Simon, Ibrahim Albluwi, Brett A. Becker, Michail Giannakos, Amruth N. Kumar, Linda Ott, James Paterson, Michael James Scott, Judy Sheard, and Claudia Szabo. 2018. Introductory Programming: A Systematic Literature Review. In *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education* (Larnaca, Cyprus) *(ITiCSE 2018 Companion)*. Association for Computing Machinery, New York, NY, USA, 55–106. https://doi.org/10.1145/3293881.3295779

[60] Jane Margolis. 2017. *Stuck in the Shallow End, updated edition: Education, Race, and Computing*. MIT press, Cambridge, Massachusetts.

[61] Jane Margolis and Allan Fisher. 2002. *Unlocking the clubhouse: Women in computing*. MIT press, Cambridge, Massachusetts.

[62] Catherine Marshall and Gretchen B. Rossman. 1999. *Designing Qualitative Research* (3rd ed.). Sage Publications, London.

[63] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. 2001. A Multi-National, Multi-Institutional Study of Assessment of Programming Skills of First-Year CS Students. *SIGCSE Bull.* 33, 4 (dec 2001), 125–180. https://doi.org/10.1145/572139.572181

[64] Brad Miller and David Ranum. 2014. Runestone Interactive: Tools for Creating Interactive Course Materials. In *Proceedings of the First ACM Conference on Learning @ Scale Conference* (Atlanta, Georgia, USA) *(L@S '14)*. Association for Computing Machinery, New York, NY, USA, 213–214. https://doi.org/10.1145/2556325.2567887

[65] George A Miller. 1956. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological review* 63, 2 (1956), 81.

[66] Briana B. Morrison, Lauren E. Margulieux, Barbara Ericson, and Mark Guzdial. 2016. Subgoals Help Students Solve Parsons Problems. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. Association for Computing Machinery, New York, NY, USA, 42–47.

[67] Kasia Muldner, Jay Jennings, and Veronica Chiarelli. 2022. A Review of Worked Examples in Programming Activities. *ACM Transactions on Computing Education* 23, 1 (2022), 1–35.

[68] Fred Paas, Alexander Renkl, and John Sweller. 2003. Cognitive load theory and instructional design: Recent developments. *Educational psychologist* 38, 1 (2003), 1–4.

[69] Fred Paas, Tamara Van Gog, and John Sweller. 2010. Cognitive load theory: New conceptualizations, specifications, and integrated research perspectives. *Educational psychology review* 22, 2 (2010), 115–121.

[70] Jennifer Parham-Mocello, Martin Erwig, Margaret Niess, Jason Weber, Madelyn Smith, and Garrett Berliner. 2023. Putting Computing on the Table: Using Physical Games to Teach Computer Science. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. ACM, New York, NY, 444–450.

[71] Miranda C. Parker, Mark Guzdial, and Shelly Engleman. 2016. Replication, Validation, and Use of a Language Independent CS1 Knowledge Assessment. In *Proceedings of the 2016 ACM Conference on International Computing Education Research* (Melbourne, VIC, Australia) *(ICER '16)*. Association for Computing Machinery, New York, NY, USA, 93–101. https://doi.org/10.1145/2960310.2960316

[72] Jack Parkinson, Sebastian Dziallas, Gary Lewandowski, Fiona Mcneill, Jim Williams, and Quintin Cutts. 2022. Experience Report: Running and Participating in a Multi-Institutional Research in Practice Project Activity (RIPPA). In *Proceedings of the 2022 Conference on United Kingdom & Ireland Computing Education*

Research (Dublin, Ireland) *(UKICER '22)*. Association for Computing Machinery, New York, NY, USA, Article 4, 7 pages. https://doi.org/10.1145/3555009.3555014

[73] Dale Parsons and Patricia Haden. 2006. Parson's Programming Puzzles: A Fun and Effective Learning Tool for First Programming Courses. In *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52* (Hobart, Australia) *(ACE '06)*. Australian Computer Society, Inc., AUS, 157–163.

[74] Dale Parsons, Krissi Wood, and Patricia Haden. 2015. What are we doing when we assess programming. In *Proceedings of the 17th Australasian Computing Education Conference (ACE 2015)*, Vol. 27. Australian Computer Society, Inc., AUS, 30.

[75] Yulia Pechorina, Keith Anderson, and Paul Denny. 2023. Metacodenition: Scaffolding the Problem-Solving Process for Novice Programmers. In *Proceedings of the 25th Australasian Computing Education Conference*. Association for Computing Machinery, New York, NY, USA, 59–68.

[76] Peter L Pirolli and John R Anderson. 1985. The role of learning from examples in the acquisition of recursive programming skills. *Canadian Journal of Psychology/Revue canadienne de psychologie* 39, 2 (1985), 240.

[77] Leo Porter, Dennis Bouvier, Quintin Cutts, Scott Grissom, Cynthia Lee, Robert McCartney, Daniel Zingaro, and Beth Simon. 2016. A Multi-Institutional Study of Peer Instruction in Introductory Computing. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education* (Memphis, Tennessee, USA) *(SIGCSE '16)*. Association for Computing Machinery, New York, NY, USA, 358–363. https://doi.org/10.1145/2839509.2844642

[78] James Prather, Brett A Becker, Michelle Craig, Paul Denny, Dastyni Loksa, and Lauren Margulieux. 2020. What do we think we think we are doing? Metacognition and self-regulation in programming. In *Proceedings of the 2020 ACM Conference on International Computing Education Research*. Association for Computing Machinery, New York, NY, USA, 2–13.

[79] James Prather, John Homer, Paul Denny, Brett Becker, John Marsden, and Garrett Powell. 2022. Scaffolding Task Planning Using Abstract Parsons Problems. In *Proceedings of the 2022 World Conference on Computers in Education (WCCE '22)*. IFIP, Japan, 1–10.

[80] James Prather, Lauren Margulieux, Jacqueline Whalley, Paul Denny, Brent N Reeves, Brett A Becker, Paramvir Singh, Garrett Powell, and Nigel Bosch. 2022. Getting By With Help From My Friends: Group Study in Introductory Programming Understood as Socially Shared Regulation. In *Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 1*. Association for Computing Machinery, New York, NY, USA, 164–176.

[81] James Prather, Raymond Pettit, Brett A Becker, Paul Denny, Dastyni Loksa, Alani Peters, Zachary Albrecht, and Krista Masci. 2019. First things first: Providing metacognitive scaffolding for interpreting problem prompts. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. Association for Computing Machinery, New York, NY, USA, 531–537.

[82] James Prather, Raymond Pettit, Kayla McMurry, Alani Peters, John Homer, and Maxine Cohen. 2018. Metacognitive difficulties faced by novice programmers in automated assessment tools. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*. Association for Computing Machinery, New York, NY, USA, 41–50.

[83] James Prather, Raymond Pettit, Kayla Holcomb McMurry, Alani Peters, John Homer, Nevan Simone, and Maxine Cohen. 2017. On novices' interaction with compiler error messages: A human factors approach. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*. Association for Computing Machinery, New York, NY, USA, 74–82.

[84] Integrating Parsons puzzles within Scratch enables efficient computational thinking learning. 2023. Integrating Parsons puzzles within Scratch enables efficient computational thinking learning. *Integrating Parsons puzzles within Scratch enables efficient computational thinking learning* 18, 22 (2023), 25.

[85] Yizhou Qian and James Lehman. 2017. Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Transactions on Computing Education (TOCE)* 18, 1 (2017), 1–24.

[86] Keith Quille and Susan Bergin. 2018. Programming: Predicting Student Success Early in CS1. a Re-Validation and Replication Study. In *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education* (Larnaca, Cyprus) *(ITiCSE 2018)*. Association for Computing Machinery, New York, NY, USA, 15–20. https://doi.org/10.1145/3197091.3197101

[87] Keith Quille, Natalie Culligan, and Susan Bergin. 2017. Insights on Gender Differences in CS1: A Multi-Institutional, Multi-Variate Study.. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education* (Bologna, Italy) *(ITiCSE '17)*. Association for Computing Machinery, New York, NY, USA, 263–268. https://doi.org/10.1145/3059009.3059048

[88] Keith Quille, Soohyun Nam Liao, Eileen Costelloe, Keith Nolan, Aidan Mooney, and Kartik Shah. 2022. PreSS: Predicting Student Success Early in CS1. A Pilot International Replication and Generalization Study. In *Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education Vol. 1* (Dublin, Ireland) *(ITiCSE '22)*. Association for Computing Machinery, New York, NY, USA, 54–60. https://doi.org/10.1145/3502718.3524755

[89] Brent Reeves, Sami Sarsa, James Prather, Paul Denny, Brett A. Becker, Arto Hellas, Bailey Kimmel, Garrett Powell, and Juho Leinonen. 2023. Evaluating the Performance of Code Generation Models for Solving Parsons Problems With Small Prompt Variations. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*. Association for Computing Machinery, New York, NY, USA, 299–305.

[90] Alexander Renkl. 2005. The worked-out-example principle in multimedia learning. *The Cambridge handbook of multimedia learning* 1 (2005), 229–245.

[91] Emma Riese, Madeleine Lorås, Martin Ukrop, and Tomáš Effenberger. 2021. Challenges Faced by Teaching Assistants in Computer Science Education Across Europe. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1* (Virtual Event, Germany) *(ITiCSE '21)*. Association for Computing Machinery, New York, NY, USA, 547–553. https://doi.org/10.1145/3430665.3456304

[92] Judy Sheard, Simon, Julian Dermoudy, Daryl D'Souza, Minjie Hu, and Dale Parsons. 2014. Benchmarking a Set of Exam Questions for Introductory Programming. In *Proceedings of the Sixteenth Australasian Computing Education Conference - Volume 148* (Auckland, New Zealand) *(ACE '14)*. Australian Computer Society, Inc., AUS, 113–121.

[93] Yu Sheng, Bin Li, Zequan Wu, Ping Zhong, and Guihua Duan. 2022. AC Language Learning Platform Based on Parsons Problems. In *International Conference on Computer Science and Education*. Springer, Association for Computing Machinery, New York, NY, USA, 541–552.

[94] Dermot Shinners-Kennedy and Sally A. Fincher. 2013. Identifying Threshold Concepts: From Dead End to a New Direction. In *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research* (San Diego, San California, USA) *(ICER '13)*. Association for Computing Machinery, New York, NY, USA, 9–18. https://doi.org/10.1145/2493394.2493396

[95] Angela A. Siegel, Mark Zarb, Bedour Alshaigy, Jeremiah Blanchard, Tom Crick, Richard Glassey, John R. Hott, Celine Latulipe, Charles Riedesel, Mali Senapathi, Simon, and David Williams. 2022. Teaching through a Global Pandemic: Educational Landscapes Before, During and After COVID-19. In *Proceedings of the 2021 Working Group Reports on Innovation and Technology in Computer Science Education* (Virtual Event, Germany) *(ITiCSE-WGR '21)*. Association for Computing Machinery, New York, NY, USA, 1–25.

[96] Angela A. Siegel, Mark Zarb, Emma Anderson, Brent Crane, Alice Gao, Celine Latulipe, Ellie Lovellette, Fiona McNeill, and Debbie Meharg. 2022. The Impact of COVID-19 on the CS Student Learning Experience: How the Pandemic Has Shaped the Educational Landscape. In *Proceedings of the 2022 Working Group Reports on Innovation and Technology in Computer Science Education* (Dublin, Ireland) *(ITiCSE-WGR '22)*. Association for Computing Machinery, New York, NY, USA, 165–190. https://doi.org/10.1145/3571785.3574126

[97] Teemu Sirkiä. 2016. Combining Parson's problems with program visualization in CS1 context. In *Proceedings of the 16th Koli Calling International Conference on Computing Education Research*. Association for Computing Machinery, New York, NY, USA, 155–159.

[98] John A Sloboda, Jane W Davidson, Michael JA Howe, and Derek G Moore. 1996. The role of practice in the development of performing musicians. *British journal of psychology* 87, 2 (1996), 287–309.

[99] David Smith and Craig Zilles. 2023. Discovering, Autogenerating, and Evaluating Distractors for Python Parsons Problems in CS1. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2023)*. ACM, New York, NY, 924–930.

[100] David H. Smith, Max Fowler, and Craig Zilles. 2023. Investigating the Role and Impact of Distractors on Parsons Problems in CS1 Assessments. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*. Association for Computing Machinery, New York, NY, USA, 417–423.

[101] Sylvia Stuurman, Harrie Passier, and Erik Barendsen. 2016. Analyzing Students' Software Redesign Strategies. In *Proceedings of the 16th Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '16)*. Association for Computing Machinery, New York, NY, USA, 110–119. https://doi.org/10.1145/2999541.2999559

[102] Lovisa Sundin, Nourhan Sakr, Juho Leinonen, Sherif Aly, and Quintin Cutts. 2021. Visual Recipes for Slicing and Dicing Data: Teaching Data Wrangling Using Subgoal Graphics. In *Proceedings of the 21st Koli Calling International Conference on Computing Education Research* (Joensuu, Finland) *(Koli Calling '21)*. Association for Computing Machinery, New York, NY, USA, Article 29, 10 pages. https://doi.org/10.1145/3488042.3488063

[103] John Sweller. 1988. Cognitive load during problem solving: Effects on learning. *Cognitive science* 12, 2 (1988), 257–285.

[104] John Sweller. 2006. The worked example effect and human cognition. *Learning and instruction* 16, 2 (2006), 165–169.

[105] John Sweller, Jeroen JG van Merriënboer, and Fred Paas. 2019. Cognitive architecture and instructional design: 20 years later. *Educational Psychology Review* 31 (2019), 261–292.

[106] Kok Cheng Tan, Daniel Zantedeschi, Amruth Kumar, and Alessio Gaspar. 2022. Genetic algorithm cleaning in sequential data mining: analyzing solutions to parsons' puzzles. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. Association for Computing Machinery, New York, NY, USA, 2330–2333.

[107] Dirk Tempelaar, Bart Rienties, and Quan Nguyen. 2018. Investigating Learning Strategies in a Dispositional Learning Analytics Context: The Case of Worked Examples. In *Proceedings of the 8th International Conference on Learning Analytics and Knowledge* (Sydney, New South Wales, Australia) *(LAK '18)*. Association for Computing Machinery, New York, NY, USA, 201–205. https://doi.org/10.1145/3170358.3170385

[108] John Gregory Trafton and Brian J Reiser. 1994. *The contributions of studying examples and solving problems to skill acquisition.* Ph. D. Dissertation. Citeseer.

[109] Ian Utting, Allison Elliott Tew, Mike McCracken, Lynda Thomas, Dennis Bouvier, Roger Frye, James Paterson, Michael Caspersen, Yifat Ben-David Kolikant, Juha Sorva, and Tadeusz Wilusz. 2013. A Fresh Look at Novice Programmers' Performance and Their Teachers' Expectations. In *Proceedings of the ITiCSE Working Group Reports Conference on Innovation and Technology in Computer Science Education-Working Group Reports* (Canterbury, England, United Kingdom) *(ITiCSE -WGR '13)*. Association for Computing Machinery, New York, NY, USA, 15–32. https://doi.org/10.1145/2543882.2543884

[110] Jeroen JG Van Merriënboer and Marcel BM De Croock. 1992. Strategies for computer-based programming instruction: Program completion vs. program generation. *Journal of Educational Computing Research* 8, 3 (1992), 365–394.

[111] Valdemar Švábenský, Richard Weiss, Jack Cook, Jan Vykopal, Pavel Čeleda, Jens Mache, Radoslav Chudovský, and Ankur Chattopadhyay. 2022. Evaluating Two Approaches to Assessing Student Progress in Cybersecurity Exercises. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education - Volume 1* (Providence, RI, USA) *(SIGCSE 2022)*. Association for Computing Machinery, New York, NY, USA, 787–793. https://doi.org/10.1145/3478431.3499414

[112] Lev S Vygotsky. 1978. Mind in society: The development of higher mental processes (E. Rice, Ed. & Trans.).

[113] Lev Semenovich Vygotsky. 1980. *Mind in society: The development of higher psychological processes.* Harvard university press, Cambridge, MA.

[114] Nathaniel Weinman, Armando Fox, and Marti Hearst. 2020. *Exploring Challenging Variations of Parsons Problems.* Association for Computing Machinery, New York, NY, USA, 1349. https://doi.org/10.1145/3328778.3372639

[115] Nathaniel Weinman, Armando Fox, and Marti A Hearst. 2021. Improving Instruction of Programming Patterns with Faded Parsons Problems. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems.* Association for Computing Machinery, New York, NY, USA, 1–4.

[116] Jacqueline L. Whalley and Raymond Lister. 2009. The BRACElet 2009.1 (Wellington) Specification. In *Proceedings of the Eleventh Australasian Conference on Computing Education - Volume 95* (Wellington, New Zealand) *(ACE '09)*. Australian Computer Society, Inc., AUS, 9–18.

[117] Joseph B Wiggins, Joseph F Grafsgaard, Kristy Elizabeth Boyer, Eric N Wiebe, and James C Lester. 2017. Do you think you can? the influence of student self-efficacy on the effectiveness of tutorial dialogue for computer science. *International Journal of Artificial Intelligence in Education* 27, 1 (2017), 130–153.

[118] Zihan Wu, Barbara J. Ericson, and Christopher Brooks. 2023. Using Micro Parsons Problems to Scaffold the Learning of Regular Expressions. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education.* ACM, New York, NY, 457–463.

[119] Mark Zarb, Bedour Alshaigy, Dennis Bouvier, Richard Glassey, Janet Hughes, and Charles Riedesel. 2018. An International Investigation into Student Concerns Regarding Transition into Higher Education Computing. In *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education* (Larnaca, Cyprus) *(ITiCSE 2018 Companion)*. Association for Computing Machinery, New York, NY, USA, 107–129. https://doi.org/10.1145/3293881.3295780

[120] Angela Zavaleta Bernuy and Brian Harrington. 2020. What Are We Asking Our Students? A Literature Map of Student Surveys in Computer Science Education. In *Proceedings of the 2020 Conference on Innovation and Technology in Computer Science Education* (Trondheim, Norway) *(ITiCSE '20)*. Association for Computing Machinery, New York, NY, USA, 418–424. https://doi.org/10.1145/3341525.3387383

[121] Rui Zhi, Min Chi, Tiffany Barnes, and Thomas W Price. 2019. Evaluating the effectiveness of parsons problems for block-based programming. In *Proceedings of the 2019 ACM Conference on International Computing Education Research.* Association for Computing Machinery, New York, NY, USA, 51–59.

[122] Xinming Zhu and Herbert A Simon. 1987. Learning mathematics from examples and by doing. *Cognition and instruction* 4, 3 (1987), 137–166.

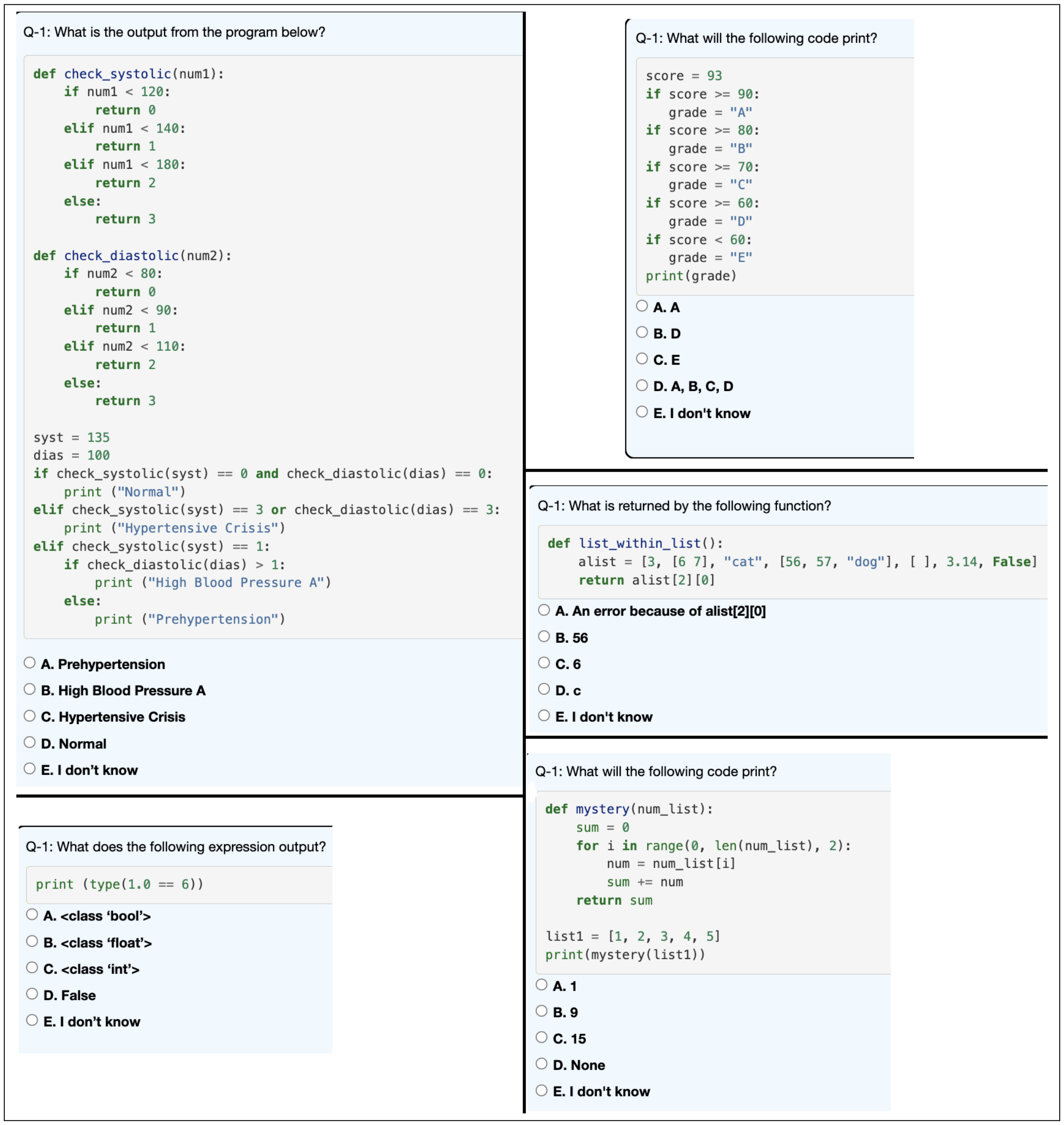
[123] Athanasios Zitouniatis, Fotis Lazarinis, and Dimitris Kanellopoulos. 2022. Teaching Computational Thinking Using Scenario-Based Learning Tools. *Education and Information Technologies* 28, 4 (oct 2022), 4017–4040.

A LEARNING CONTEXTS AND TASKS

This appendix provides additional detail on the tasks associated with each study.

A.1 Pretest

The optional pretest, which can be run as timed or untimed, is designed to assess the student's level of knowledge of the subject matter.



Q-1: What is the output from the program below?

```
def check_systolic(num1):
    if num1 < 120:
        return 0
    elif num1 < 140:
        return 1
    elif num1 < 180:
        return 2
    else:
        return 3

def check_diastolic(num2):
    if num2 < 80:
        return 0
    elif num2 < 90:
        return 1
    elif num2 < 110:
        return 2
    else:
        return 3

syst = 135
dias = 100
if check_systolic(syst) == 0 and check_diastolic(dias) == 0:
    print ("Normal")
elif check_systolic(syst) == 3 or check_diastolic(dias) == 3:
    print ("Hypertensive Crisis")
elif check_systolic(syst) == 1:
    if check_diastolic(dias) > 1:
        print ("High Blood Pressure A")
    else:
        print ("Prehypertension")
```

- A. Prehypertension
- B. High Blood Pressure A
- C. Hypertensive Crisis
- D. Normal
- E. I don't know

Q-1: What will the following code print?

```
score = 93
if score >= 90:
   grade = "A"
if score >= 80:
   grade = "B"
if score >= 70:
   grade = "C"
if score >= 60:
   grade = "D"
if score < 60:
   grade = "E"
print(grade)
```

- A. A
- B. D
- C. E
- D. A, B, C, D
- E. I don't know

Q-1: What is returned by the following function?

```
def list_within_list():
    alist = [3, [6 7], "cat", [56, 57, "dog"], [ ], 3.14, False]
    return alist[2][0]
```

- A. An error because of alist[2][0]
- B. 56
- C. 6
- D. c
- E. I don't know

Q-1: What does the following expression output?

```
print (type(1.0 == 6))
```

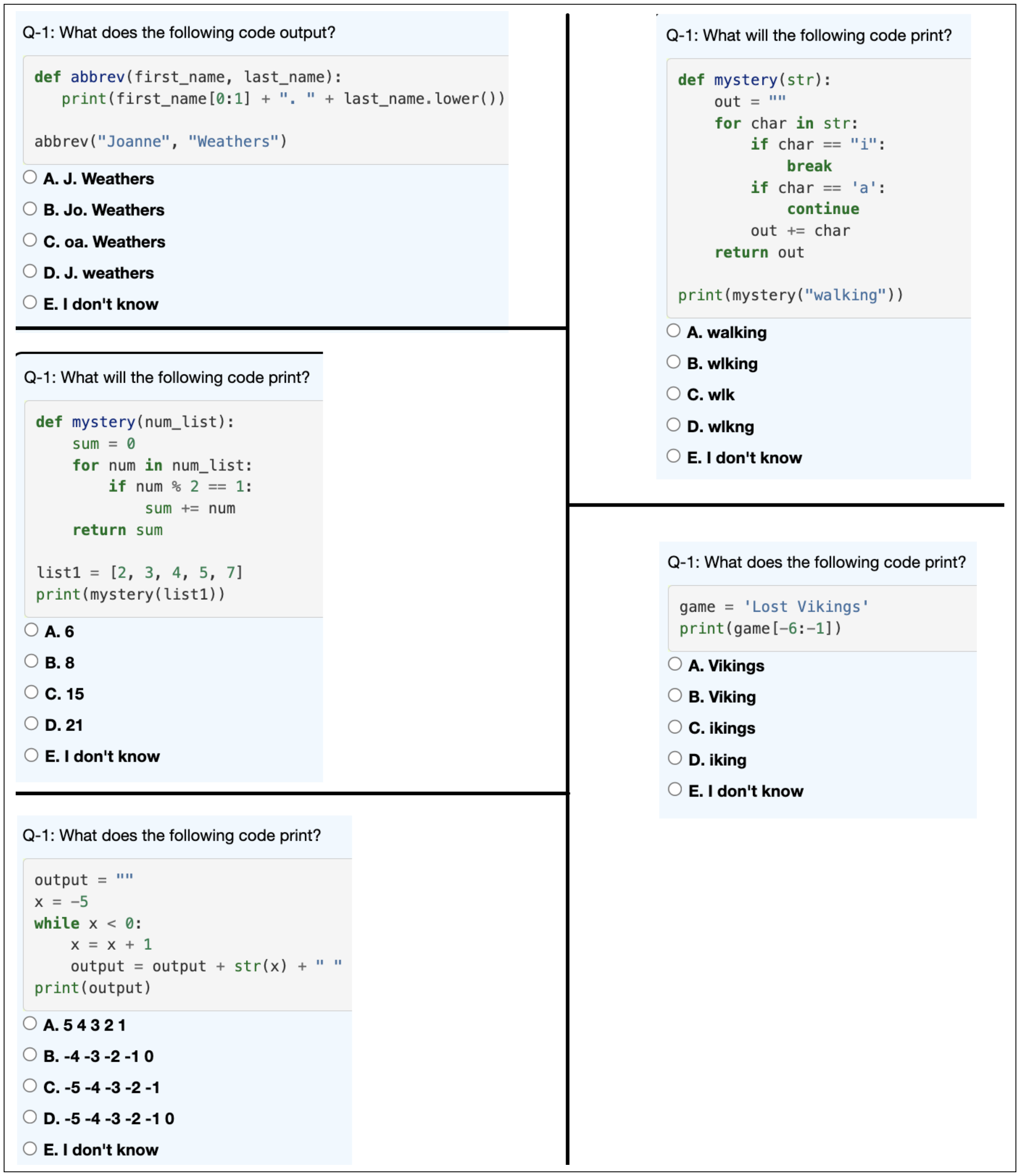
- A. <class 'bool'>
- B. <class 'float'>
- C. <class 'int'>
- D. False
- E. I don't know

Q-1: What will the following code print?

```
def mystery(num_list):
    sum = 0
    for i in range(0, len(num_list), 2):
        num = num_list[i]
        sum += num
    return sum

list1 = [1, 2, 3, 4, 5]
print(mystery(list1))
```

- A. 1
- B. 9
- C. 15
- D. None
- E. I don't know

Figure 26: First 5 questions from 10-question multiple-choice pretest



Q-1: What does the following code output?

```
def abbrev(first_name, last_name):
   print(first_name[0:1] + ". " + last_name.lower())

abbrev("Joanne", "Weathers")
```

- ○ **A. J. Weathers**
- ○ **B. Jo. Weathers**
- ○ **C. oa. Weathers**
- ○ **D. J. weathers**
- ○ **E. I don't know**

Q-1: What will the following code print?

```
def mystery(num_list):
    sum = 0
    for num in num_list:
        if num % 2 == 1:
            sum += num
    return sum

list1 = [2, 3, 4, 5, 7]
print(mystery(list1))
```

- ○ **A. 6**
- ○ **B. 8**
- ○ **C. 15**
- ○ **D. 21**
- ○ **E. I don't know**

Q-1: What does the following code print?

```
output = ""
x = -5
while x < 0:
    x = x + 1
    output = output + str(x) + " "
print(output)
```

- ○ **A. 5 4 3 2 1**
- ○ **B. -4 -3 -2 -1 0**
- ○ **C. -5 -4 -3 -2 -1**
- ○ **D. -5 -4 -3 -2 -1 0**
- ○ **E. I don't know**

Q-1: What will the following code print?

```
def mystery(str):
    out = ""
    for char in str:
        if char == "i":
            break
        if char == 'a':
            continue
        out += char
    return out

print(mystery("walking"))
```

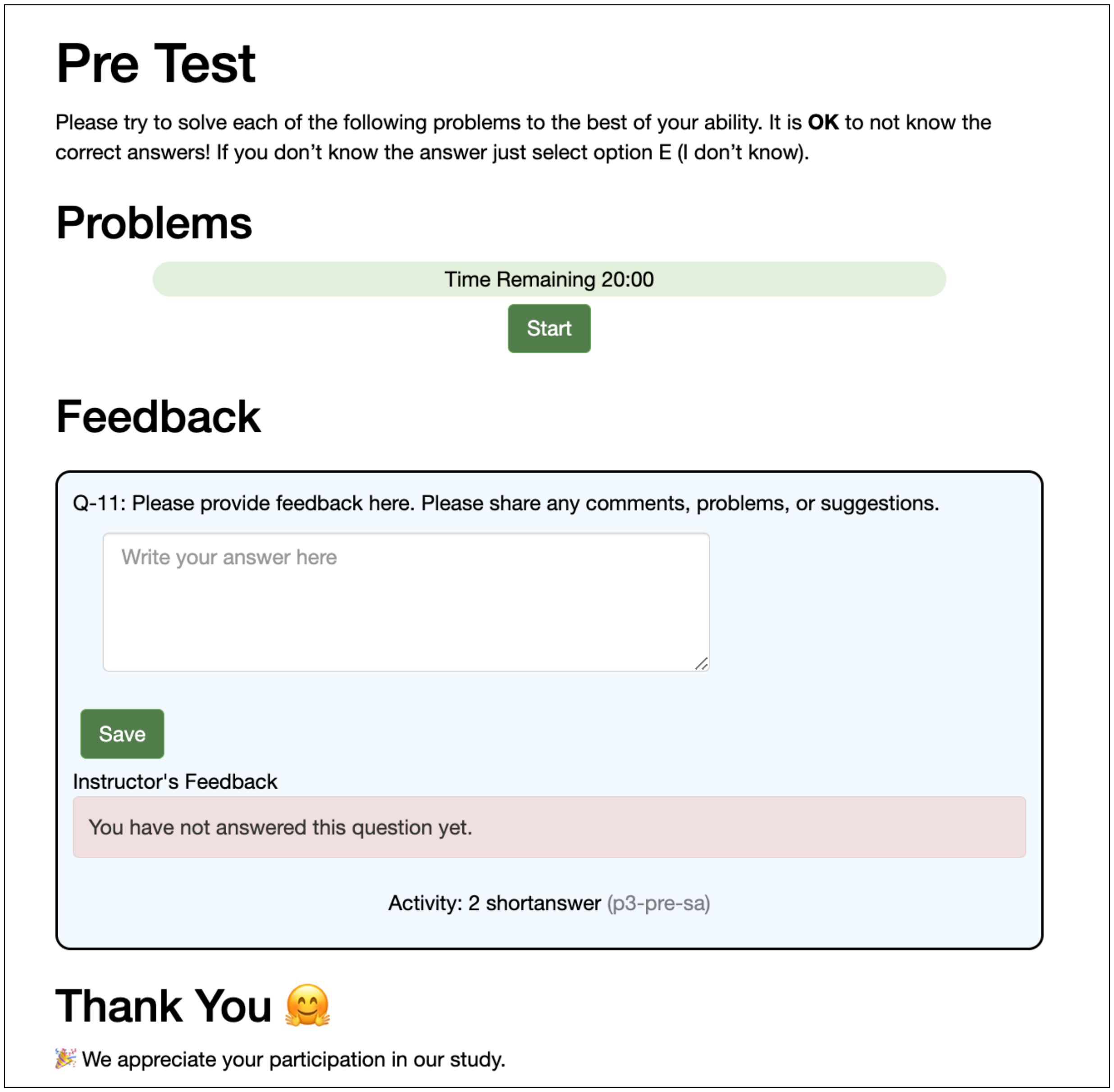
- ○ **A. walking**
- ○ **B. wlking**
- ○ **C. wlk**
- ○ **D. wlkng**
- ○ **E. I don't know**

Q-1: What does the following code print?

```
game = 'Lost Vikings'
print(game[-6:-1])
```

- ○ **A. Vikings**
- ○ **B. Viking**
- ○ **C. ikings**
- ○ **D. iking**
- ○ **E. I don't know**

Figure 27: Last 6 questions from 10-question multiple-choice pretest


Pre Test

Please try to solve each of the following problems to the best of your ability. It is **OK** to not know the correct answers! If you don't know the answer just select option E (I don't know).

Problems

Time Remaining 20:00

Start

Feedback

Q-11: Please provide feedback here. Please share any comments, problems, or suggestions.

Write your answer here

Save

Instructor's Feedback

You have not answered this question yet.

Activity: 2 shortanswer (p3-pre-sa)

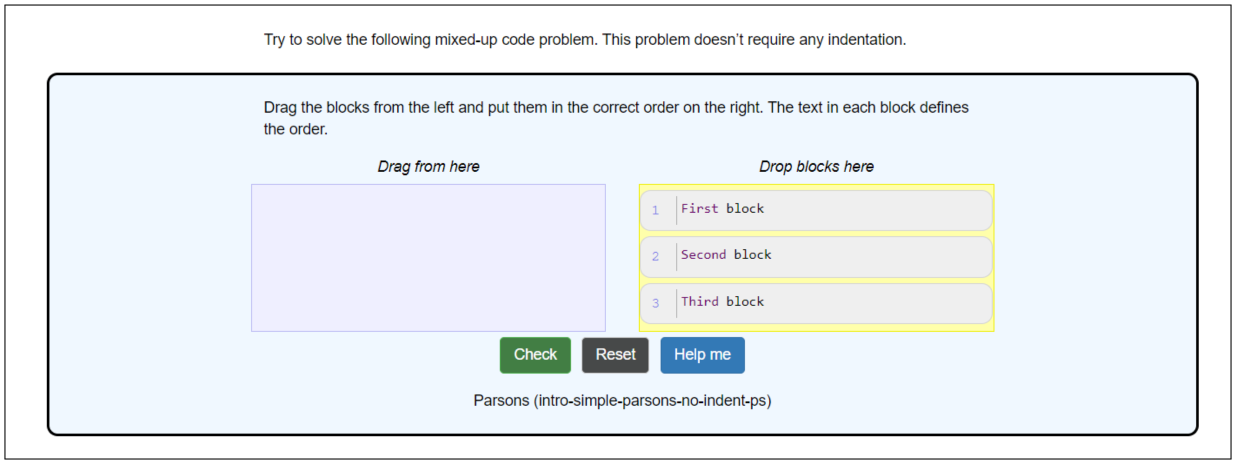
Thank You 🤗

🎉 We appreciate your participation in our study.

Figure 28: Pretest final feedback question

A.2 Introduction to Problem Types

All of the studies included a primer on how to use the interface, which also served to introduce the problem types and help students familiarise themselves with Parson's problems.



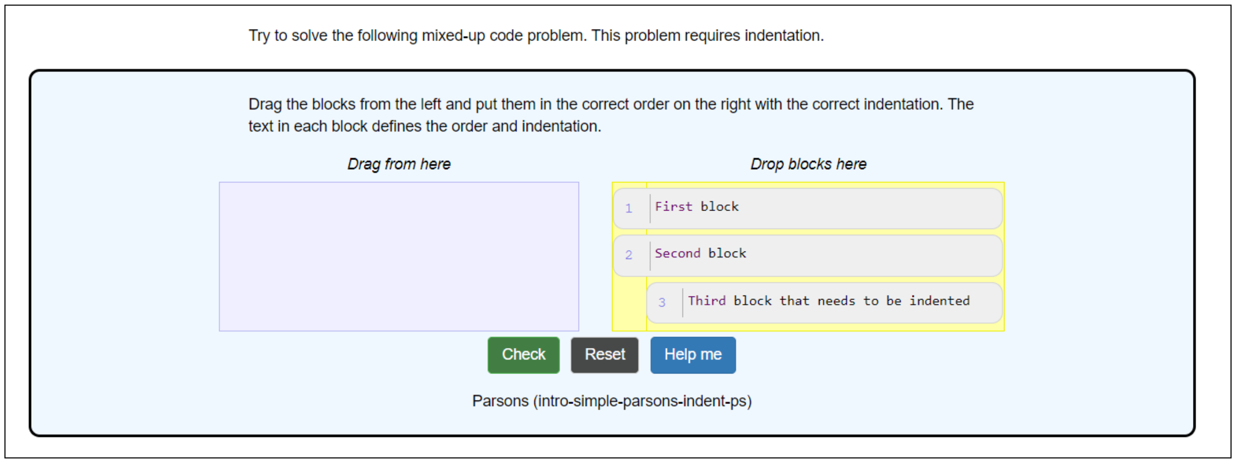
Try to solve the following mixed-up code problem. This problem doesn't require any indentation.

Drag the blocks from the left and put them in the correct order on the right. The text in each block defines the order.

```
First block
Second block
Third block
```

Parsons (intro-simple-parsons-no-indent-ps)

Figure 29: Introduction to problem types - Students familiarise themselves with the drag and drop interface for Parsons problems



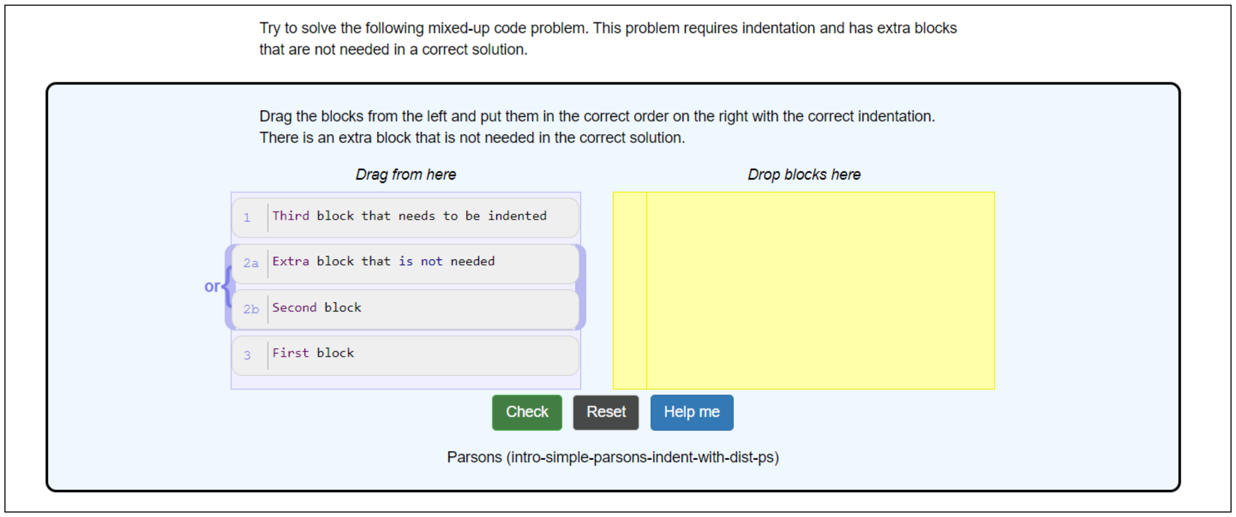
Try to solve the following mixed-up code problem. This problem requires indentation.

Drag the blocks from the left and put them in the correct order on the right with the correct indentation. The text in each block defines the order and indentation.

```
First block
Second block
    Third block that needs to be indented
```

Parsons (intro-simple-parsons-indent-ps)

Figure 30: Introduction to problem types - Students are introduced to indentation in Parsons problems


Try to solve the following mixed-up code problem. This problem requires indentation and has extra blocks that are not needed in a correct solution.

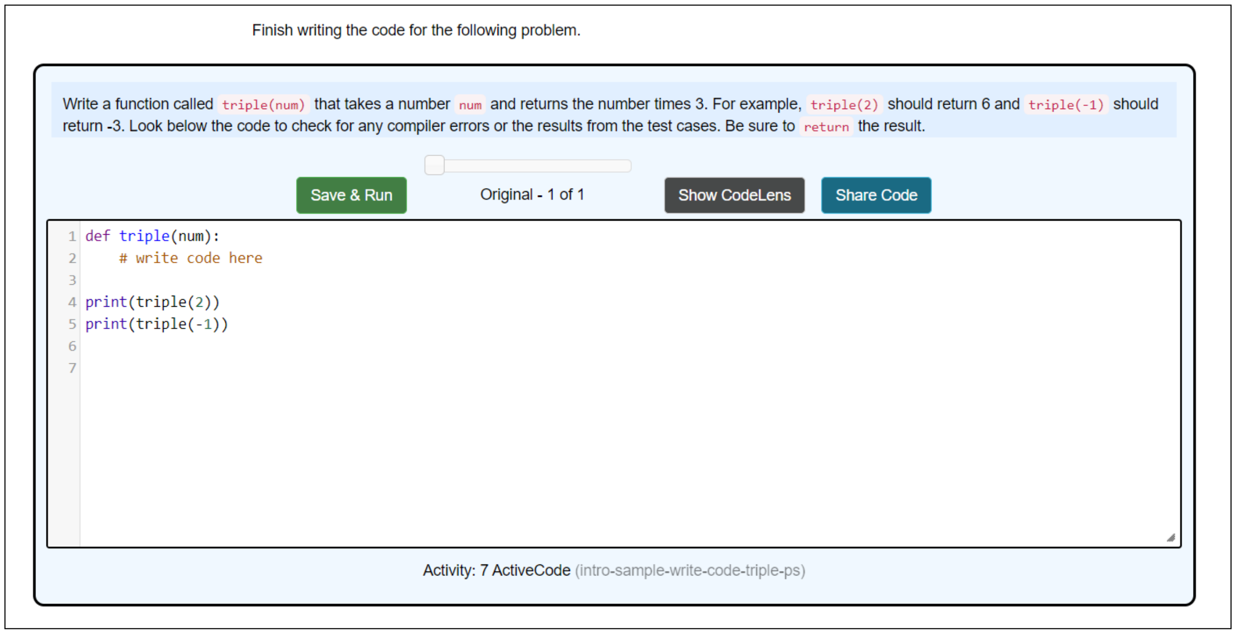
Drag the blocks from the left and put them in the correct order on the right with the correct indentation. There is an extra block that is not needed in the correct solution.

Drag from here

```
1   Third block that needs to be indented
2a  Extra block that is not needed
2b  Second block
3   First block
```

Parsons (intro-simple-parsons-indent-with-dist-ps)

Figure 31: Introduction to problem types - Students practice parsons problems with distractor blocks i.e. blocks that are not needed in a correct solution


Finish writing the code for the following problem.

Write a function called `triple(num)` that takes a number `num` and returns the number times 3. For example, `triple(2)` should return 6 and `triple(-1)` should return -3. Look below the code to check for any compiler errors or the results from the test cases. Be sure to `return` the result.

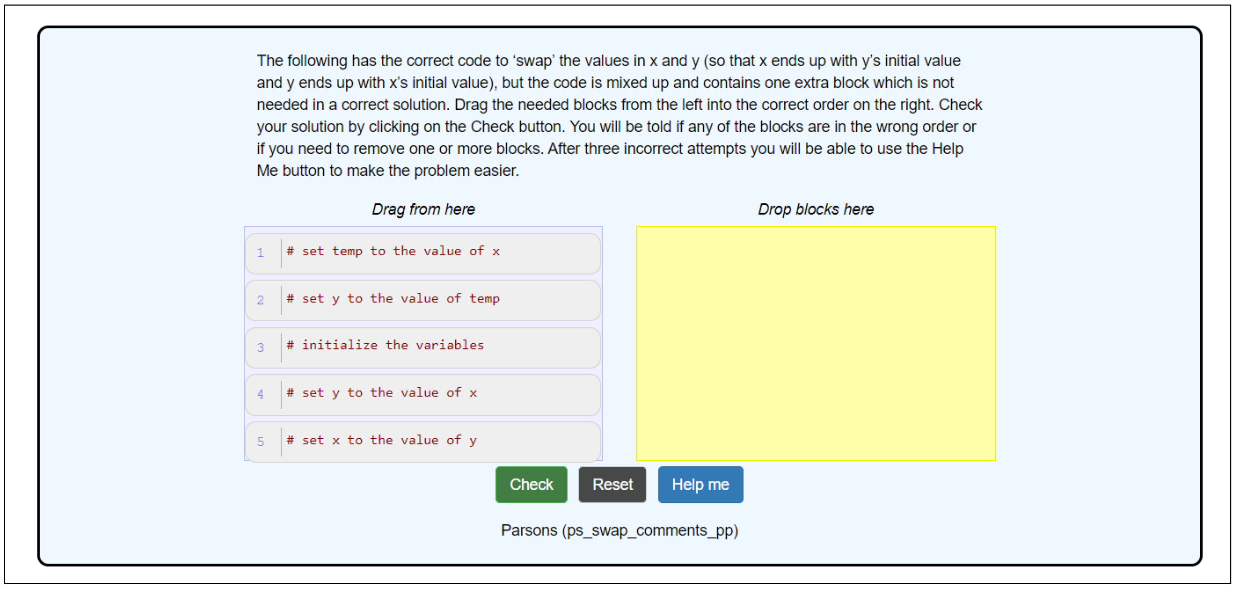
```
def triple(num):
    # write code here

print(triple(2))
print(triple(-1))
```

Activity: 7 ActiveCode (intro-sample-write-code-triple-ps)

Figure 32: Introduction to problem types - Students practice writing code with unit tests

A.3 *python-swap*

Students are tasked with swapping the value of one variable with the value of another variable.



The following has the correct code to 'swap' the values in x and y (so that x ends up with y's initial value and y ends up with x's initial value), but the code is mixed up and contains one extra block which is not needed in a correct solution. Drag the needed blocks from the left into the correct order on the right. Check your solution by clicking on the Check button. You will be told if any of the blocks are in the wrong order or if you need to remove one or more blocks. After three incorrect attempts you will be able to use the Help Me button to make the problem easier.

Drag from here

```
# set temp to the value of x
```

```
# set y to the value of temp
```

```
# initialize the variables
```

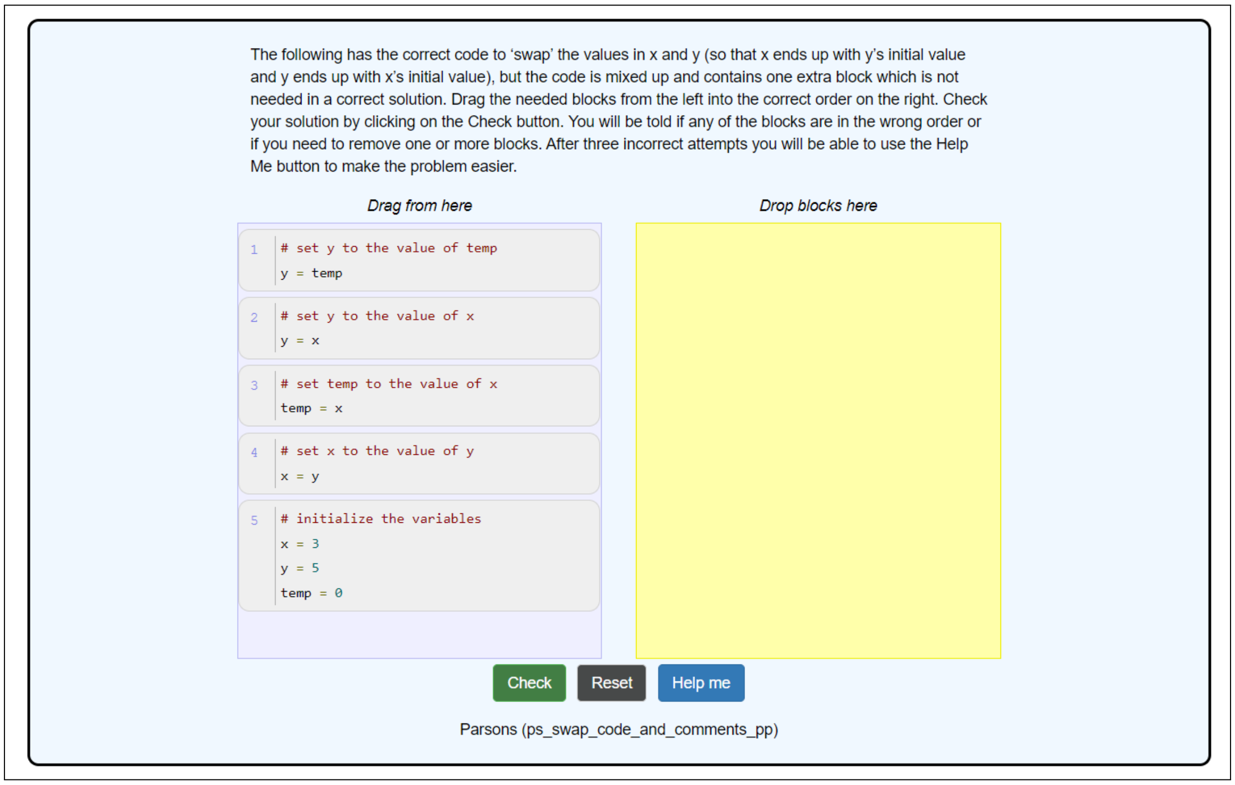
```
# set y to the value of x
```

```
# set x to the value of y
```

Drop blocks here

Parsons (ps_swap_comments_pp)

Figure 33: *python-swap* - Students reorganise comment blocks which contain the logic of the algorithm for swapping the values of two variables


The following has the correct code to 'swap' the values in x and y (so that x ends up with y's initial value and y ends up with x's initial value), but the code is mixed up and contains one extra block which is not needed in a correct solution. Drag the needed blocks from the left into the correct order on the right. Check your solution by clicking on the Check button. You will be told if any of the blocks are in the wrong order or if you need to remove one or more blocks. After three incorrect attempts you will be able to use the Help Me button to make the problem easier.

```
# set y to the value of temp
y = temp
```

```
# set y to the value of x
y = x
```

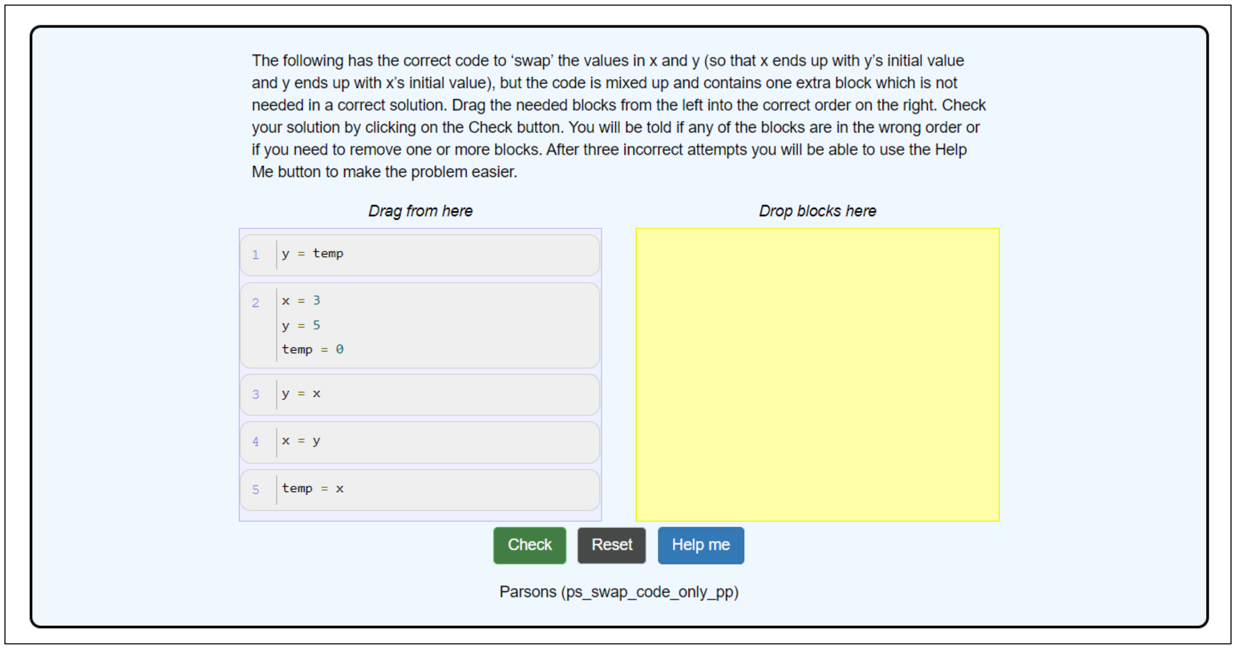
```
# set temp to the value of x
temp = x
```

```
# set x to the value of y
x = y
```

```
# initialize the variables
x = 3
y = 5
temp = 0
```

Parsons (ps_swap_code_and_comments_pp)

Figure 34: *python-swap* - Students practice combining the code with comments blocks to teach the logic of the process


The following has the correct code to 'swap' the values in x and y (so that x ends up with y's initial value and y ends up with x's initial value), but the code is mixed up and contains one extra block which is not needed in a correct solution. Drag the needed blocks from the left into the correct order on the right. Check your solution by clicking on the Check button. You will be told if any of the blocks are in the wrong order or if you need to remove one or more blocks. After three incorrect attempts you will be able to use the Help Me button to make the problem easier.

```
y = temp
```

```
x = 3
y = 5
temp = 0
```

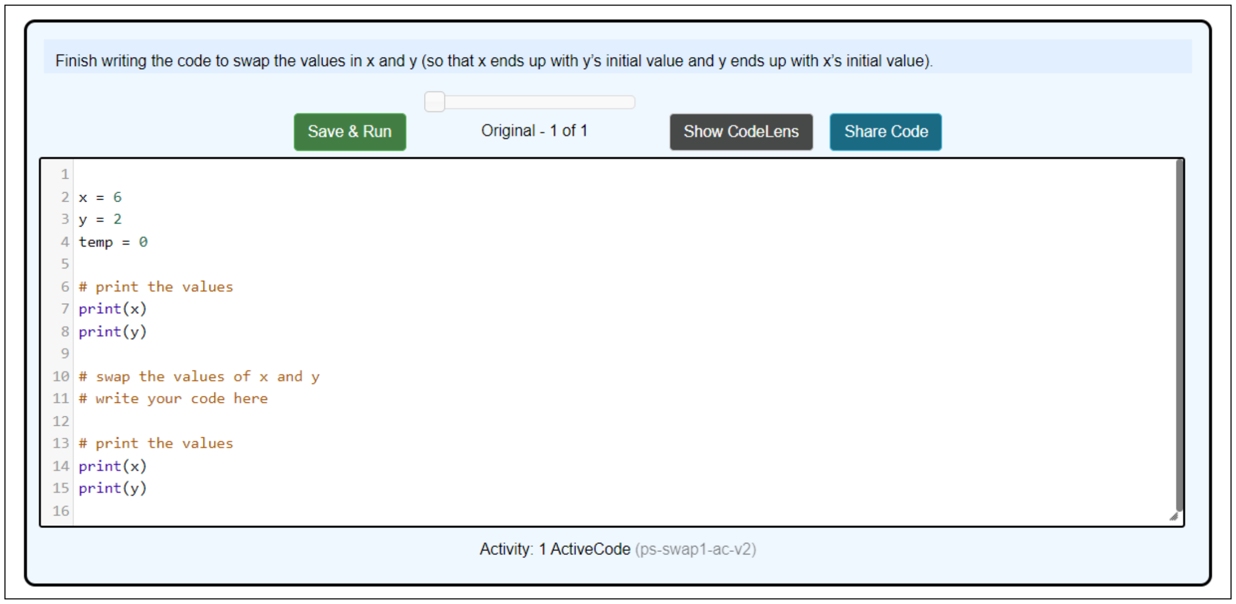
```
y = x
```

```
x = y
```

```
temp = x
```

Parsons (ps_swap_code_only_pp)

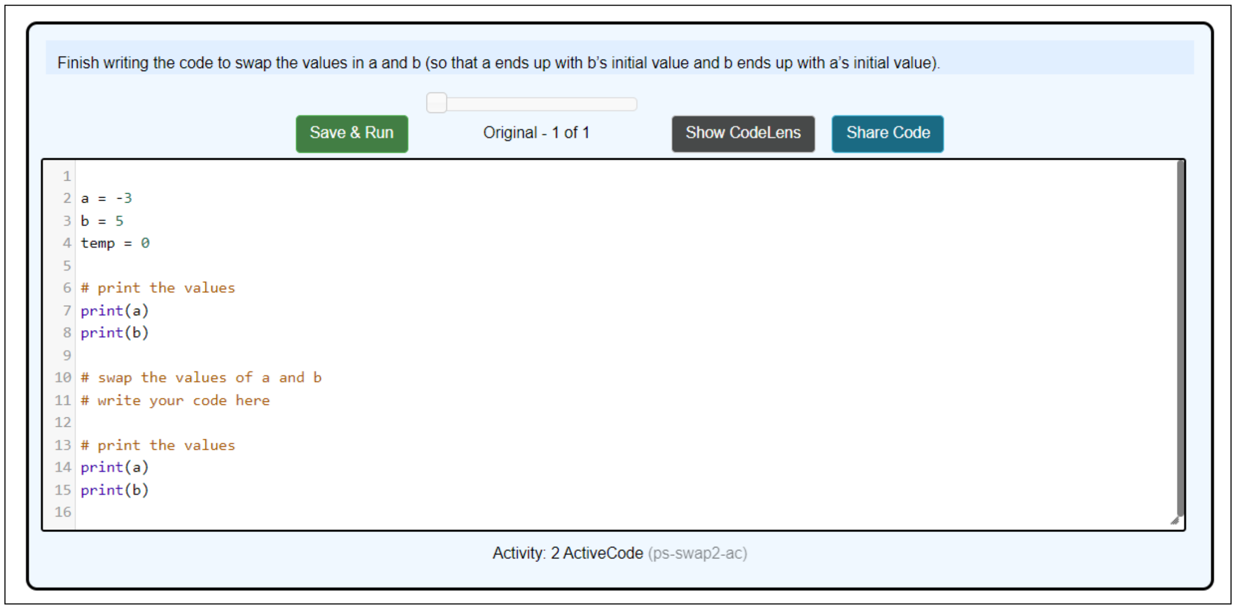
Figure 35: *python-swap* - Students are tasked with a Parsons problem to swap the values of two variables


Finish writing the code to swap the values in x and y (so that x ends up with y's initial value and y ends up with x's initial value).

```python

x = 6
y = 2
temp = 0

# print the values
print(x)
print(y)

# swap the values of x and y
# write your code here

# print the values
print(x)
print(y)

```

Activity: 1 ActiveCode (ps-swap1-ac-v2)

Figure 36: *python-swap* - After completing the Parson's problem exercise, students proceed to write a solution using code


Finish writing the code to swap the values in a and b (so that a ends up with b's initial value and b ends up with a's initial value).

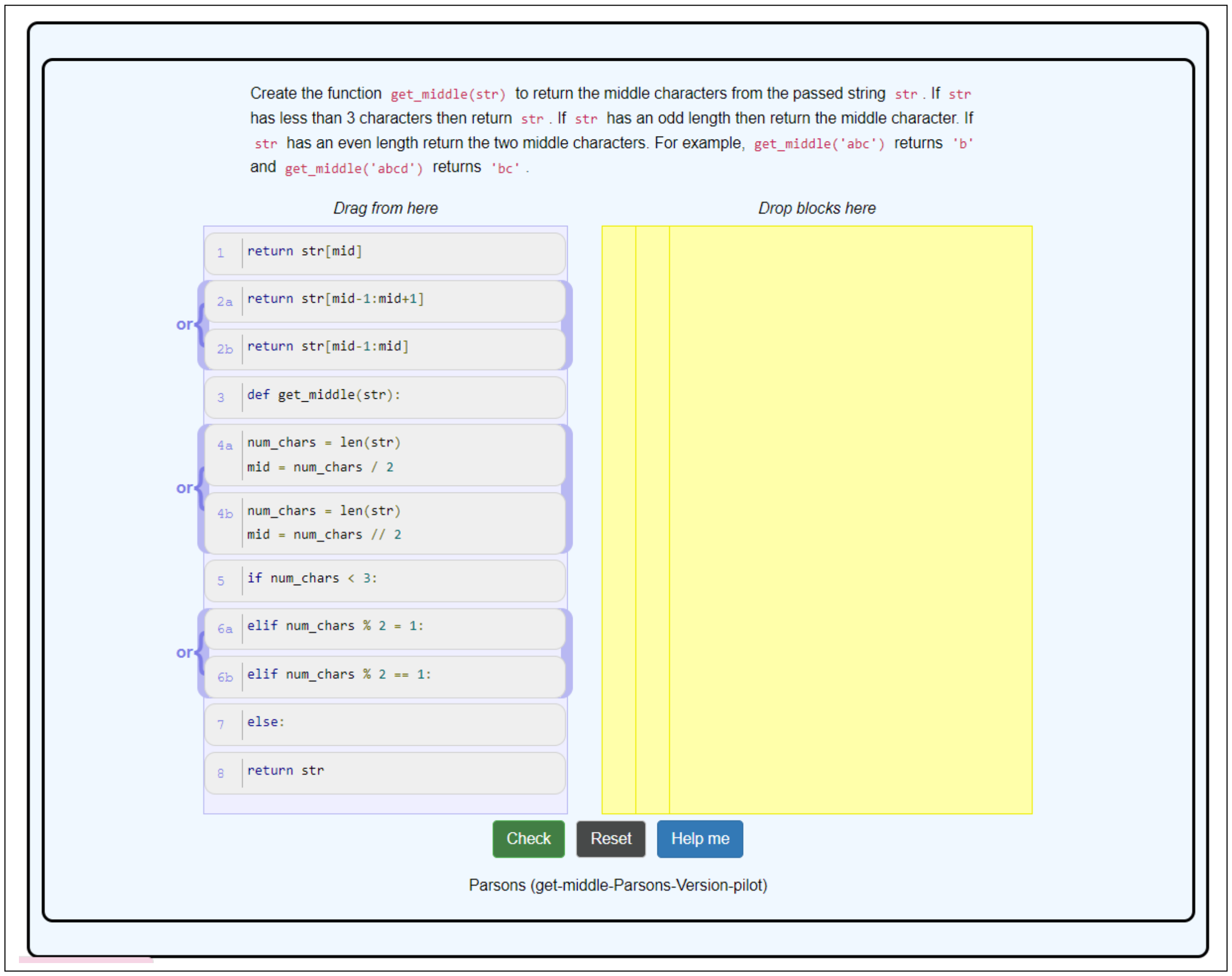
```python

a = -3
b = 5
temp = 0

# print the values
print(a)
print(b)

# swap the values of a and b
# write your code here

# print the values
print(a)
print(b)

```

Activity: 2 ActiveCode (ps-swap2-ac)

Figure 37: *python-swap* - A similar code-writing task with different variable names for near transfer

A.4 *p3pt*

Students are tasked with defining functions in Python that solve defined problems.



Create the function `get_middle(str)` to return the middle characters from the passed string `str`. If `str` has less than 3 characters then return `str`. If `str` has an odd length then return the middle character. If `str` has an even length return the two middle characters. For example, `get_middle('abc')` returns `'b'` and `get_middle('abcd')` returns `'bc'`.

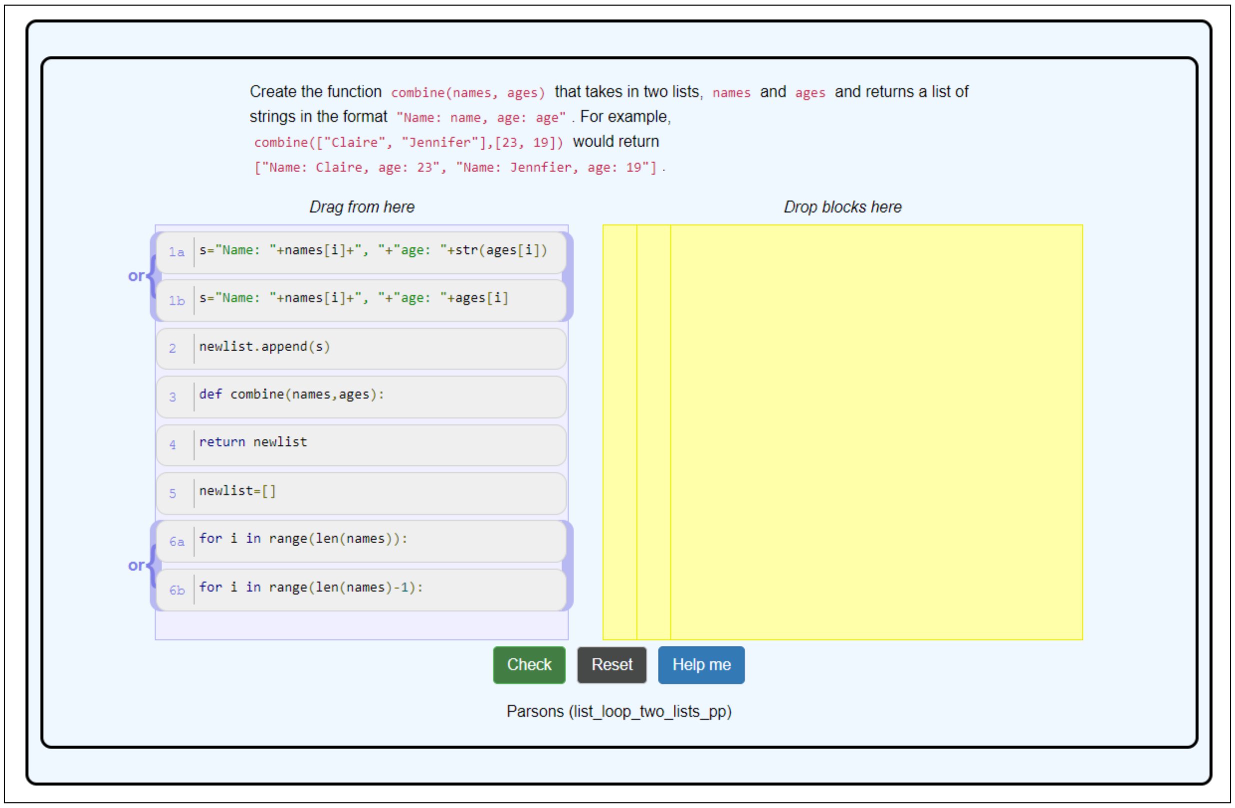
Drag from here

```
1   return str[mid]
2a  return str[mid-1:mid+1]
 or
2b  return str[mid-1:mid]
3   def get_middle(str):
4a  num_chars = len(str)
    mid = num_chars / 2
 or
4b  num_chars = len(str)
    mid = num_chars // 2
5   if num_chars < 3:
6a  elif num_chars % 2 = 1:
 or
6b  elif num_chars % 2 == 1:
7   else:
8   return str
```

Drop blocks here

Parsons (get-middle-Parsons-Version-pilot)

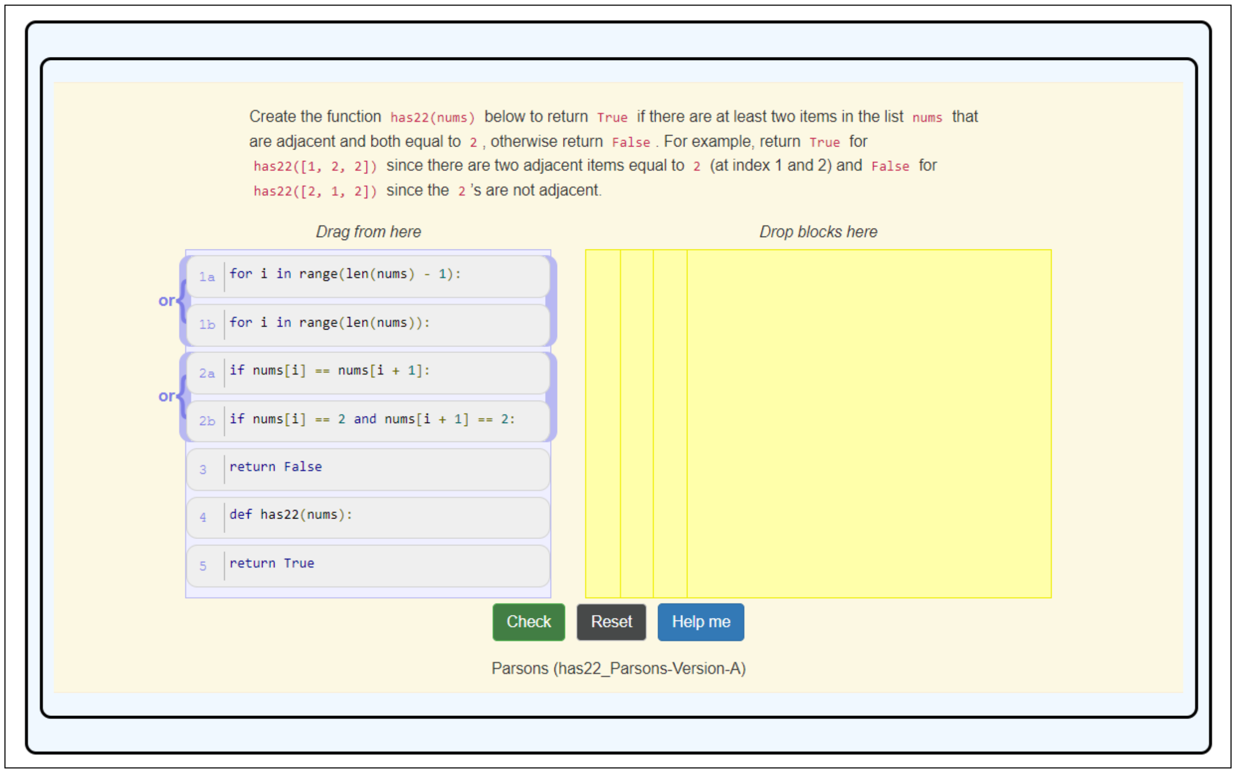
Figure 38: *p3pt* - Defining a function which extracts the middle characters from a string (Parsons)


Create the function `combine(names, ages)` that takes in two lists, `names` and `ages` and returns a list of strings in the format `"Name: name, age: age"`. For example, `combine(["Claire", "Jennifer"],[23, 19])` would return `["Name: Claire, age: 23", "Name: Jennfier, age: 19"]`.

```
1a s="Name: "+names[i]+", "+"age: "+str(ages[i])
or
1b s="Name: "+names[i]+", "+"age: "+ages[i]
2 newlist.append(s)
3 def combine(names,ages):
4 return newlist
5 newlist=[]
6a for i in range(len(names)):
or
6b for i in range(len(names)-1):
```

Parsons (list_loop_two_lists_pp)

Figure 39: *p3pt* - Defining a function that concatenates two lists according to requirements (Parsons)


Create the function `has22(nums)` below to return `True` if there are at least two items in the list `nums` that are adjacent and both equal to `2`, otherwise return `False`. For example, return `True` for `has22([1, 2, 2])` since there are two adjacent items equal to `2` (at index 1 and 2) and `False` for `has22([2, 1, 2])` since the `2`'s are not adjacent.

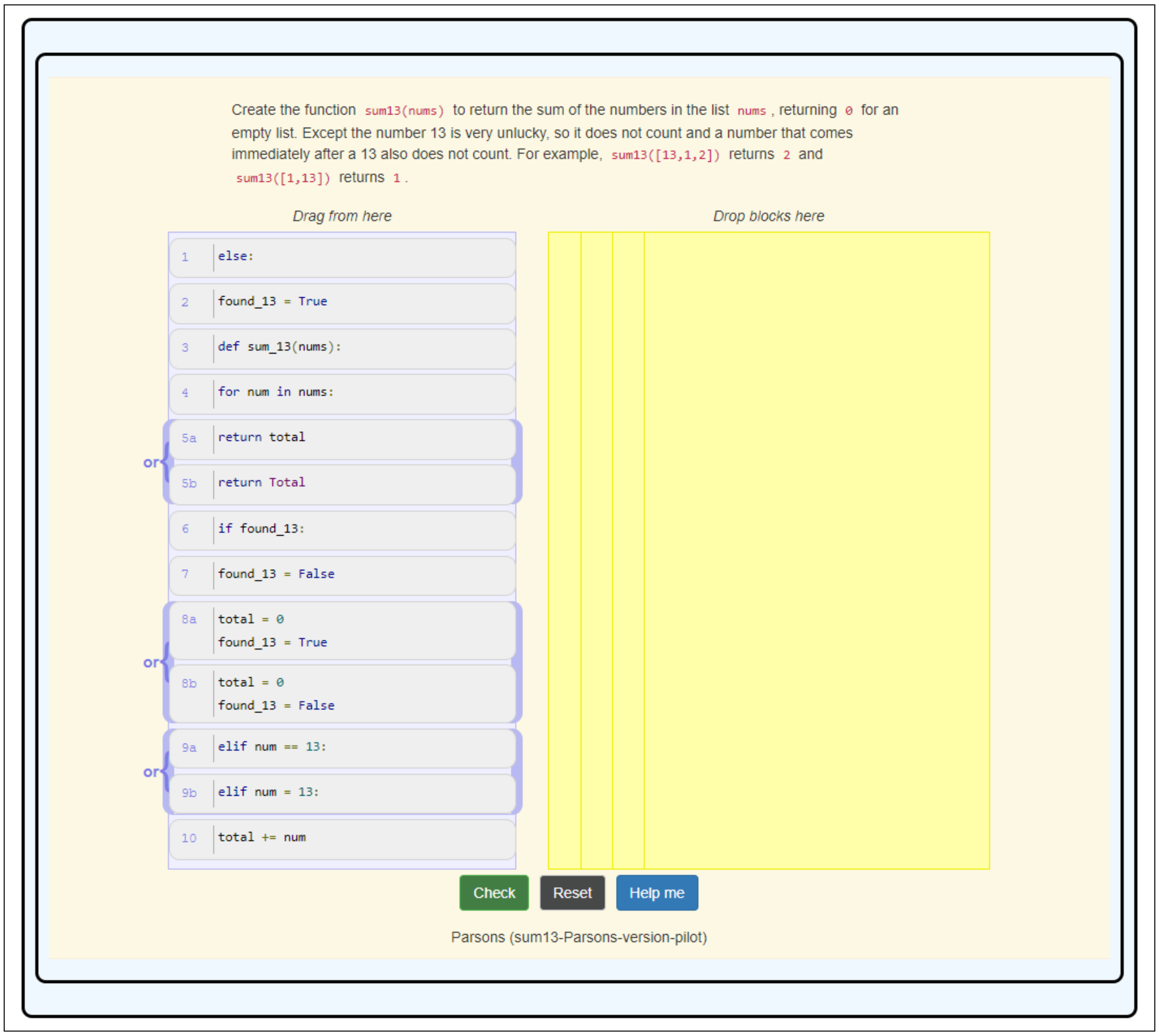
Drag from here

```
1a  for i in range(len(nums) - 1):
or
1b  for i in range(len(nums)):

2a  if nums[i] == nums[i + 1]:
or
2b  if nums[i] == 2 and nums[i + 1] == 2:

3   return False
4   def has22(nums):
5   return True
```

Drop blocks here

Parsons (has22_Parsons-Version-A)

Figure 40: ***p3pt*** **- Defining a function to search a list for adjacent values of two (Parsons)**

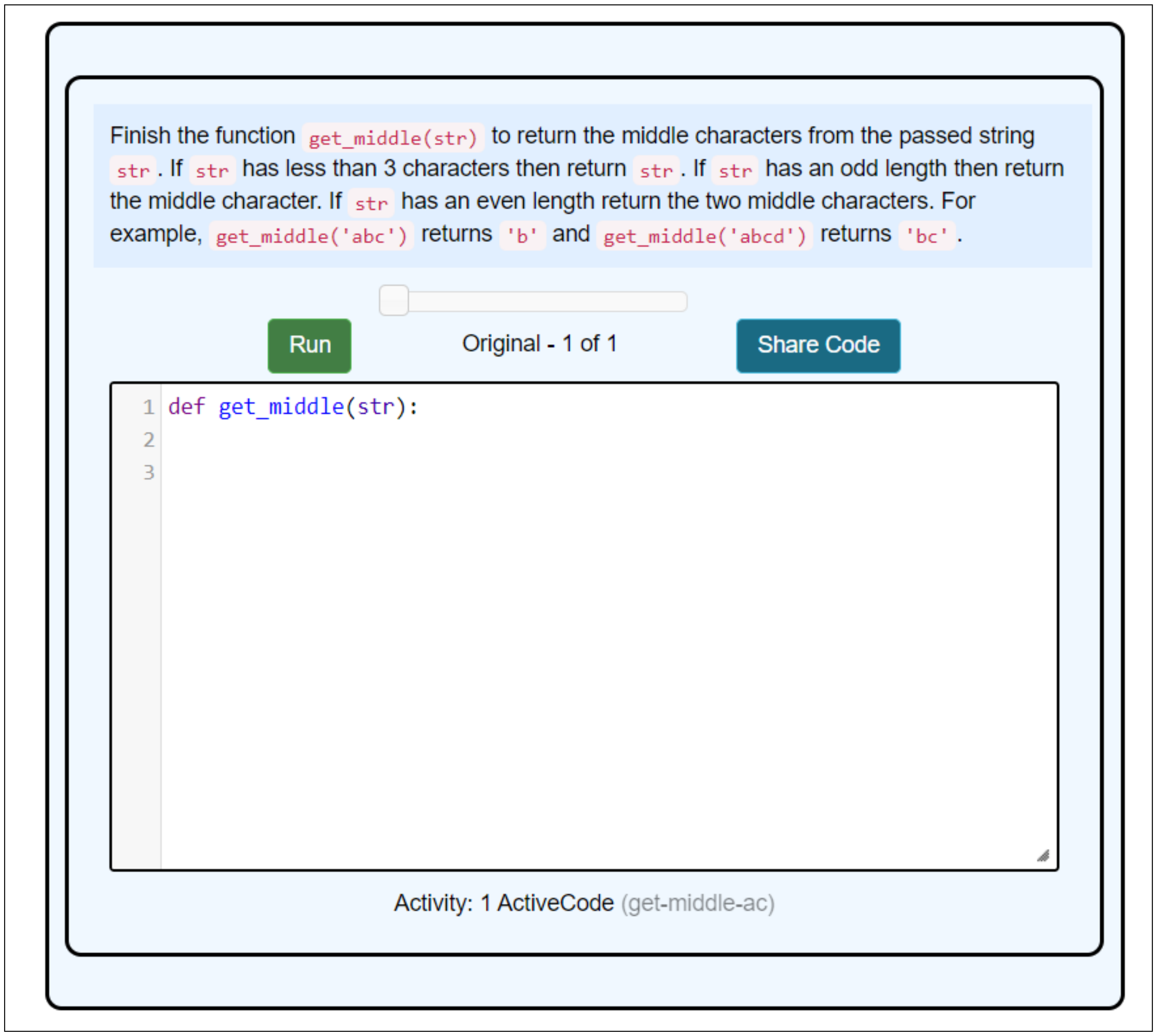

Create the function `sum13(nums)` to return the sum of the numbers in the list `nums`, returning `0` for an empty list. Except the number 13 is very unlucky, so it does not count and a number that comes immediately after a 13 also does not count. For example, `sum13([13,1,2])` returns `2` and `sum13([1,13])` returns `1`.

Drag from here

```
1   else:
2   found_13 = True
3   def sum_13(nums):
4   for num in nums:
5a  return total
5b  return Total
6   if found_13:
7   found_13 = False
8a  total = 0
    found_13 = True
8b  total = 0
    found_13 = False
9a  elif num == 13:
9b  elif num = 13:
10  total += num
```

Parsons (sum13-Parsons-version-pilot)

Figure 41: *p3pt* - Defining a function to sum a list of numbers excluding any element that follows the value of 13 (Parsons)

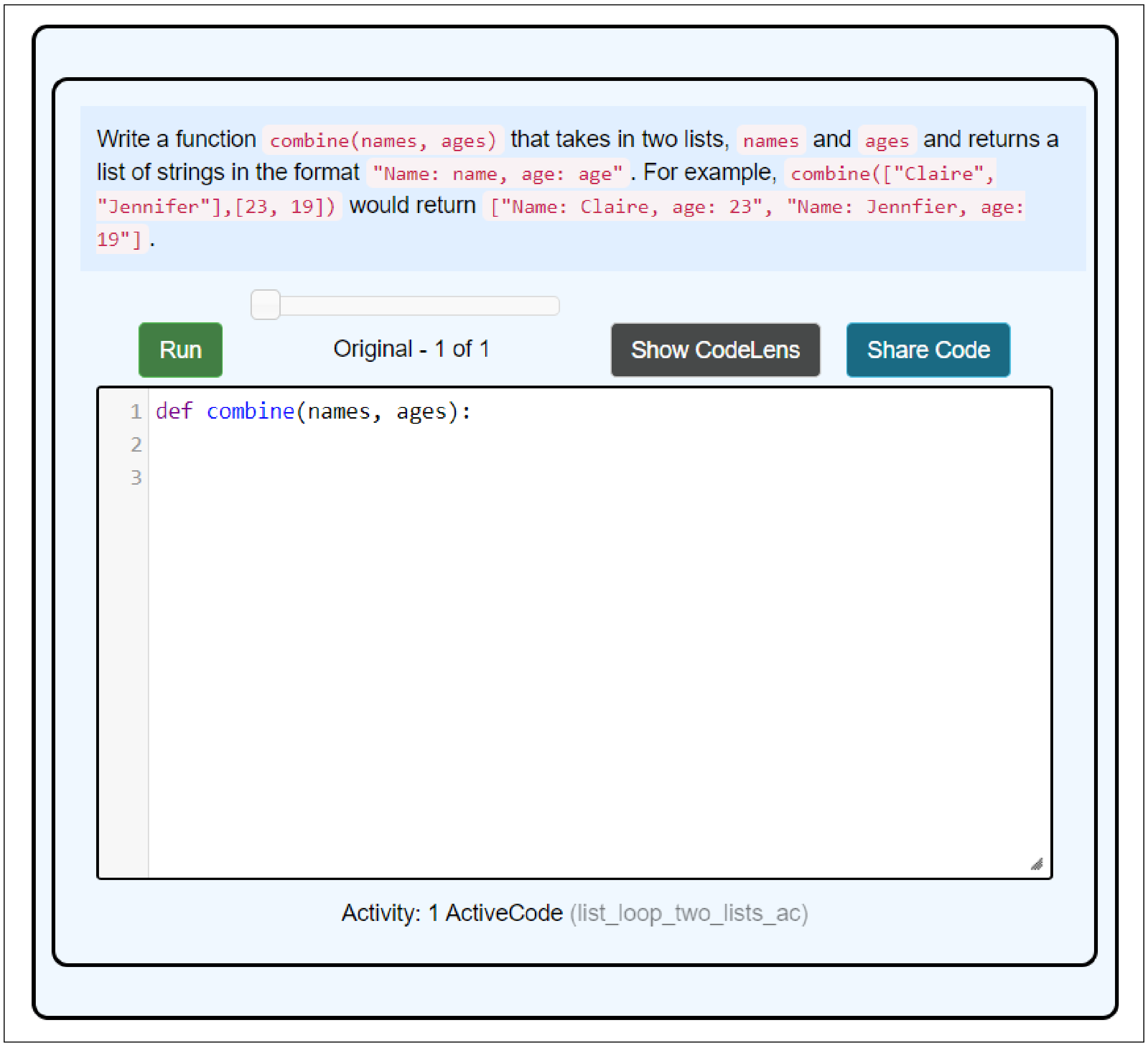


Finish the function `get_middle(str)` to return the middle characters from the passed string `str`. If `str` has less than 3 characters then return `str`. If `str` has an odd length then return the middle character. If `str` has an even length return the two middle characters. For example, `get_middle('abc')` returns `'b'` and `get_middle('abcd')` returns `'bc'`.

```
def get_middle(str):


```

Activity: 1 ActiveCode (get-middle-ac)

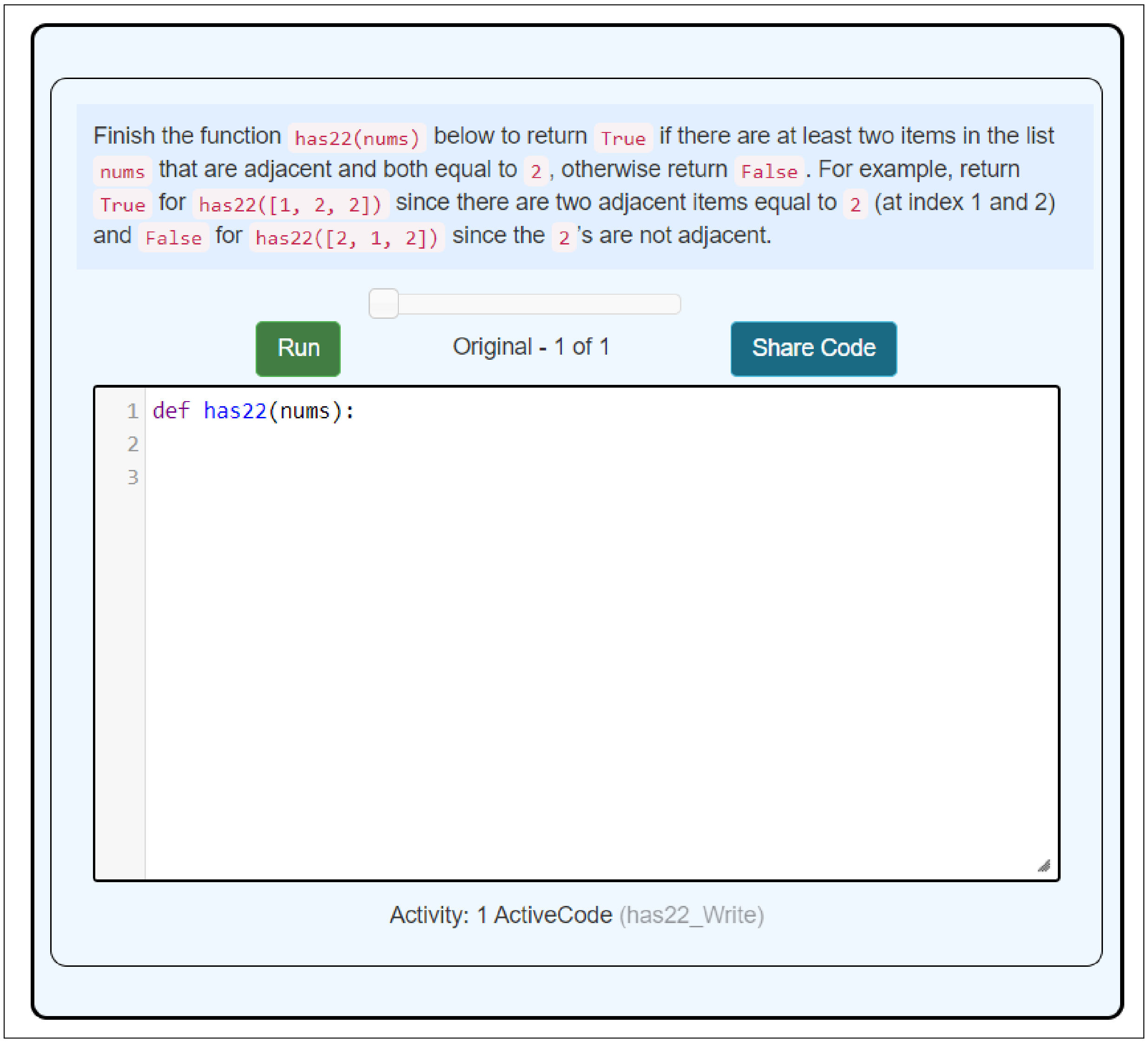
Figure 42: *p3pt* - Defining a function which extracts the middle characters from a string (code writing)


Write a function `combine(names, ages)` that takes in two lists, `names` and `ages` and returns a list of strings in the format `"Name: name, age: age"`. For example, `combine(["Claire", "Jennifer"],[23, 19])` would return `["Name: Claire, age: 23", "Name: Jennfier, age: 19"]`.

```
def combine(names, ages):


```

Activity: 1 ActiveCode (list_loop_two_lists_ac)

Figure 43: *p3pt* - Defining a function that combines two lists according to requirements (code writing)

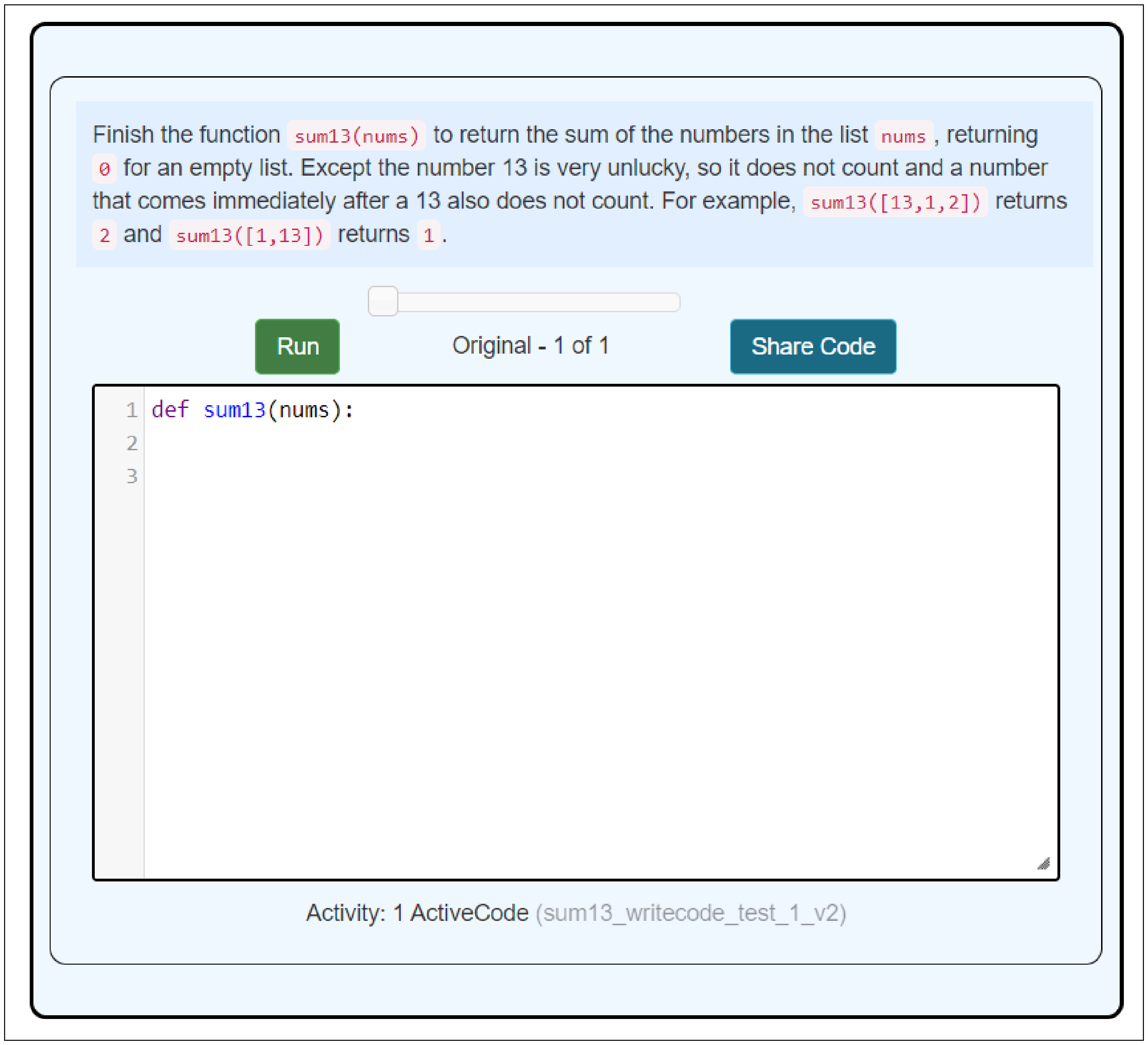


Finish the function `has22(nums)` below to return `True` if there are at least two items in the list `nums` that are adjacent and both equal to `2`, otherwise return `False`. For example, return `True` for `has22([1, 2, 2])` since there are two adjacent items equal to `2` (at index 1 and 2) and `False` for `has22([2, 1, 2])` since the `2`'s are not adjacent.

```
def has22(nums):


```

Activity: 1 ActiveCode (has22_Write)

Figure 44: ***p3pt*** **- Defining a function to search a list for adjacent values of two (code writing)**

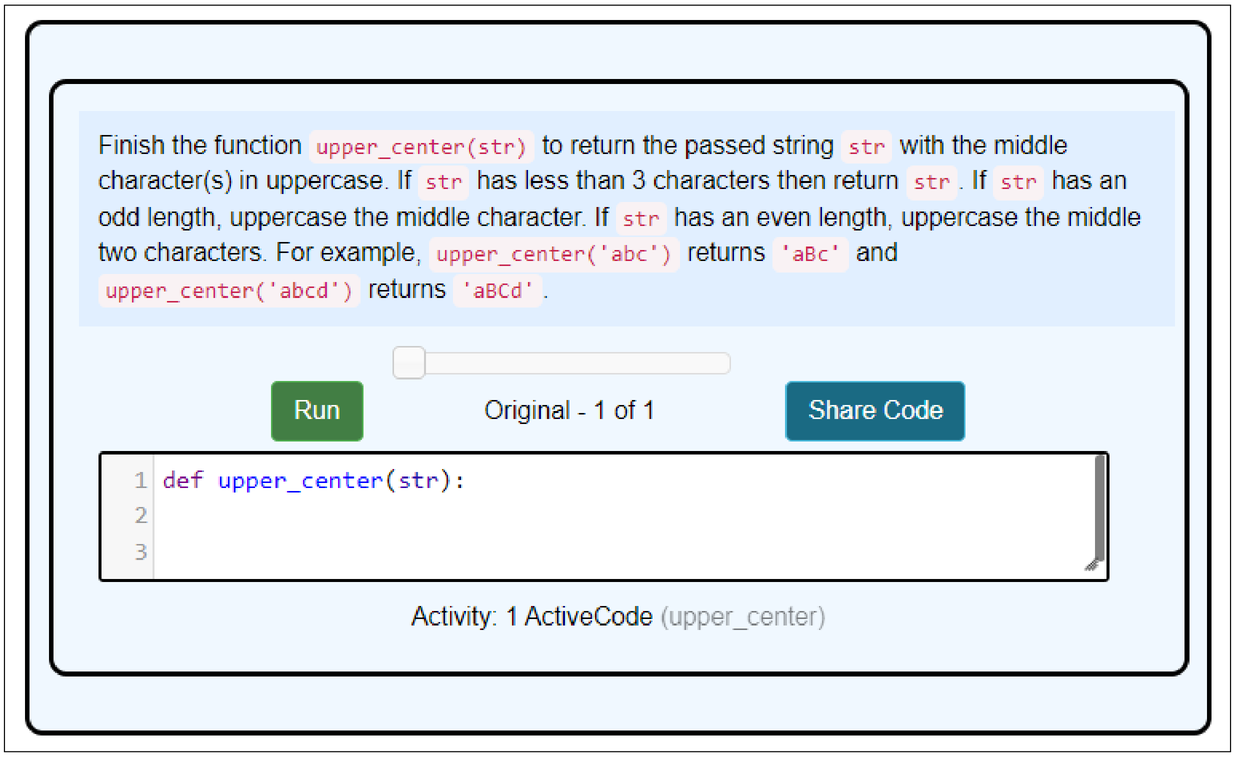


Finish the function `sum13(nums)` to return the sum of the numbers in the list `nums`, returning `0` for an empty list. Except the number 13 is very unlucky, so it does not count and a number that comes immediately after a 13 also does not count. For example, `sum13([13,1,2])` returns `2` and `sum13([1,13])` returns `1`.

```python
def sum13(nums):


```

Activity: 1 ActiveCode (sum13_writecode_test_1_v2)

Figure 45: ***p3pt*** **- Defining a function to sum a list of numbers excluding any element that follows the value of 13 (code writing)**

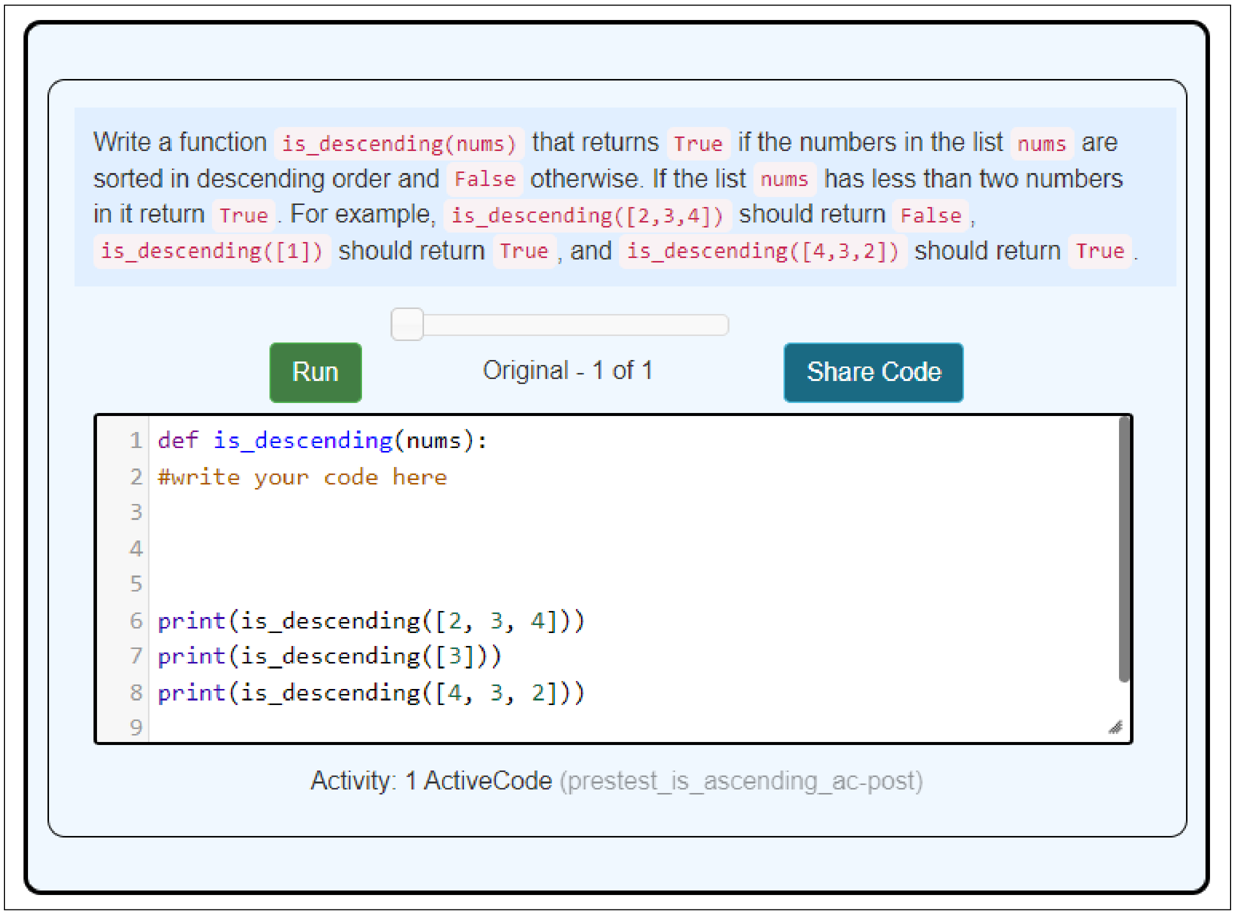


Finish the function `upper_center(str)` to return the passed string `str` with the middle character(s) in uppercase. If `str` has less than 3 characters then return `str`. If `str` has an odd length, uppercase the middle character. If `str` has an even length, uppercase the middle two characters. For example, `upper_center('abc')` returns `'aBc'` and `upper_center('abcd')` returns `'aBCd'`.

```
def upper_center(str):


```

Activity: 1 ActiveCode (upper_center)

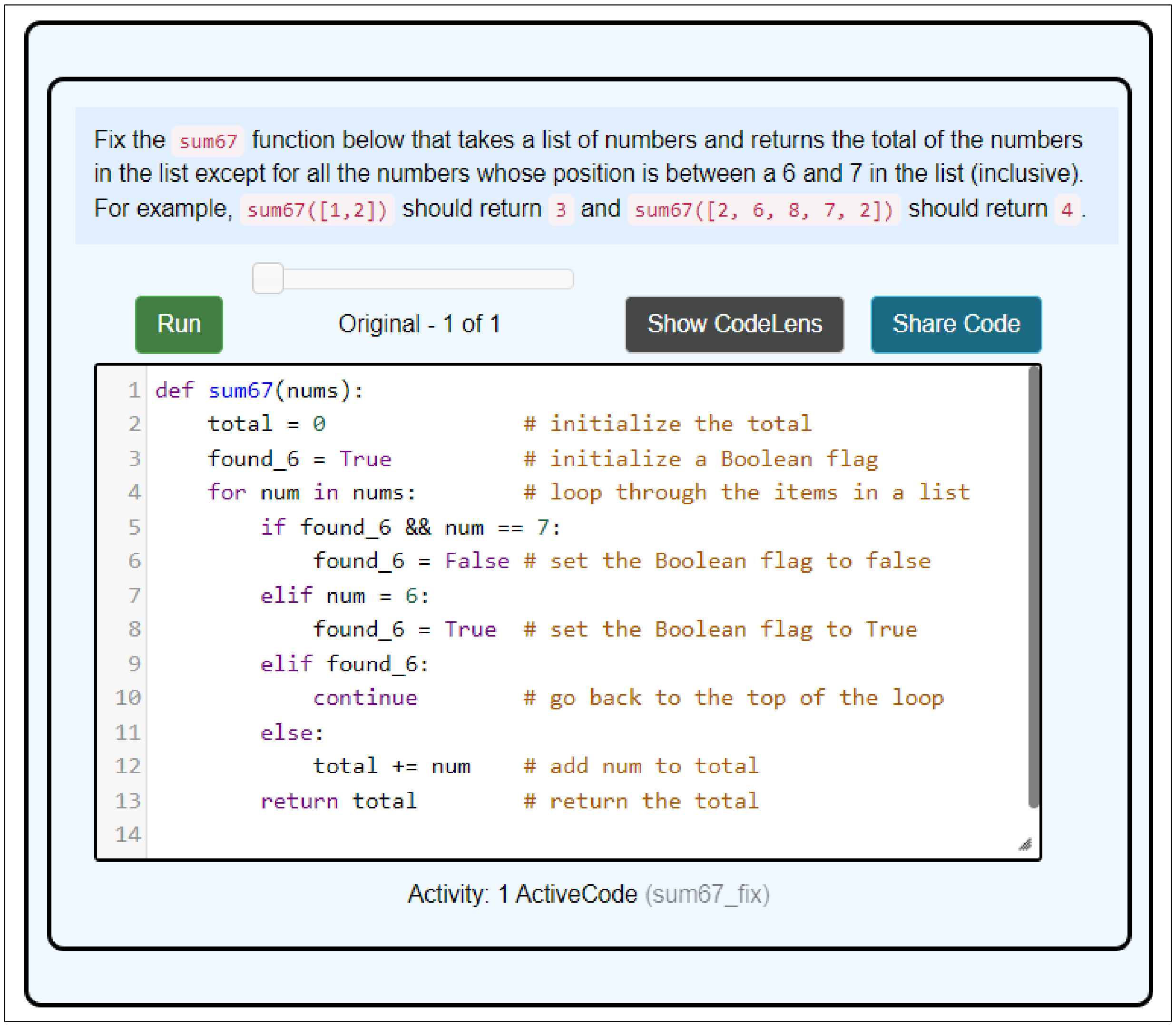
Figure 46: ***p3pt*** **- Posttest based on manipulating and returning a substring**


Write a function `is_descending(nums)` that returns `True` if the numbers in the list `nums` are sorted in descending order and `False` otherwise. If the list `nums` has less than two numbers in it return `True`. For example, `is_descending([2,3,4])` should return `False`, `is_descending([1])` should return `True`, and `is_descending([4,3,2])` should return `True`.

```python
def is_descending(nums):
#write your code here



print(is_descending([2, 3, 4]))
print(is_descending([3]))
print(is_descending([4, 3, 2]))

```

Activity: 1 ActiveCode (prestest_is_ascending_ac-post)

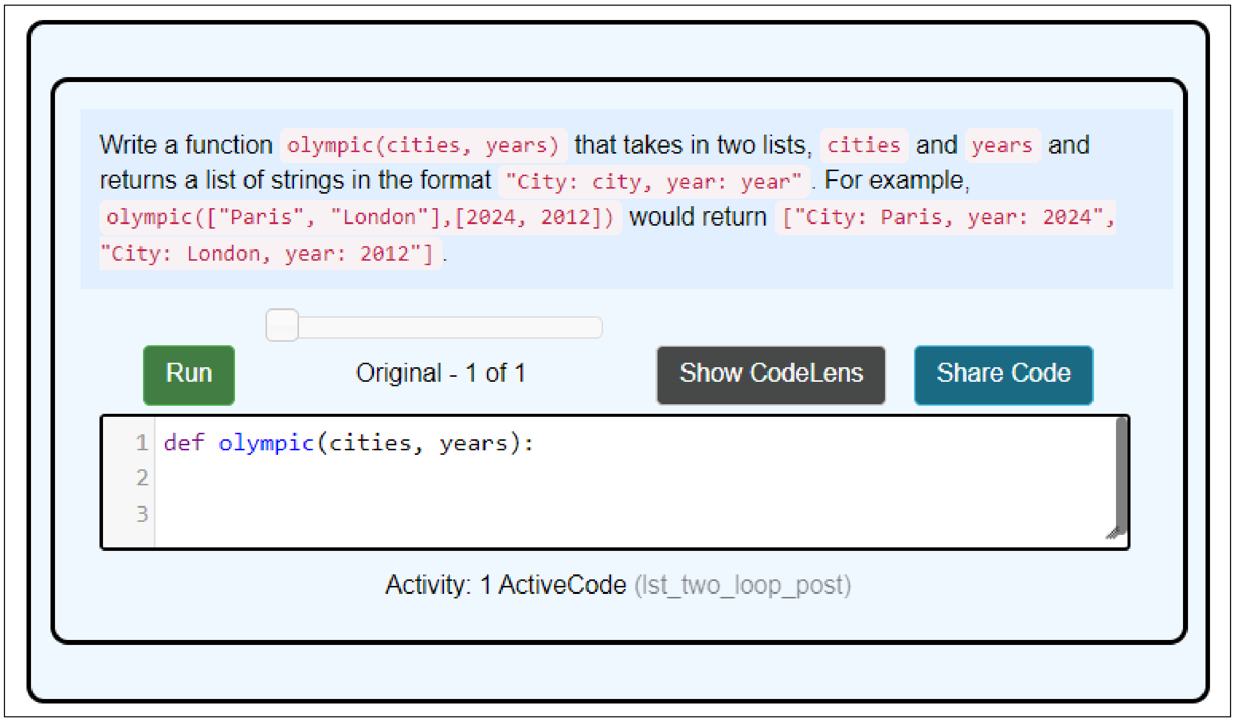
Figure 47: *p3pt* - Posttest based on determining whether a list is in descending order


Fix the `sum67` function below that takes a list of numbers and returns the total of the numbers in the list except for all the numbers whose position is between a 6 and 7 in the list (inclusive). For example, `sum67([1,2])` should return `3` and `sum67([2, 6, 8, 7, 2])` should return `4`.

```python
def sum67(nums):
    total = 0              # initialize the total
    found_6 = True         # initialize a Boolean flag
    for num in nums:       # loop through the items in a list
        if found_6 && num == 7:
            found_6 = False # set the Boolean flag to false
        elif num = 6:
            found_6 = True  # set the Boolean flag to True
        elif found_6:
            continue        # go back to the top of the loop
        else:
            total += num    # add num to total
        return total        # return the total

```

Activity: 1 ActiveCode (sum67_fix)

Figure 48: *p3pt* - Posttest based on conditional summation


Write a function `olympic(cities, years)` that takes in two lists, `cities` and `years` and returns a list of strings in the format `"City: city, year: year"`. For example, `olympic(["Paris", "London"],[2024, 2012])` would return `["City: Paris, year: 2024", "City: London, year: 2012"]`.

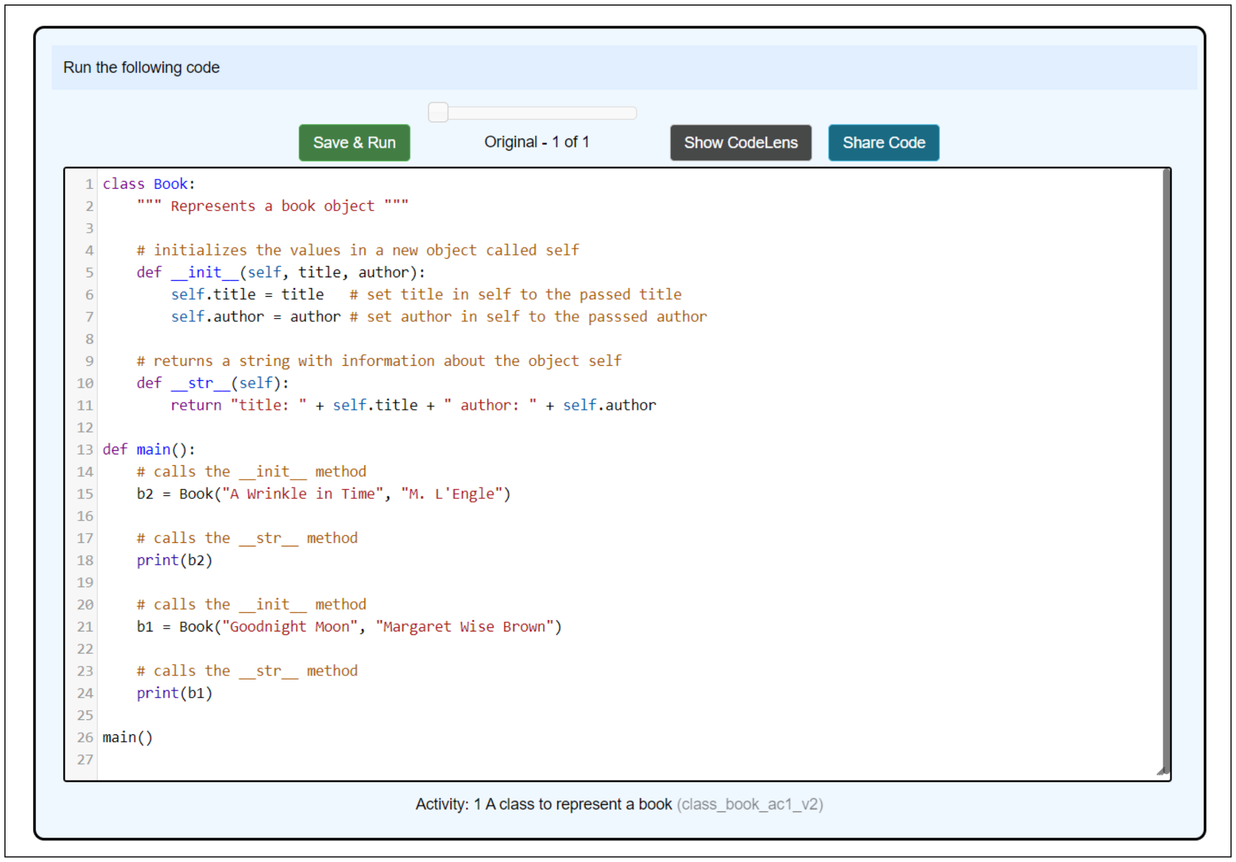
```
def olympic(cities, years):


```

Activity: 1 ActiveCode (lst_two_loop_post)

Figure 49: *p3pt* - Posttest based on combining two lists

A.5 Introduction to Classes

The *classexp* and *classtog* studies involved object-orientated concepts and programming constructs. Brief instruction on how to define classes are provided as part of these respective studies.

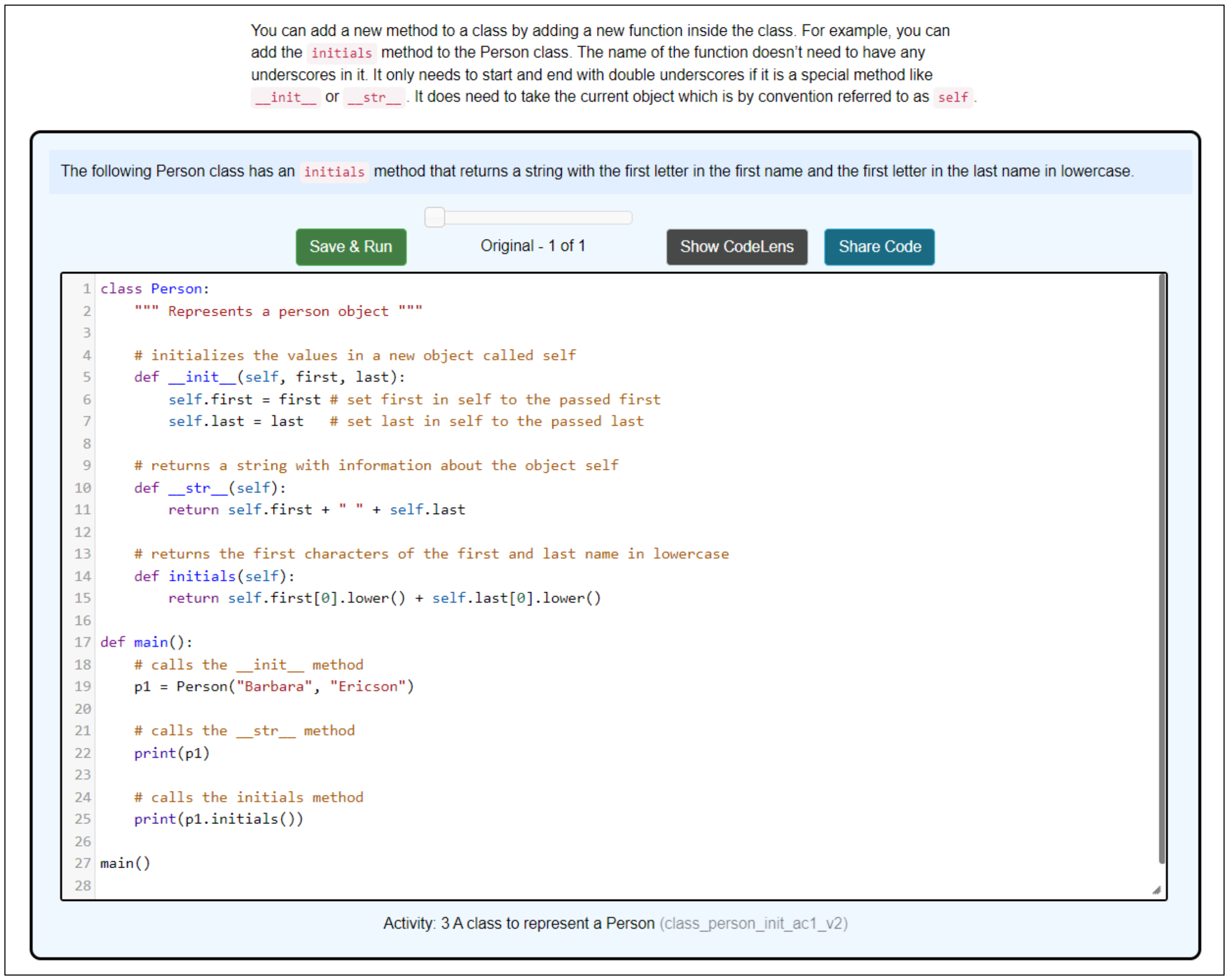


Run the following code

```python
class Book:
    """ Represents a book object """

    # initializes the values in a new object called self
    def __init__(self, title, author):
        self.title = title   # set title in self to the passed title
        self.author = author # set author in self to the passsed author

    # returns a string with information about the object self
    def __str__(self):
        return "title: " + self.title + " author: " + self.author

def main():
    # calls the __init__ method
    b2 = Book("A Wrinkle in Time", "M. L'Engle")

    # calls the __str__ method
    print(b2)

    # calls the __init__ method
    b1 = Book("Goodnight Moon", "Margaret Wise Brown")

    # calls the __str__ method
    print(b1)

main()

```

Activity: 1 A class to represent a book (class_book_ac1_v2)

Figure 50: Introduction to classes - Instructions on how to create a class


You can add a new method to a class by adding a new function inside the class. For example, you can add the `initials` method to the Person class. The name of the function doesn't need to have any underscores in it. It only needs to start and end with double underscores if it is a special method like `__init__` or `__str__`. It does need to take the current object which is by convention referred to as `self`.

The following Person class has an `initials` method that returns a string with the first letter in the first name and the first letter in the last name in lowercase.

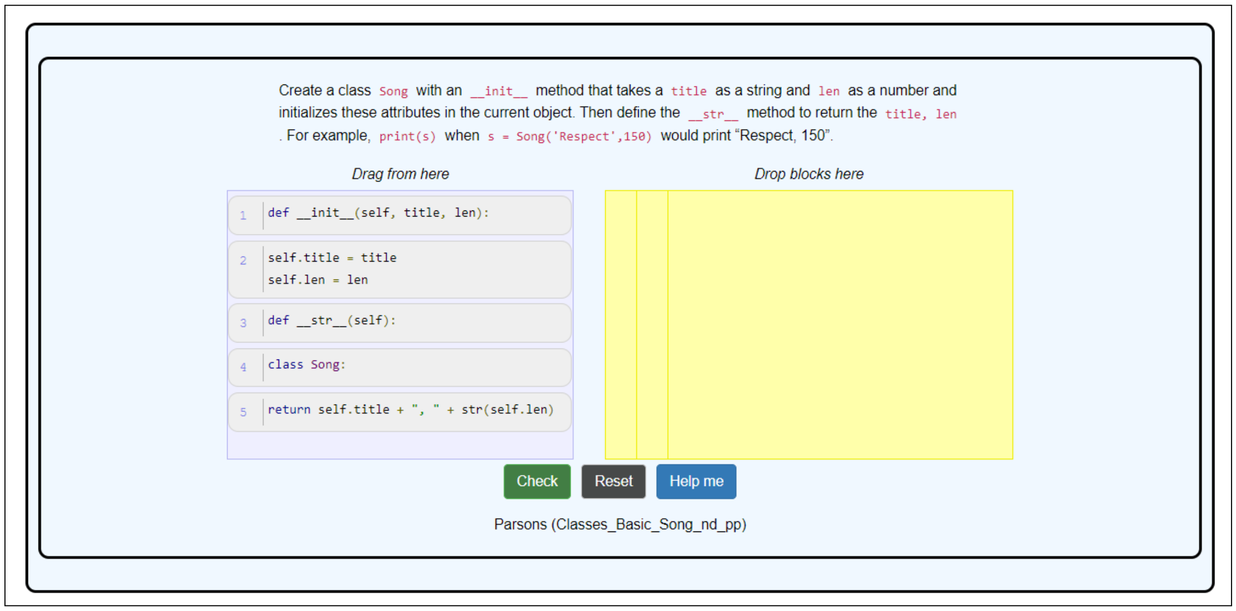
```python
class Person:
    """ Represents a person object """

    # initializes the values in a new object called self
    def __init__(self, first, last):
        self.first = first # set first in self to the passed first
        self.last = last   # set last in self to the passed last

    # returns a string with information about the object self
    def __str__(self):
        return self.first + " " + self.last

    # returns the first characters of the first and last name in lowercase
    def initials(self):
        return self.first[0].lower() + self.last[0].lower()

def main():
    # calls the __init__ method
    p1 = Person("Barbara", "Ericson")

    # calls the __str__ method
    print(p1)

    # calls the initials method
    print(p1.initials())

main()

```

Activity: 3 A class to represent a Person (class_person_init_ac1_v2)

Figure 51: Introduction to classes - worked example

A.6 *class-exp*

Students are tasked with creating classes. One arrangement of the learning materials uses distractors while the other does not.



Create a class `Song` with an `__init__` method that takes a `title` as a string and `len` as a number and initializes these attributes in the current object. Then define the `__str__` method to return the `title, len`. For example, `print(s)` when `s = Song('Respect',150)` would print "Respect, 150".

```
def __init__(self, title, len):
```

```
self.title = title
self.len = len
```

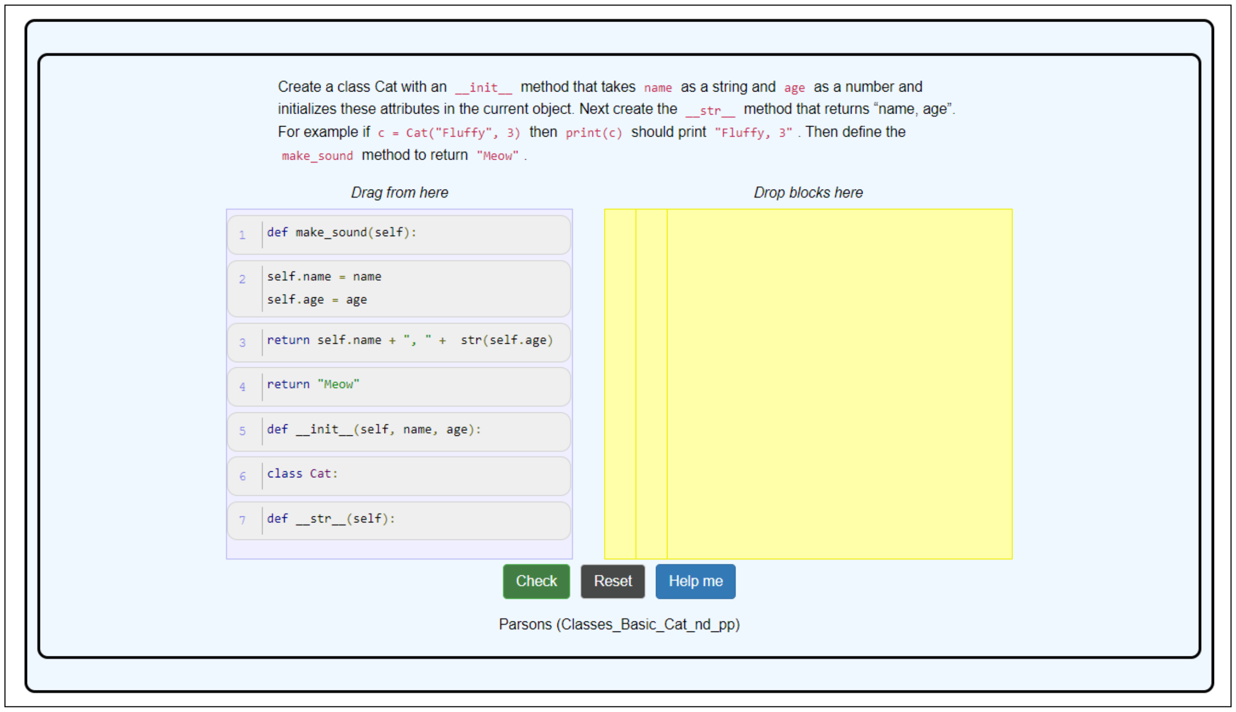
```
def __str__(self):
```

```
class Song:
```

```
return self.title + ", " + str(self.len)
```

Parsons (Classes_Basic_Song_nd_pp)

Figure 52: *class-exp* - Define a song class with name and duration attributes



Create a class Cat with an `__init__` method that takes `name` as a string and `age` as a number and initializes these attributes in the current object. Next create the `__str__` method that returns "name, age". For example if `c = Cat("Fluffy", 3)` then `print(c)` should print `"Fluffy, 3"`. Then define the `make_sound` method to return `"Meow"`.

```
def make_sound(self):
```

```
self.name = name
self.age = age
```

```
return self.name + ", " +  str(self.age)
```

```
return "Meow"
```

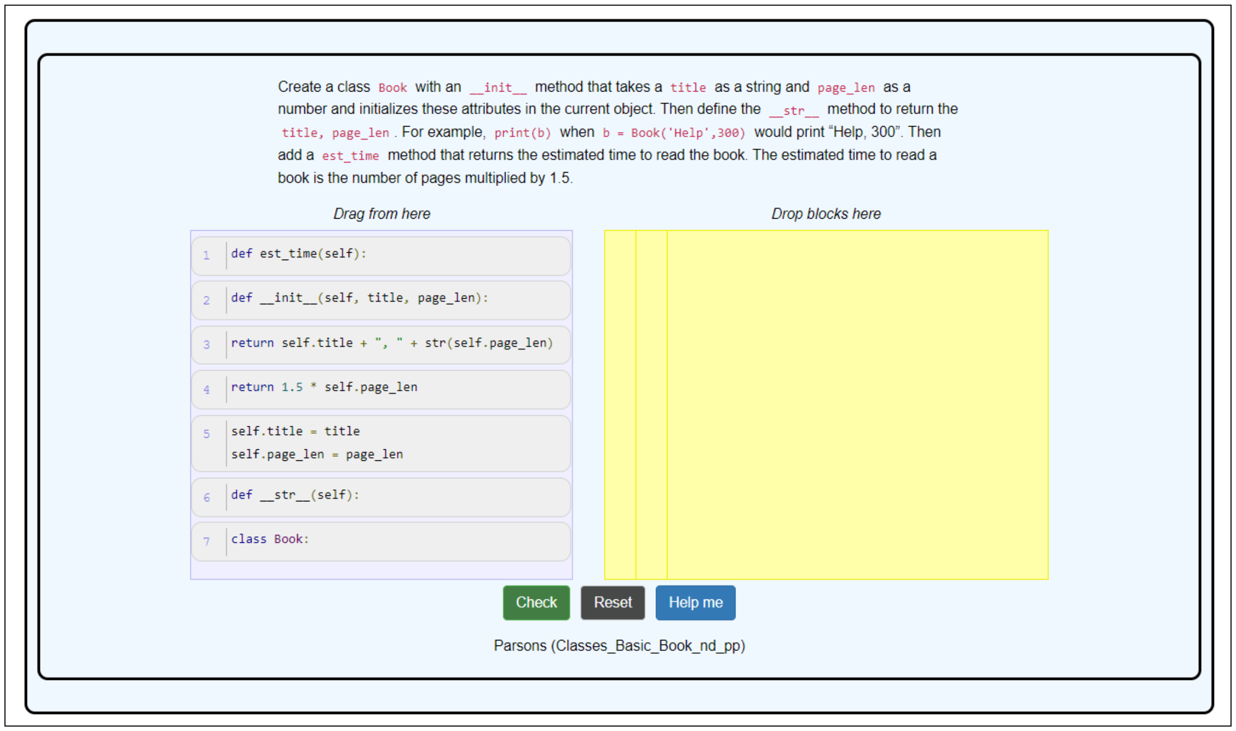
```
def __init__(self, name, age):
```

```
class Cat:
```

```
def __str__(self):
```

Parsons (Classes_Basic_Cat_nd_pp)

Figure 53: *class-exp* - Define a cat class with name and age attributes and a method for making sound


Create a class `Book` with an `__init__` method that takes a `title` as a string and `page_len` as a number and initializes these attributes in the current object. Then define the `__str__` method to return the `title, page_len`. For example, `print(b)` when `b = Book('Help',300)` would print “Help, 300”. Then add a `est_time` method that returns the estimated time to read the book. The estimated time to read a book is the number of pages multiplied by 1.5.

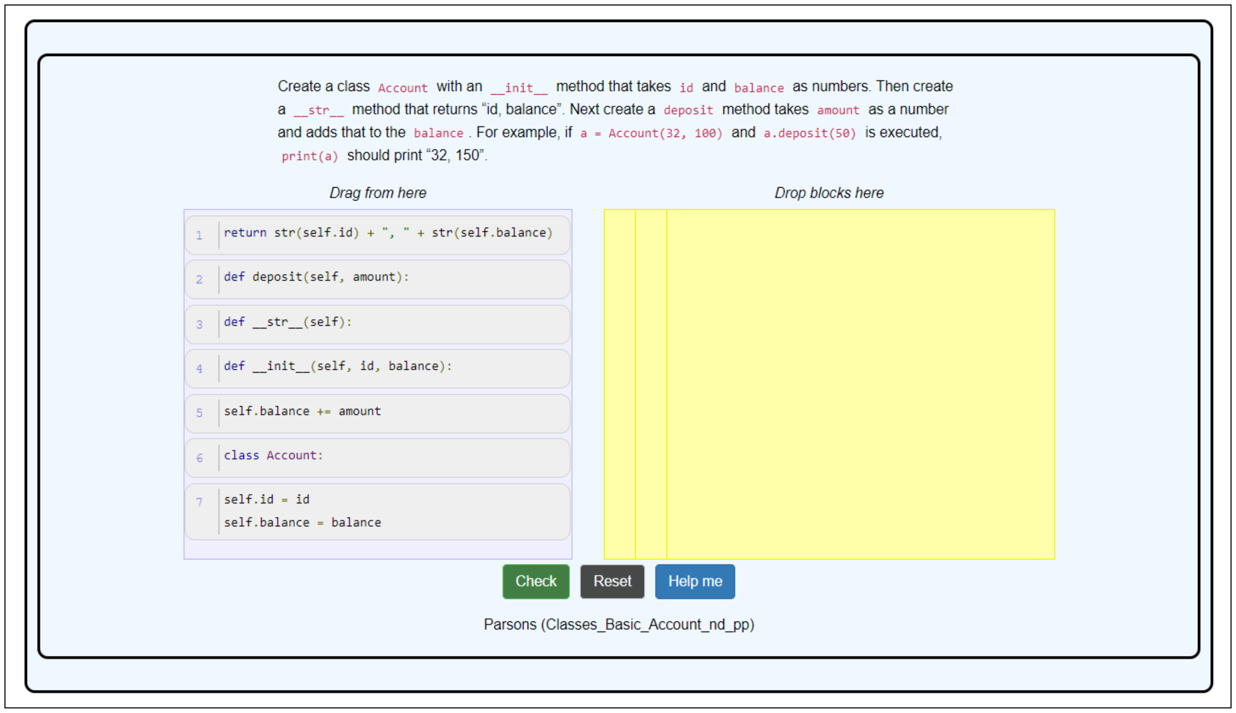
Drag from here

```
def est_time(self):
```
```
def __init__(self, title, page_len):
```
```
return self.title + ", " + str(self.page_len)
```
```
return 1.5 * self.page_len
```
```
self.title = title
self.page_len = page_len
```
```
def __str__(self):
```
```
class Book:
```

Drop blocks here

Parsons (Classes_Basic_Book_nd_pp)

Figure 54: ***class-exp*** **- Define a book class with title and length attributes and a string-conversion method**


Create a class `Account` with an `__init__` method that takes `id` and `balance` as numbers. Then create a `__str__` method that returns “id, balance”. Next create a `deposit` method takes `amount` as a number and adds that to the `balance`. For example, if `a = Account(32, 100)` and `a.deposit(50)` is executed, `print(a)` should print “32, 150”.

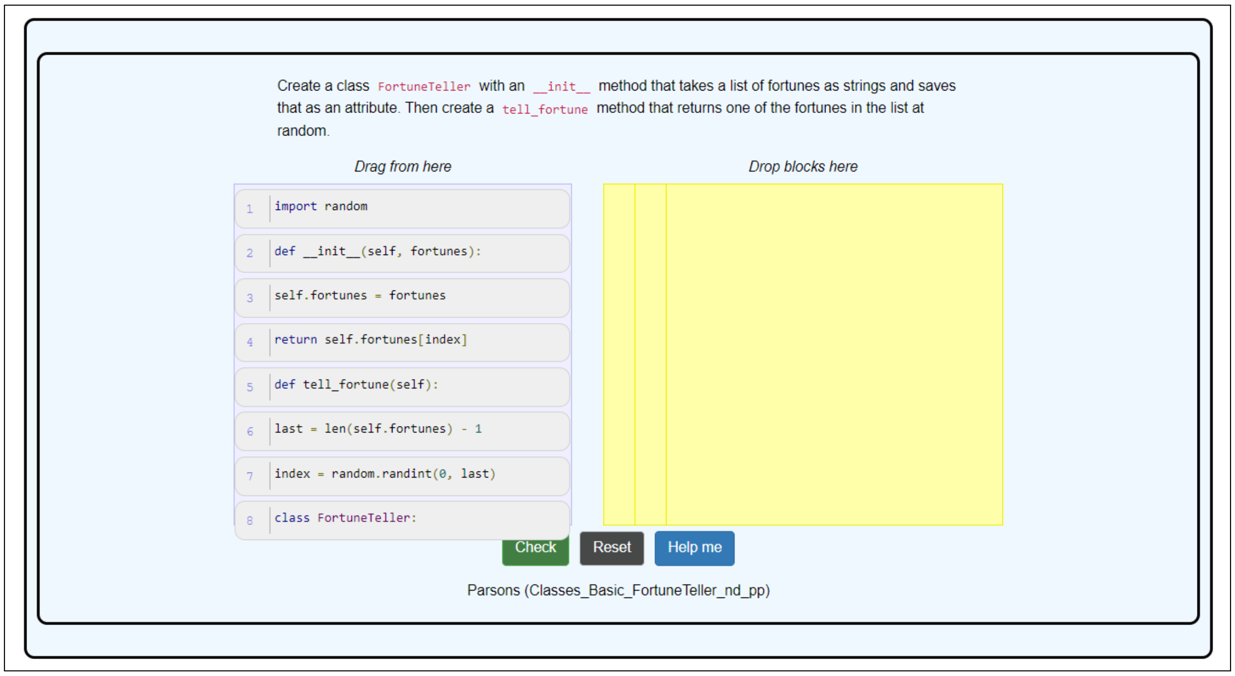
Drag from here

```
return str(self.id) + ", " + str(self.balance)
```
```
def deposit(self, amount):
```
```
def __str__(self):
```
```
def __init__(self, id, balance):
```
```
self.balance += amount
```
```
class Account:
```
```
self.id = id
self.balance = balance
```

Drop blocks here

Parsons (Classes_Basic_Account_nd_pp)

Figure 55: ***class-exp*** **- Define a bank account class with identifier and balance attributes and a deposit method**

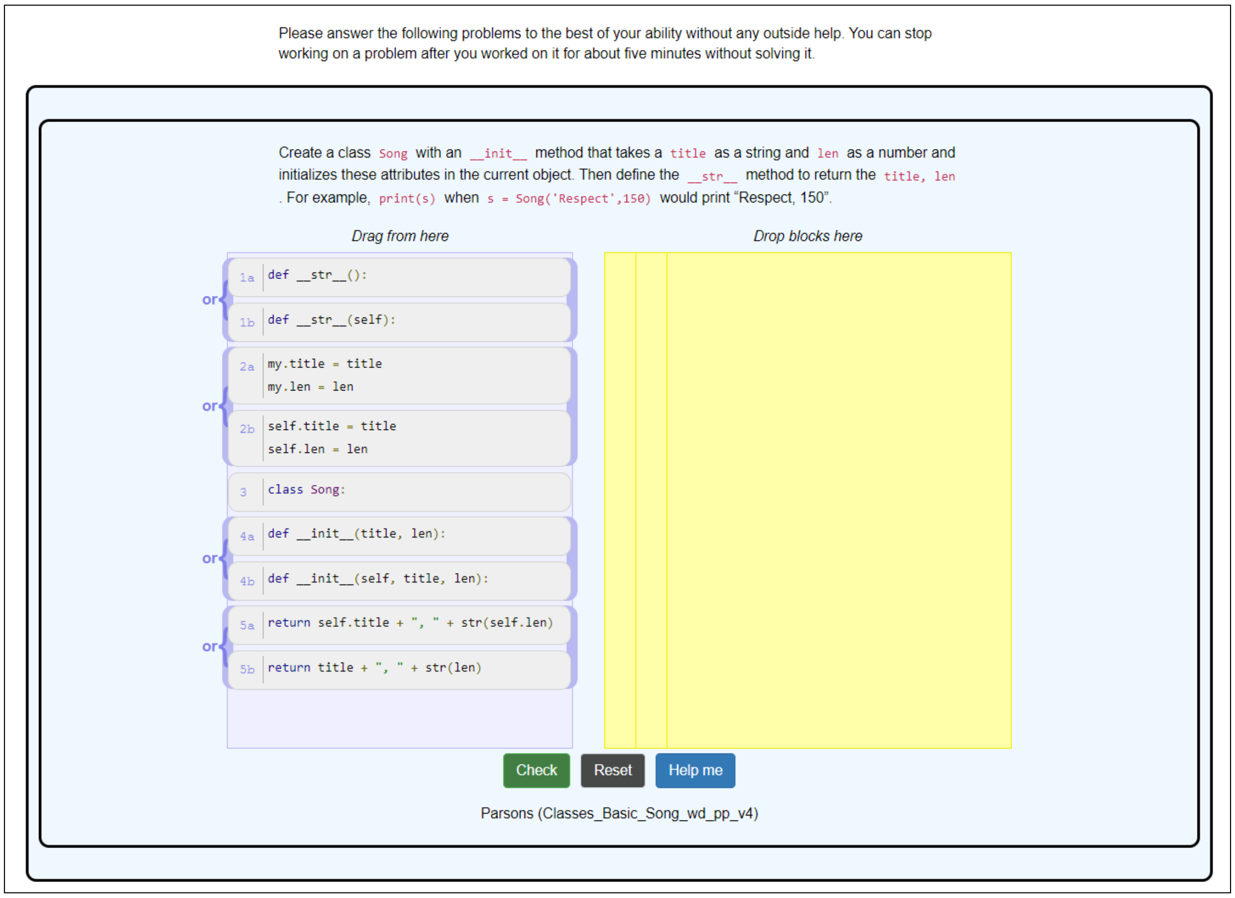


Create a class `FortuneTeller` with an `__init__` method that takes a list of fortunes as strings and saves that as an attribute. Then create a `tell_fortune` method that returns one of the fortunes in the list at random.

```
import random
```
```
def __init__(self, fortunes):
```
```
self.fortunes = fortunes
```
```
return self.fortunes[index]
```
```
def tell_fortune(self):
```
```
last = len(self.fortunes) - 1
```
```
index = random.randint(0, last)
```
```
class FortuneTeller:
```

Parsons (Classes_Basic_FortuneTeller_nd_pp)

Figure 56: *class-exp* - Define a fortune teller class which randomly selects a fortune from a predefined list

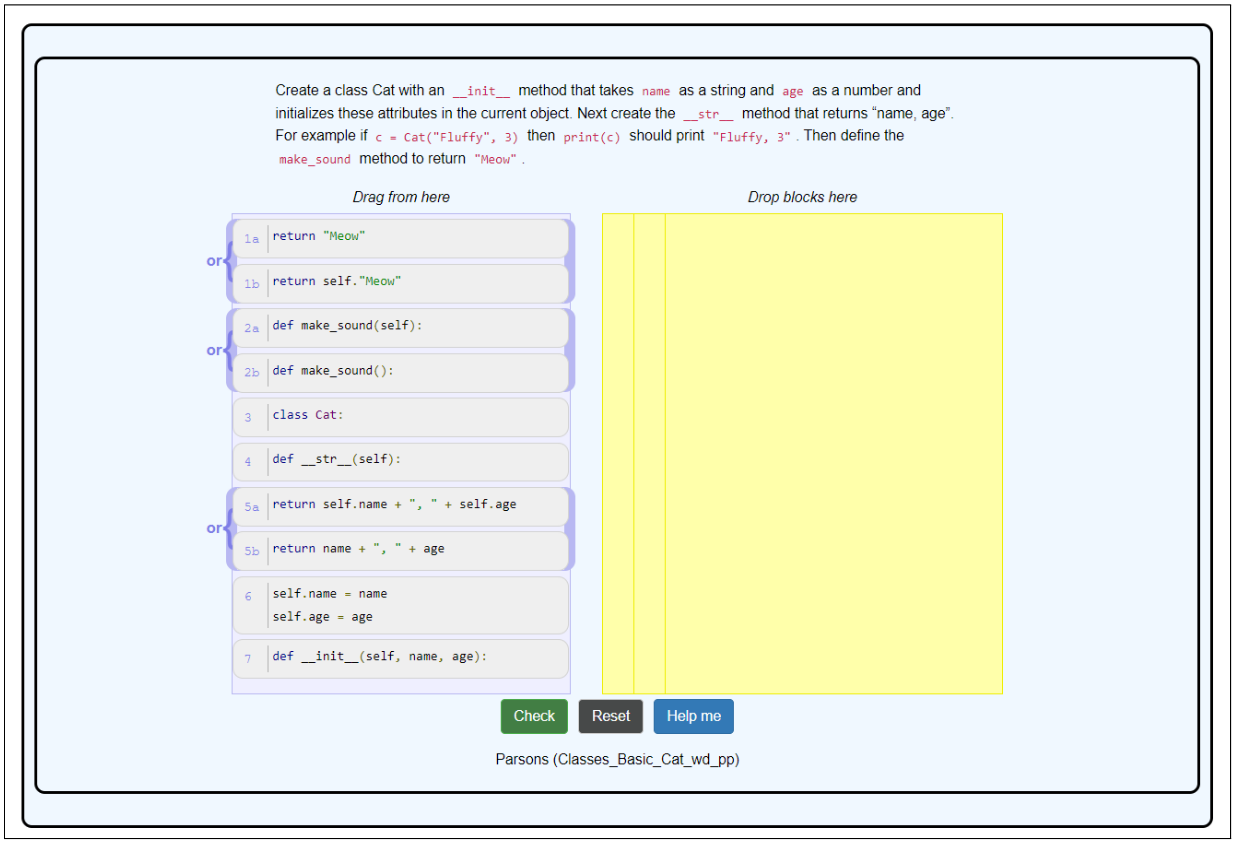

Please answer the following problems to the best of your ability without any outside help. You can stop working on a problem after you worked on it for about five minutes without solving it.

Create a class `Song` with an `__init__` method that takes a `title` as a string and `len` as a number and initializes these attributes in the current object. Then define the `__str__` method to return the `title, len`. For example, `print(s)` when `s = Song('Respect',150)` would print "Respect, 150".

```
1a  def __str__():
or
1b  def __str__(self):

2a  my.title = title
    my.len = len
or
2b  self.title = title
    self.len = len

3   class Song:

4a  def __init__(title, len):
or
4b  def __init__(self, title, len):

5a  return self.title + ", " + str(self.len)
or
5b  return title + ", " + str(len)
```

Parsons (Classes_Basic_Song_wd_pp_v4)

Figure 57: *class-exp* - Define a song class with name and duration attributes (with distractors)

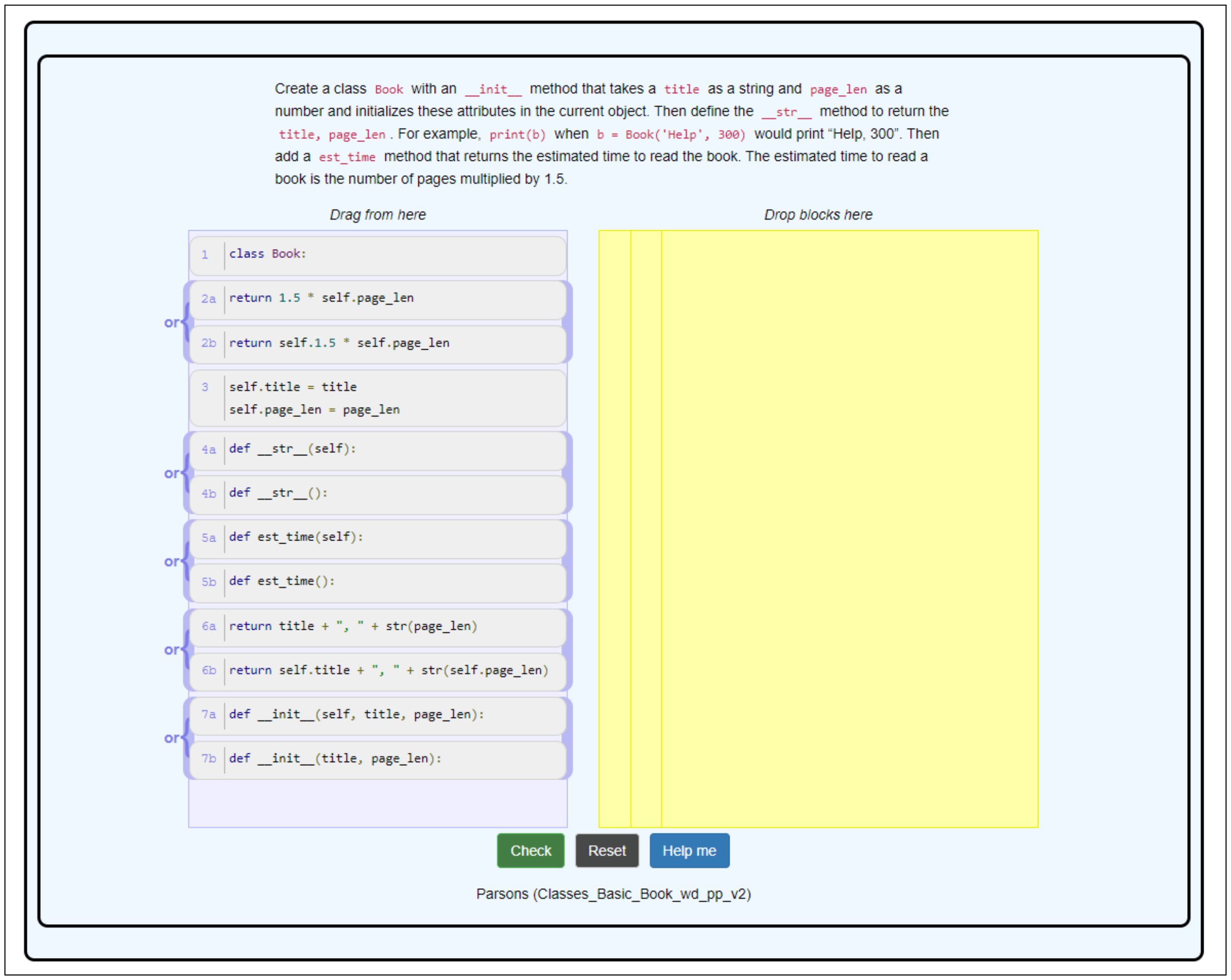


Create a class Cat with an `__init__` method that takes `name` as a string and `age` as a number and initializes these attributes in the current object. Next create the `__str__` method that returns "name, age". For example if `c = Cat("Fluffy", 3)` then `print(c)` should print `"Fluffy, 3"` . Then define the `make_sound` method to return `"Meow"` .

```
1a return "Meow"
1b return self."Meow"
2a def make_sound(self):
2b def make_sound():
3  class Cat:
4  def __str__(self):
5a return self.name + ", " + self.age
5b return name + ", " + age
6  self.name = name
   self.age = age
7  def __init__(self, name, age):
```

Parsons (Classes_Basic_Cat_wd_pp)

Figure 58: *class-exp* Define a cat class with name and age attributes and a method for making sound (with distractors)


Create a class `Book` with an `__init__` method that takes a `title` as a string and `page_len` as a number and initializes these attributes in the current object. Then define the `__str__` method to return the `title, page_len`. For example, `print(b)` when `b = Book('Help', 300)` would print “Help, 300”. Then add a `est_time` method that returns the estimated time to read the book. The estimated time to read a book is the number of pages multiplied by 1.5.

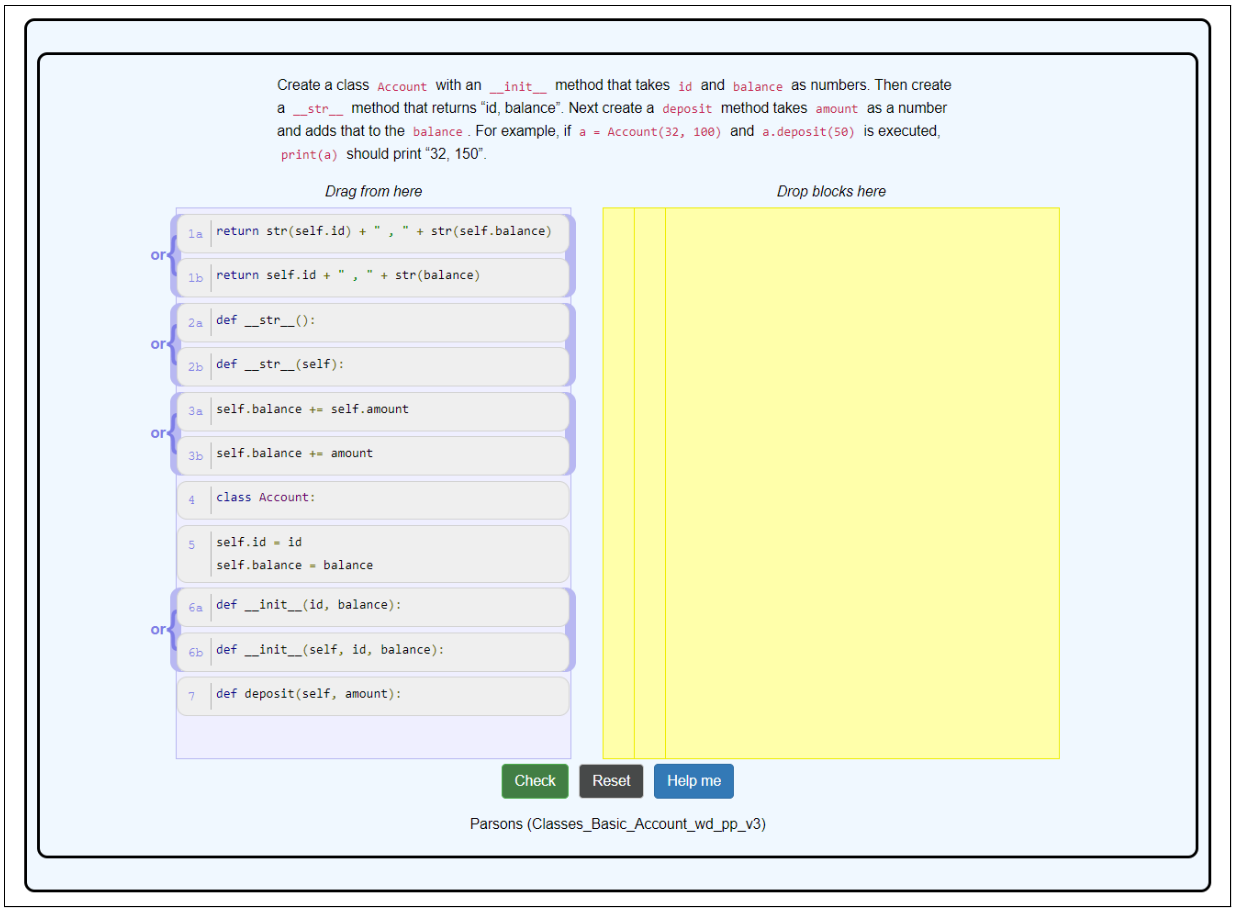
Drag from here

```
1   class Book:

2a  return 1.5 * self.page_len
or
2b  return self.1.5 * self.page_len

3   self.title = title
    self.page_len = page_len

4a  def __str__(self):
or
4b  def __str__():

5a  def est_time(self):
or
5b  def est_time():

6a  return title + ", " + str(page_len)
or
6b  return self.title + ", " + str(self.page_len)

7a  def __init__(self, title, page_len):
or
7b  def __init__(title, page_len):
```

Drop blocks here

Parsons (Classes_Basic_Book_wd_pp_v2)

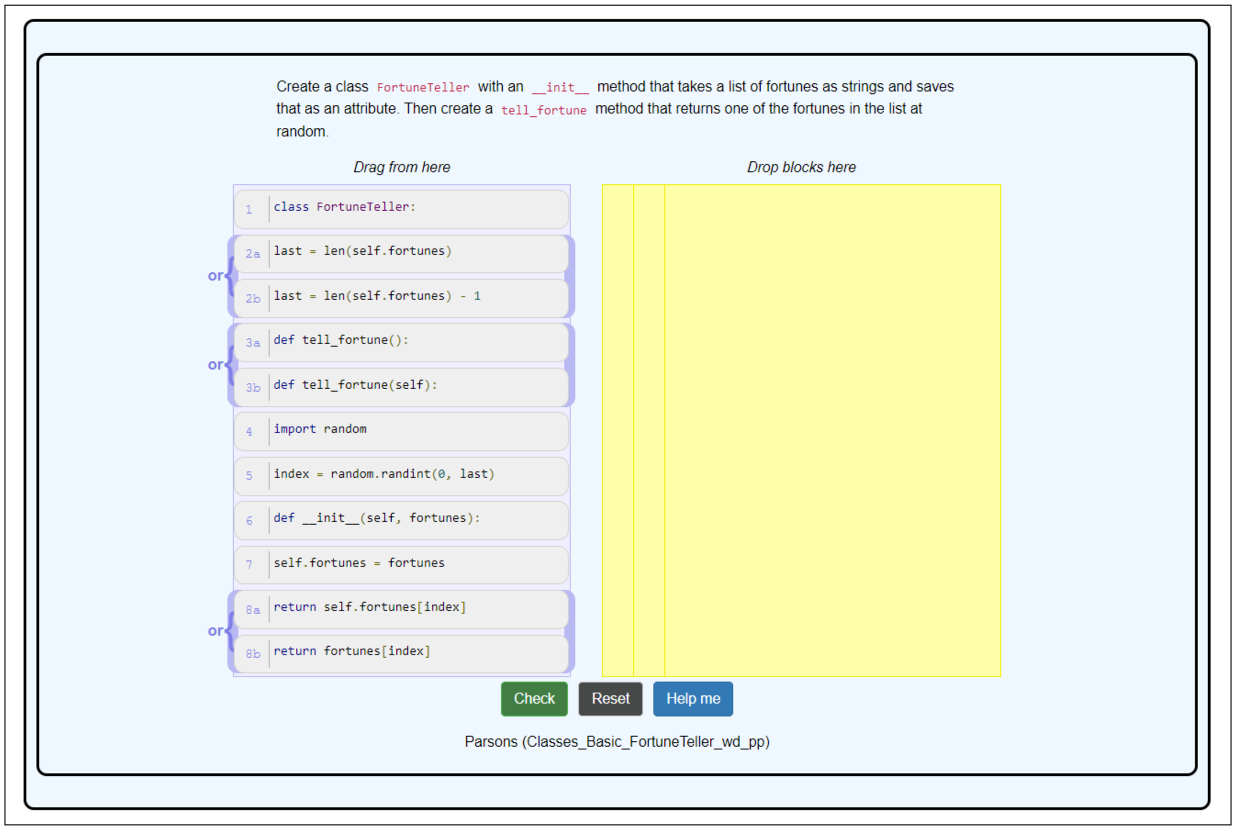
Figure 59: ***class-exp-*** **Define a book class with title and length attributes and with a string-conversion method (with distractors)**


Create a class `Account` with an `__init__` method that takes `id` and `balance` as numbers. Then create a `__str__` method that returns “id, balance”. Next create a `deposit` method takes `amount` as a number and adds that to the `balance` . For example, if `a = Account(32, 100)` and `a.deposit(50)` is executed, `print(a)` should print “32, 150”.

```
1a  return str(self.id) + " , " + str(self.balance)
or
1b  return self.id + " , " + str(balance)
2a  def __str__():
or
2b  def __str__(self):
3a  self.balance += self.amount
or
3b  self.balance += amount
4   class Account:
5   self.id = id
    self.balance = balance
6a  def __init__(id, balance):
or
6b  def __init__(self, id, balance):
7   def deposit(self, amount):
```

Parsons (Classes_Basic_Account_wd_pp_v3)

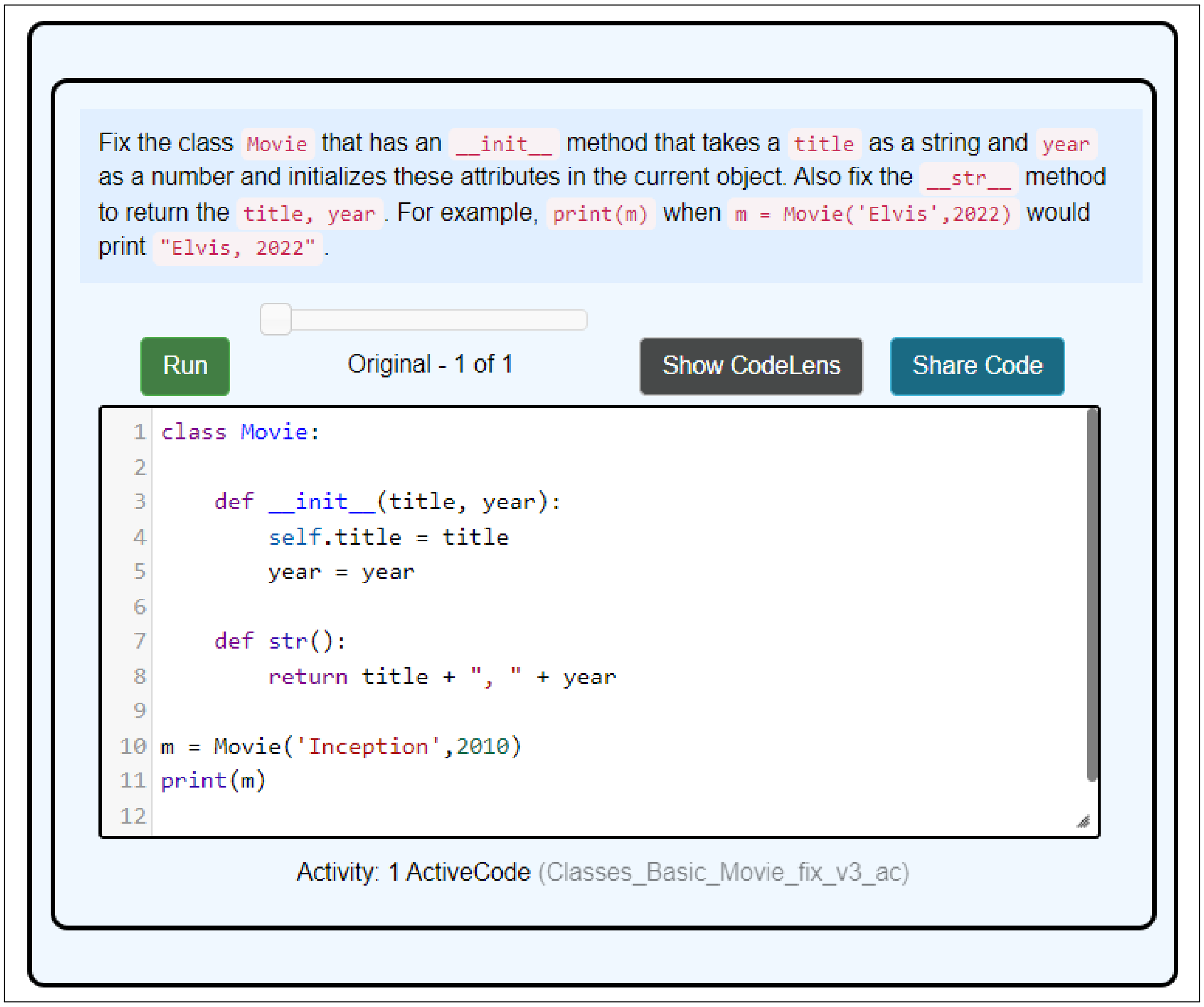
Figure 60: *class-exp* - Define a bank account class with identifier and balance attributes and a deposit method (with distractors)


Create a class FortuneTeller with an __init__ method that takes a list of fortunes as strings and saves that as an attribute. Then create a tell_fortune method that returns one of the fortunes in the list at random.

```
1  class FortuneTeller:
2a last = len(self.fortunes)
2b last = len(self.fortunes) - 1
3a def tell_fortune():
3b def tell_fortune(self):
4  import random
5  index = random.randint(0, last)
6  def __init__(self, fortunes):
7  self.fortunes = fortunes
8a return self.fortunes[index]
8b return fortunes[index]
```

Parsons (Classes_Basic_FortuneTeller_wd_pp)

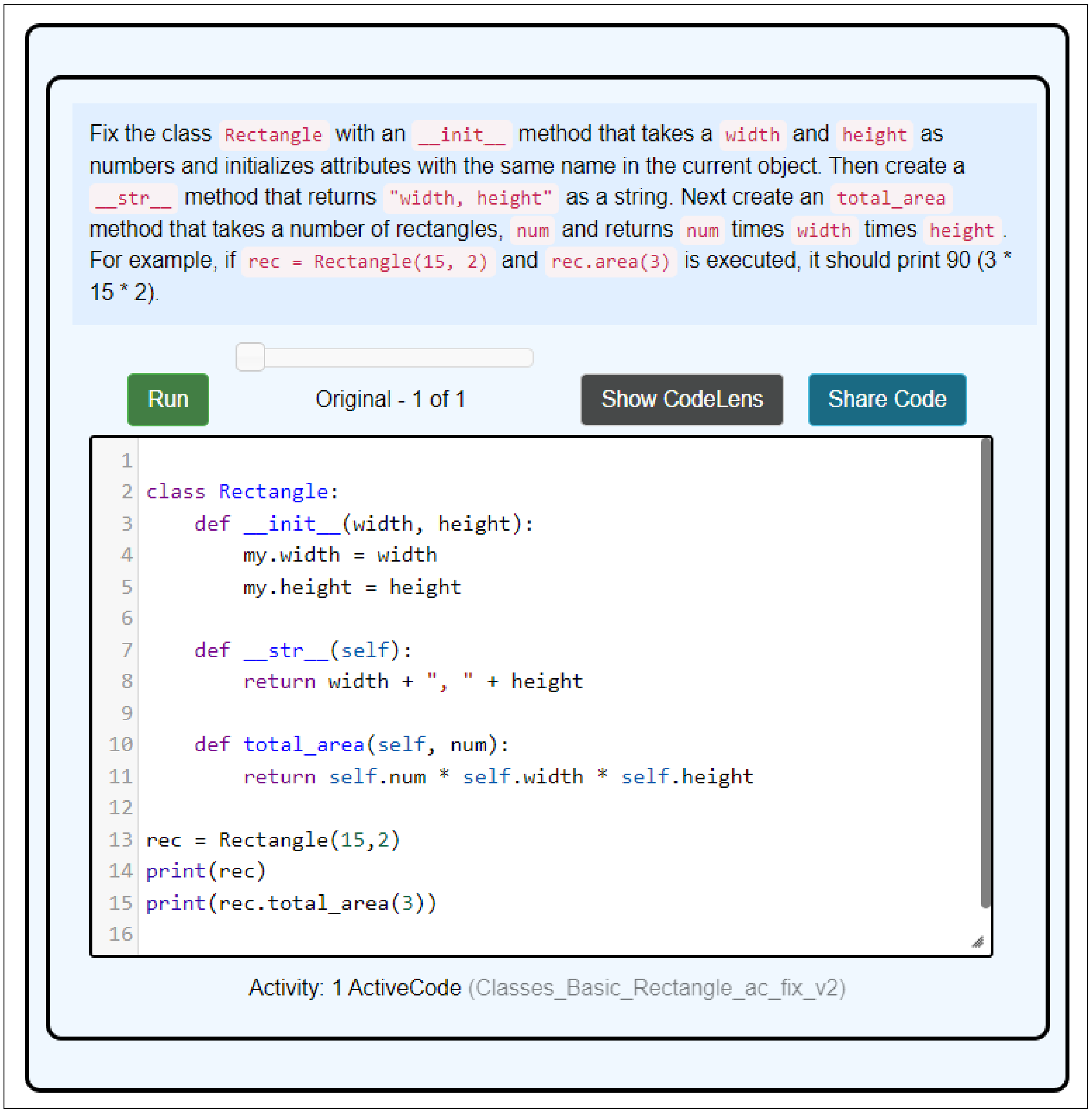
Figure 61: *class-exp* - Define a fortune teller class which randomly selects a fortune from a predefined list *class-exp* (with distractors)


Fix the class `Movie` that has an `__init__` method that takes a `title` as a string and `year` as a number and initializes these attributes in the current object. Also fix the `__str__` method to return the `title, year`. For example, `print(m)` when `m = Movie('Elvis',2022)` would print `"Elvis, 2022"`.

```
class Movie:

    def __init__(title, year):
        self.title = title
        year = year

    def str():
        return title + ", " + year

m = Movie('Inception',2010)
print(m)

```

Activity: 1 ActiveCode (Classes_Basic_Movie_fix_v3_ac)

Figure 62: ***class-exp*** **- Posttest task for defining a movie class**

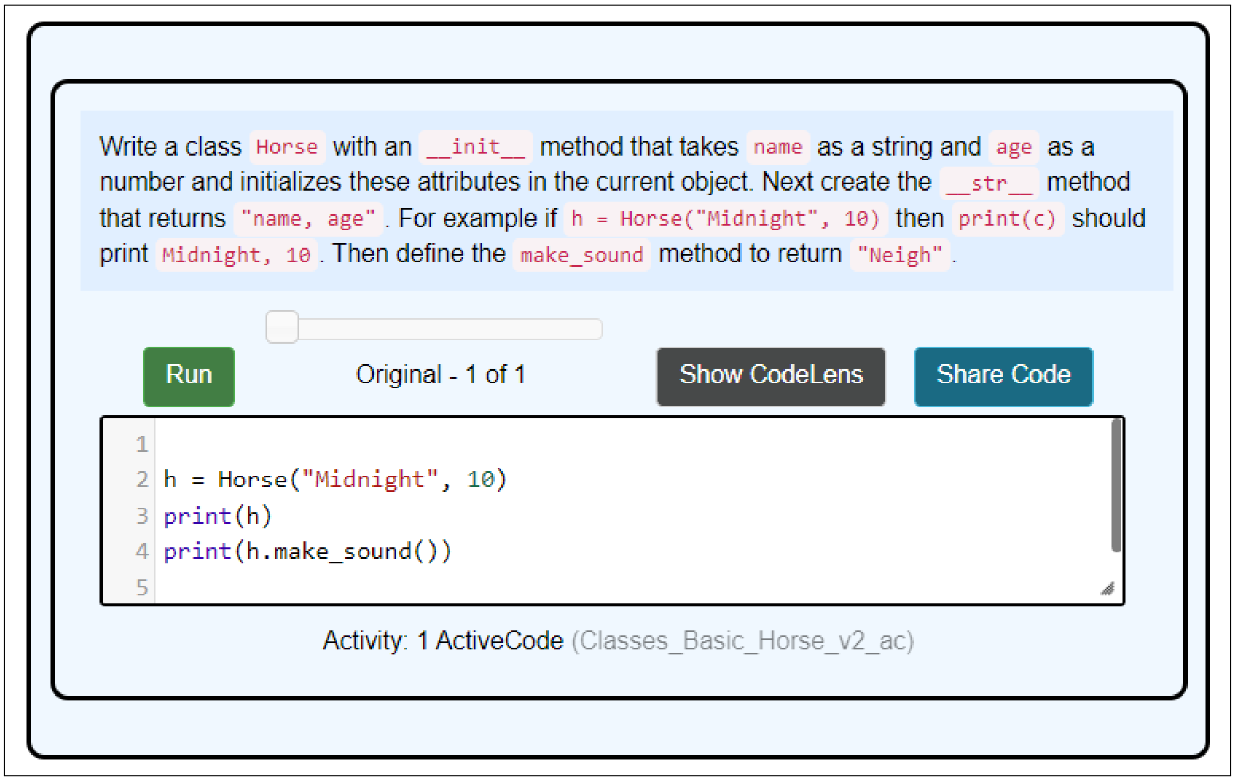


Fix the class `Rectangle` with an `__init__` method that takes a `width` and `height` as numbers and initializes attributes with the same name in the current object. Then create a `__str__` method that returns `"width, height"` as a string. Next create an `total_area` method that takes a number of rectangles, `num` and returns `num` times `width` times `height`. For example, if `rec = Rectangle(15, 2)` and `rec.area(3)` is executed, it should print 90 (3 * 15 * 2).

```python

class Rectangle:
    def __init__(width, height):
        my.width = width
        my.height = height

    def __str__(self):
        return width + ", " + height

    def total_area(self, num):
        return self.num * self.width * self.height

rec = Rectangle(15,2)
print(rec)
print(rec.total_area(3))

```

Activity: 1 ActiveCode (Classes_Basic_Rectangle_ac_fix_v2)

Figure 63: *class-exp* - Posttest task for defining a rectangle class with a string-conversion method

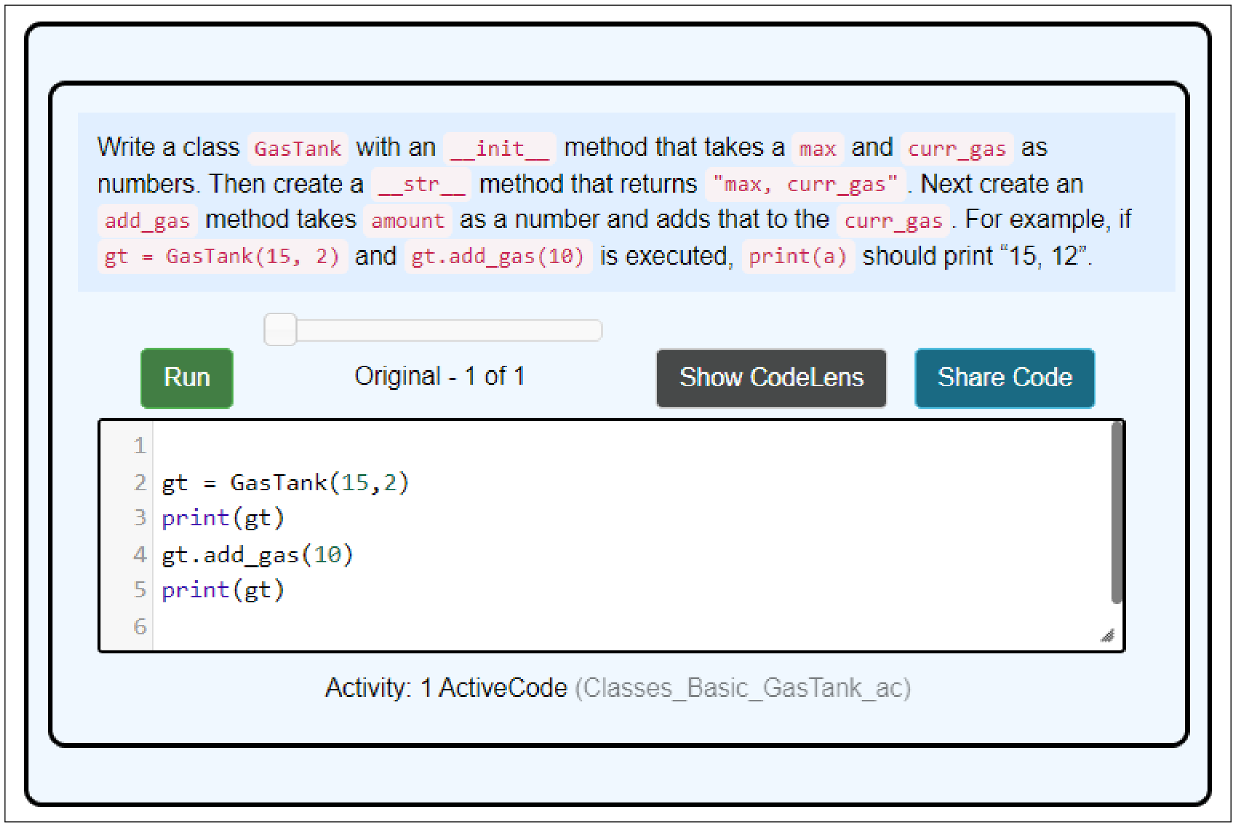


Write a class `Horse` with an `__init__` method that takes `name` as a string and `age` as a number and initializes these attributes in the current object. Next create the `__str__` method that returns `"name, age"`. For example if `h = Horse("Midnight", 10)` then `print(c)` should print `Midnight, 10`. Then define the `make_sound` method to return `"Neigh"`.

```

h = Horse("Midnight", 10)
print(h)
print(h.make_sound())

```

Activity: 1 ActiveCode (Classes_Basic_Horse_v2_ac)

Figure 64: *class-exp* - Posttest task for defining a horse class and a method for making sound

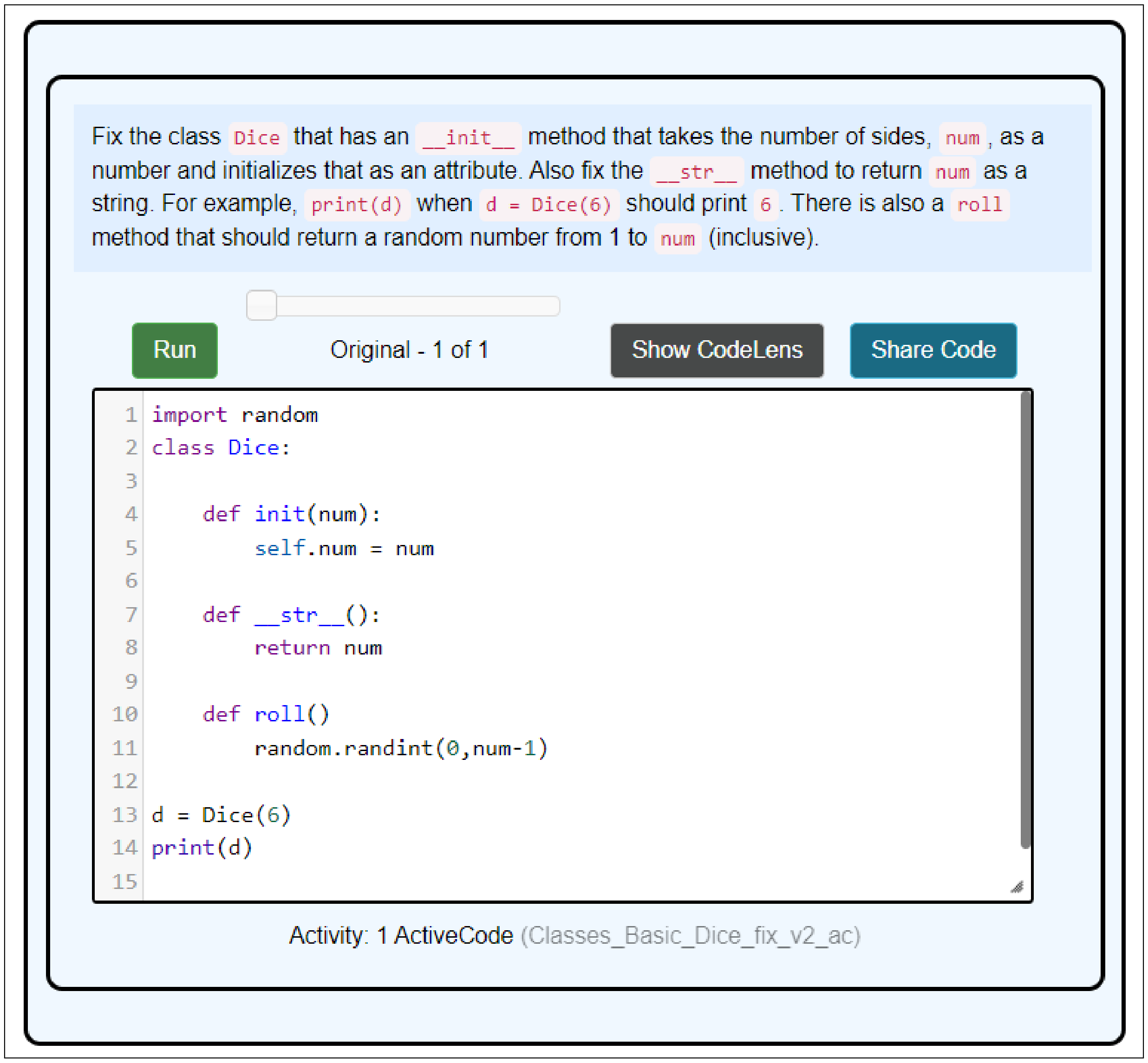


Write a class `GasTank` with an `__init__` method that takes a `max` and `curr_gas` as numbers. Then create a `__str__` method that returns `"max, curr_gas"`. Next create an `add_gas` method takes `amount` as a number and adds that to the `curr_gas`. For example, if `gt = GasTank(15, 2)` and `gt.add_gas(10)` is executed, `print(a)` should print “15, 12”.

```python

gt = GasTank(15,2)
print(gt)
gt.add_gas(10)
print(gt)

```

Activity: 1 ActiveCode (Classes_Basic_GasTank_ac)

Figure 65: *class-exp* - Posttest task for defining a gas tank class and a method to add gas



Fix the class `Dice` that has an `__init__` method that takes the number of sides, `num`, as a number and initializes that as an attribute. Also fix the `__str__` method to return `num` as a string. For example, `print(d)` when `d = Dice(6)` should print `6`. There is also a `roll` method that should return a random number from 1 to `num` (inclusive).

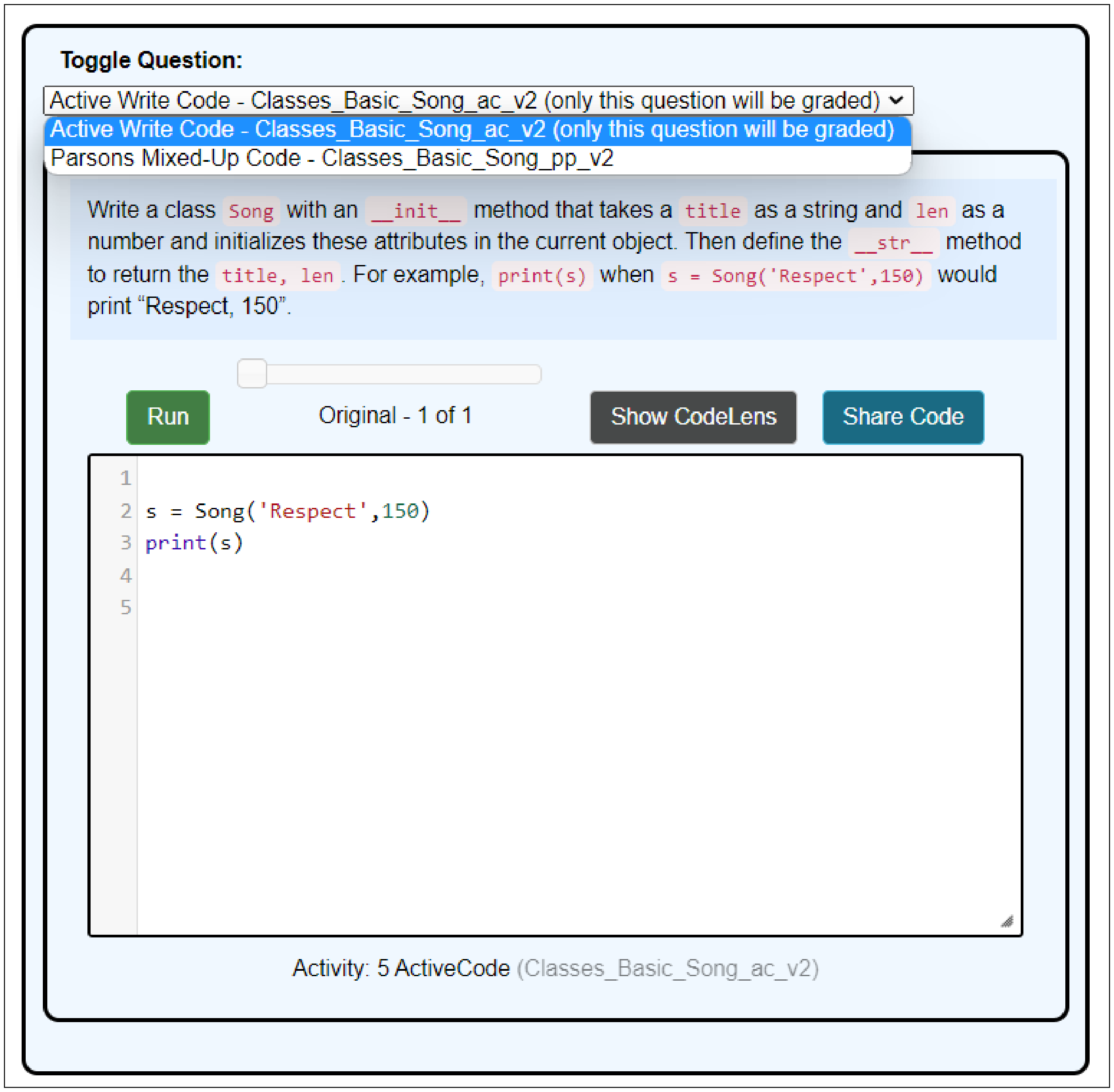
```
import random
class Dice:

    def init(num):
        self.num = num

    def __str__():
        return num

    def roll()
        random.randint(0,num-1)

d = Dice(6)
print(d)

```

Activity: 1 ActiveCode (Classes_Basic_Dice_fix_v2_ac)

Figure 66: *class-exp* - Posttest task for defining a dice class with a given number of sides and a roll method

A.7 *class-tog*

Similar to the previous activity, students are tasked with defining classes, with the same practice and post-test items. However, in this study, students are able to switch freely between a Parsons problem and code using a drop-down menu throughout the practice activities.

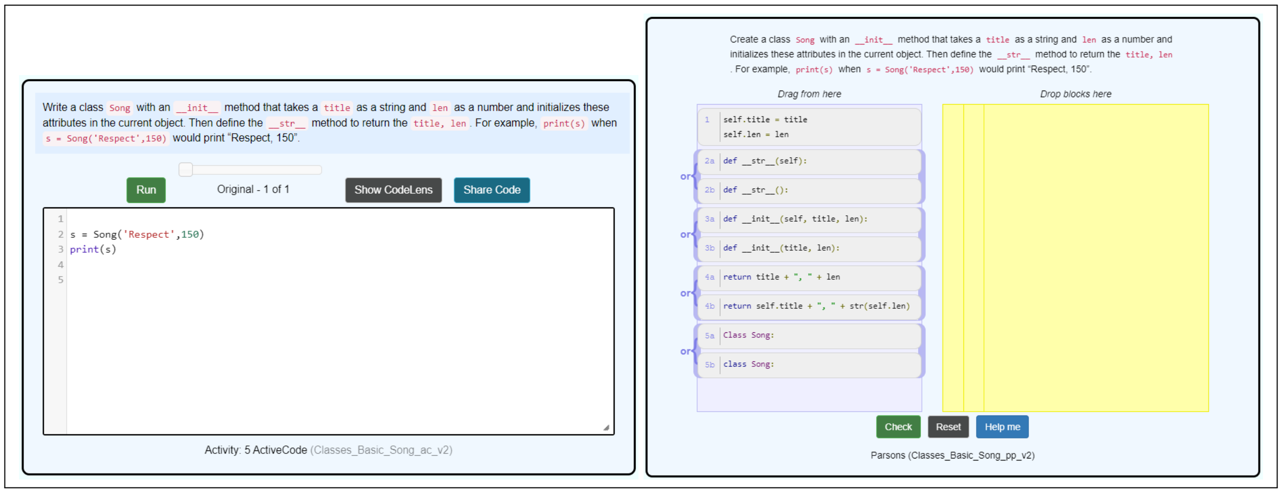

Toggle Question:
Active Write Code - Classes_Basic_Song_ac_v2 (only this question will be graded)
Active Write Code - Classes_Basic_Song_ac_v2 (only this question will be graded)
Parsons Mixed-Up Code - Classes_Basic_Song_pp_v2
Run
Original - 1 of 1
Show CodeLens
Share Code

Write a class `Song` with an `__init__` method that takes a `title` as a string and `len` as a number and initializes these attributes in the current object. Then define the `__str__` method to return the `title, len`. For example, `print(s)` when `s = Song('Respect',150)` would print "Respect, 150".

```

s = Song('Respect',150)
print(s)


```

Activity: 5 ActiveCode (Classes_Basic_Song_ac_v2)

Figure 67: Students have the ability to toggle between code and a Parsons problem


Write a class `Song` with an `__init__` method that takes a `title` as a string and `len` as a number and initializes these attributes in the current object. Then define the `__str__` method to return the `title, len`. For example, `print(s)` when `s = Song('Respect',150)` would print "Respect, 150".

```

s = Song('Respect',150)
print(s)


```

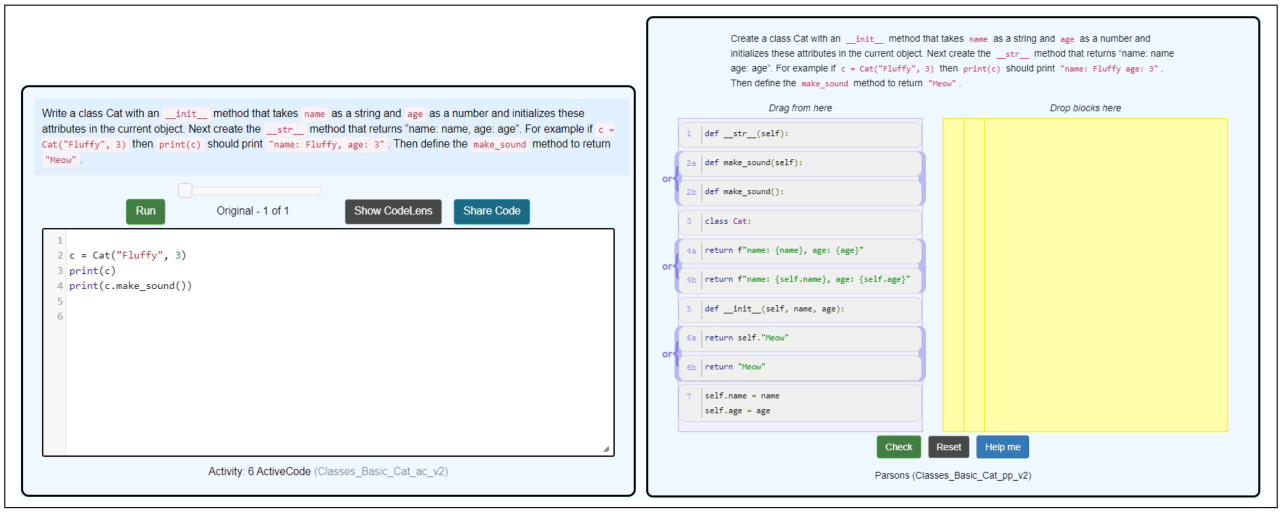
Activity: 5 ActiveCode (Classes_Basic_Song_ac_v2)

Create a class `Song` with an `__init__` method that takes a `title` as a string and `len` as a number and initializes these attributes in the current object. Then define the `__str__` method to return the `title, len`. For example, `print(s)` when `s = Song('Respect',150)` would print "Respect, 150".

```
1   self.title = title
    self.len = len
2a  def __str__(self):
or
2b  def __str__():
3a  def __init__(self, title, len):
or
3b  def __init__(title, len):
4a  return title + ", " + len
or
4b  return self.title + ", " + str(self.len)
5a  Class Song:
or
5b  class Song:
```

Parsons (Classes_Basic_Song_pp_v2)

Figure 68: *class-tog* - Define a song class with name and duration attributes


Write a class Cat with an `__init__` method that takes `name` as a string and `age` as a number and initializes these attributes in the current object. Next create the `__str__` method that returns "name: name, age: age". For example if `c = Cat("Fluffy", 3)` then `print(c)` should print `"name: Fluffy, age: 3"`. Then define the `make_sound` method to return `"Meow"`.

```

c = Cat("Fluffy", 3)
print(c)
print(c.make_sound())


```

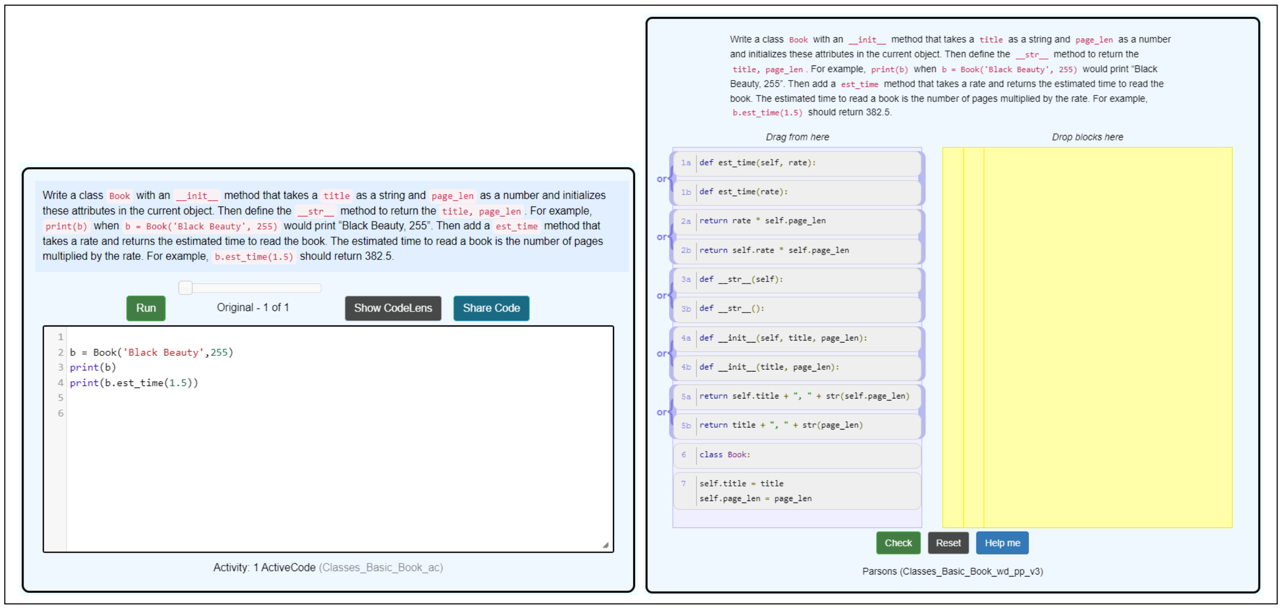
Activity: 6 ActiveCode (Classes_Basic_Cat_ac_v2)

Create a class Cat with an `__init__` method that takes `name` as a string and `age` as a number and initializes these attributes in the current object. Next create the `__str__` method that returns "name: name age: age". For example if `c = Cat("Fluffy", 3)` then `print(c)` should print `"name: Fluffy age: 3"`. Then define the `make_sound` method to return `"Meow"`.

```
1   def __str__(self):
2a  def make_sound(self):
or
2b  def make_sound():
3   class Cat:
4a  return f"name: {name}, age: {age}"
or
4b  return f"name: {self.name}, age: {self.age}"
5   def __init__(self, name, age):
6a  return self."Meow"
or
6b  return "Meow"
7   self.name = name
    self.age = age
```

Parsons (Classes_Basic_Cat_pp_v2)

Figure 69: *class-tog* - Define a cat class with name and age attributes and with a method for making sound



Write a class `Book` with an `__init__` method that takes a `title` as a string and `page_len` as a number and initializes these attributes in the current object. Then define the `__str__` method to return the `title, page_len`. For example, `print(b)` when `b = Book('Black Beauty', 255)` would print "Black Beauty, 255". Then add a `est_time` method that takes a rate and returns the estimated time to read the book. The estimated time to read a book is the number of pages multiplied by the rate. For example, `b.est_time(1.5)` should return 382.5.

```
b = Book('Black Beauty',255)
print(b)
print(b.est_time(1.5))
```

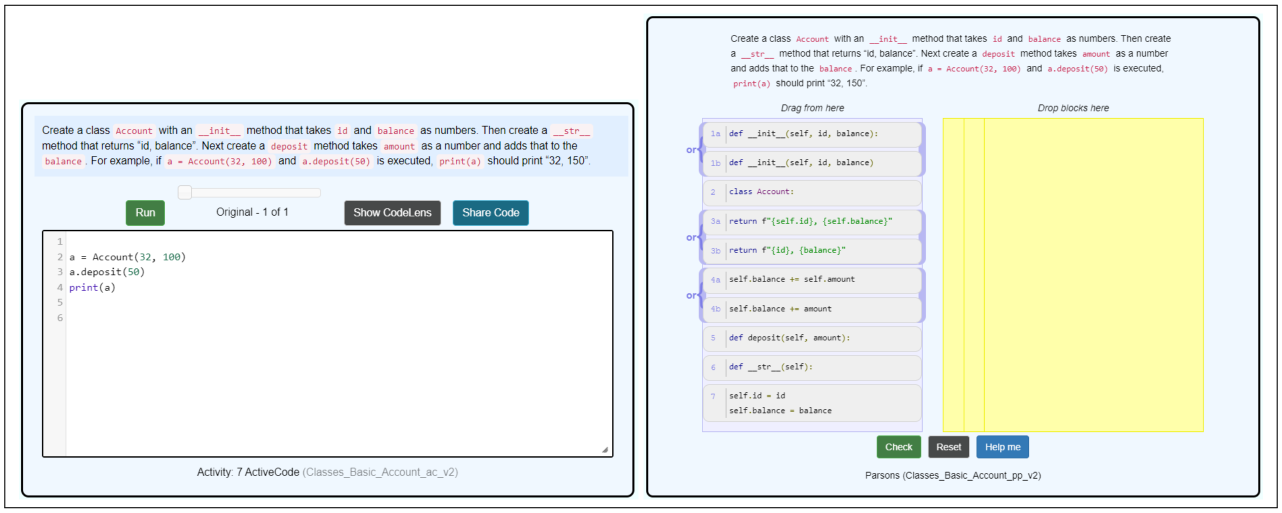
Activity: 1 ActiveCode (Classes_Basic_Book_ac)

Write a class `Book` with an `__init__` method that takes a `title` as a string and `page_len` as a number and initializes these attributes in the current object. Then define the `__str__` method to return the `title, page_len`. For example, `print(b)` when `b = Book('Black Beauty', 255)` would print "Black Beauty, 255". Then add a `est_time` method that takes a rate and returns the estimated time to read the book. The estimated time to read a book is the number of pages multiplied by the rate. For example, `b.est_time(1.5)` should return 382.5.

```
1a  def est_time(self, rate):
or
1b  def est_time(rate):
2a  return rate * self.page_len
or
2b  return self.rate * self.page_len
3a  def __str__(self):
or
3b  def __str__():
4a  def __init__(self, title, page_len):
or
4b  def __init__(title, page_len):
5a  return self.title + ", " + str(self.page_len)
or
5b  return title + ", " + str(page_len)
6   class Book:
7   self.title = title
    self.page_len = page_len
```

Parsons (Classes_Basic_Book_wd_pp_v3)

Figure 70: *class-tog* - Define a book class with title and length attributes and a string-conversion method



Create a class `Account` with an `__init__` method that takes `id` and `balance` as numbers. Then create a `__str__` method that returns "id, balance". Next create a `deposit` method takes `amount` as a number and adds that to the `balance`. For example, if `a = Account(32, 100)` and `a.deposit(50)` is executed, `print(a)` should print "32, 150".

```
a = Account(32, 100)
a.deposit(50)
print(a)
```

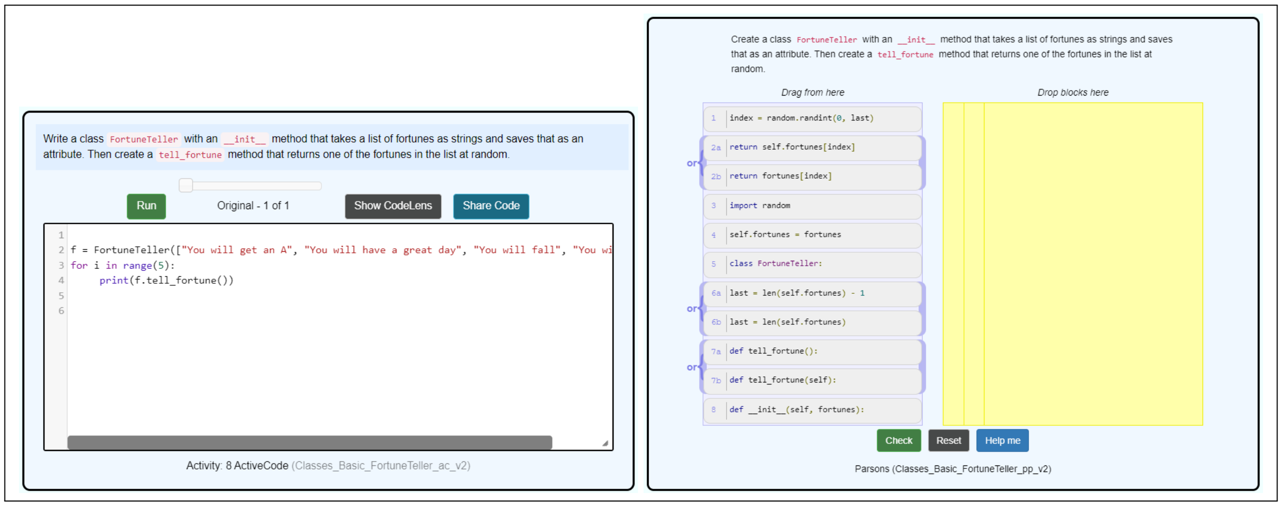
Activity: 7 ActiveCode (Classes_Basic_Account_ac_v2)

Create a class `Account` with an `__init__` method that takes `id` and `balance` as numbers. Then create a `__str__` method that returns "id, balance". Next create a `deposit` method takes `amount` as a number and adds that to the `balance`. For example, if `a = Account(32, 100)` and `a.deposit(50)` is executed, `print(a)` should print "32, 150".

```
1a  def __init__(self, id, balance):
or
1b  def __init__(self, id, balance)
2   class Account:
3a  return f"{self.id}, {self.balance}"
or
3b  return f"{id}, {balance}"
4a  self.balance += self.amount
or
4b  self.balance += amount
5   def deposit(self, amount):
6   def __str__(self):
7   self.id = id
    self.balance = balance
```

Parsons (Classes_Basic_Account_pp_v2)

Figure 71: *class-tog* - Define a bank account class with identifier and balance attributes and a deposit method



Write a class `FortuneTeller` with an `__init__` method that takes a list of fortunes as strings and saves that as an attribute. Then create a `tell_fortune` method that returns one of the fortunes in the list at random.

```

f = FortuneTeller(["You will get an A", "You will have a great day", "You will fall", "You wi
for i in range(5):
    print(f.tell_fortune())
```

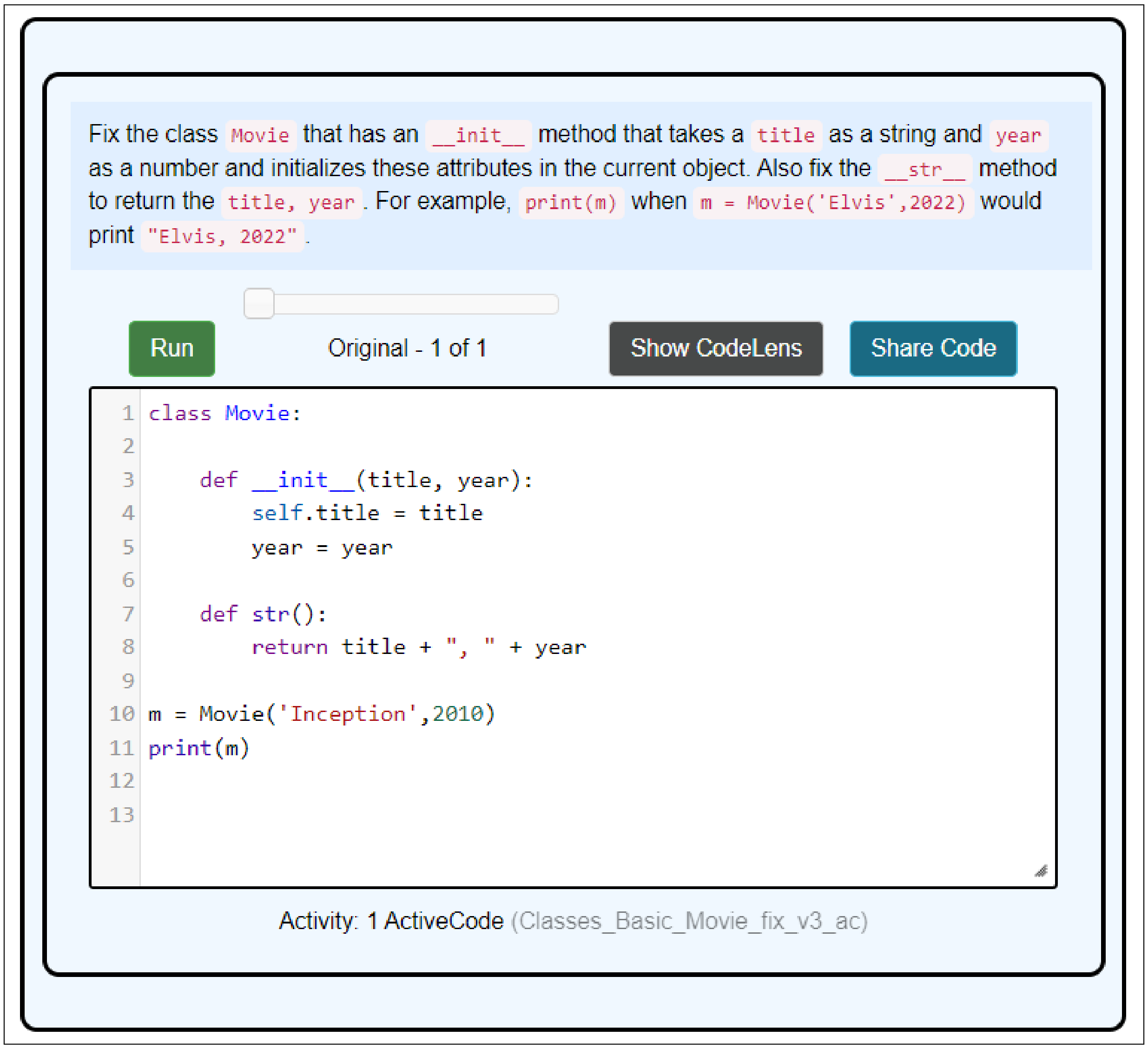
Activity: 8 ActiveCode (Classes_Basic_FortuneTeller_ac_v2)

Create a class `FortuneTeller` with an `__init__` method that takes a list of fortunes as strings and saves that as an attribute. Then create a `tell_fortune` method that returns one of the fortunes in the list at random.

```
1  index = random.randint(0, last)
2a return self.fortunes[index]
   or
2b return fortunes[index]
3  import random
4  self.fortunes = fortunes
5  class FortuneTeller:
6a last = len(self.fortunes) - 1
   or
6b last = len(self.fortunes)
7a def tell_fortune():
   or
7b def tell_fortune(self):
8  def __init__(self, fortunes):
```

Parsons (Classes_Basic_FortuneTeller_pp_v2)

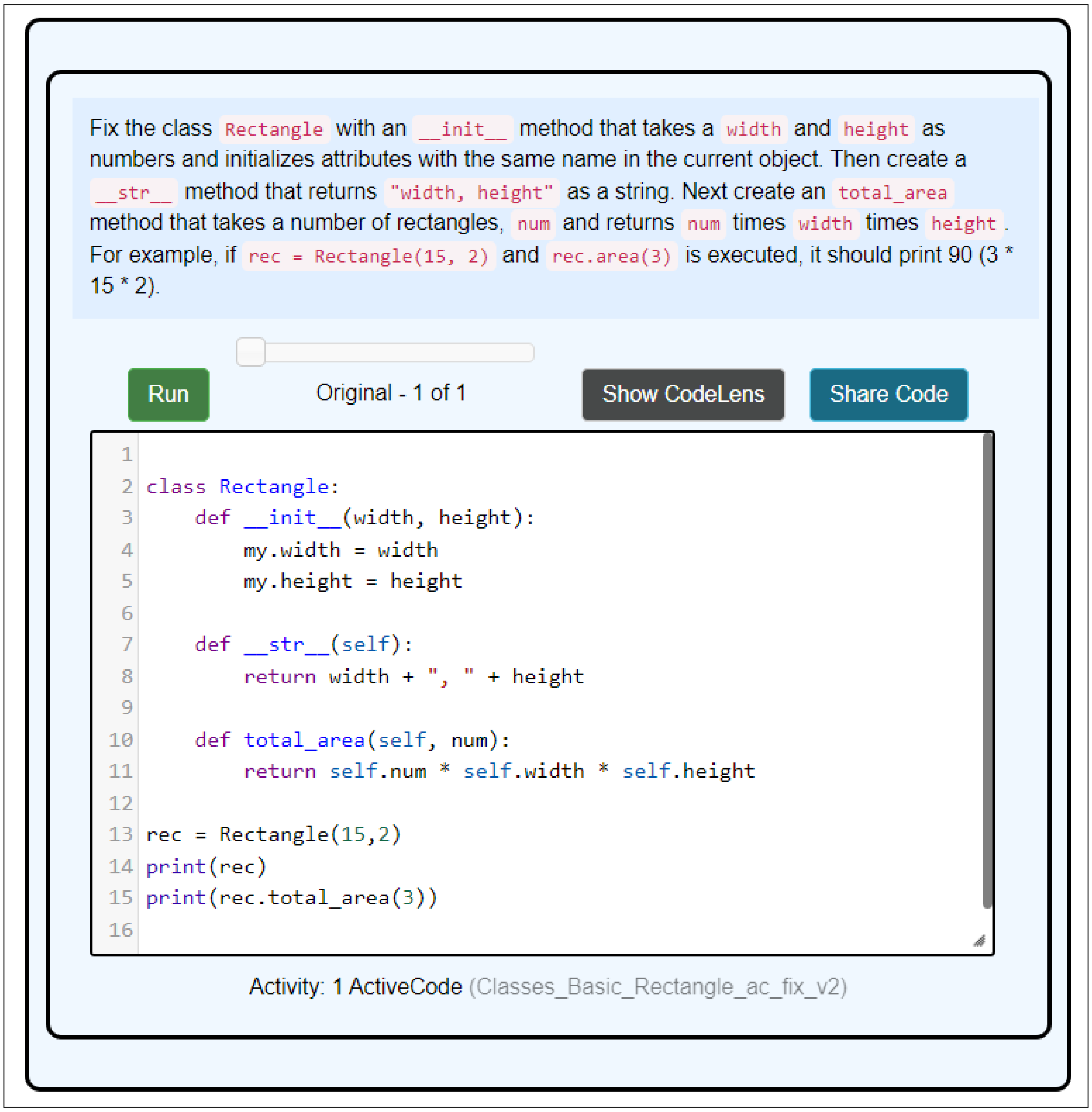
Figure 72: *class-tog* - Define a fortune teller class which randomly selects a fortune from a predefined list (with distractors)


Fix the class `Movie` that has an `__init__` method that takes a `title` as a string and `year` as a number and initializes these attributes in the current object. Also fix the `__str__` method to return the `title, year`. For example, `print(m)` when `m = Movie('Elvis',2022)` would print `"Elvis, 2022"`.

```
class Movie:

    def __init__(title, year):
        self.title = title
        year = year

    def str():
        return title + ", " + year

m = Movie('Inception',2010)
print(m)


```

Activity: 1 ActiveCode (Classes_Basic_Movie_fix_v3_ac)

Figure 73: *class-tog* - Post-postest task for defining a movie class

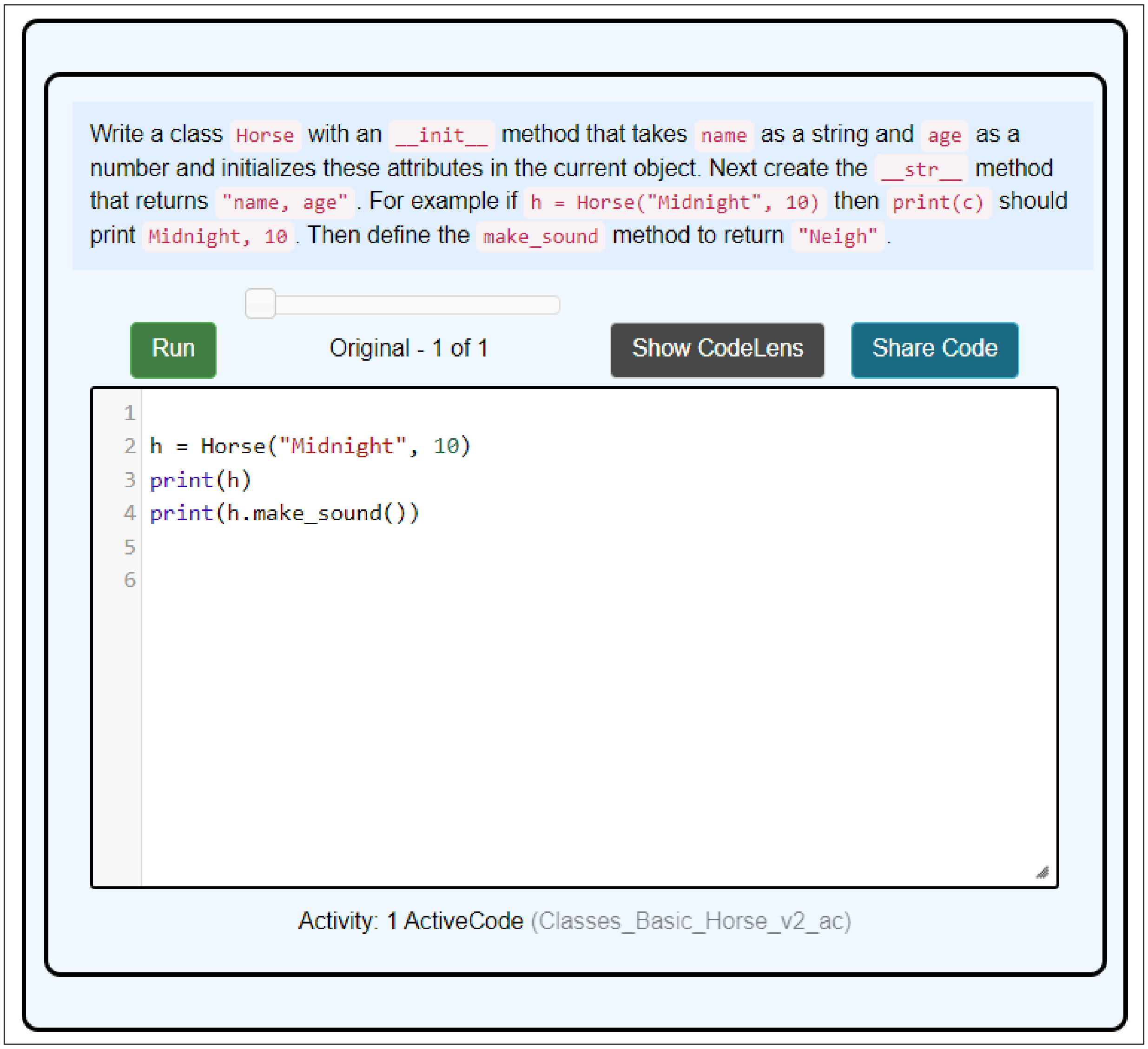


Fix the class `Rectangle` with an `__init__` method that takes a `width` and `height` as numbers and initializes attributes with the same name in the current object. Then create a `__str__` method that returns `"width, height"` as a string. Next create an `total_area` method that takes a number of rectangles, `num` and returns `num` times `width` times `height`. For example, if `rec = Rectangle(15, 2)` and `rec.area(3)` is executed, it should print 90 (3 * 15 * 2).

```python

class Rectangle:
    def __init__(width, height):
        my.width = width
        my.height = height

    def __str__(self):
        return width + ", " + height

    def total_area(self, num):
        return self.num * self.width * self.height

rec = Rectangle(15,2)
print(rec)
print(rec.total_area(3))

```

Activity: 1 ActiveCode (Classes_Basic_Rectangle_ac_fix_v2)

Figure 74: *class-tog* - Post-posttest task for defining a rectangle class and string-conversion method

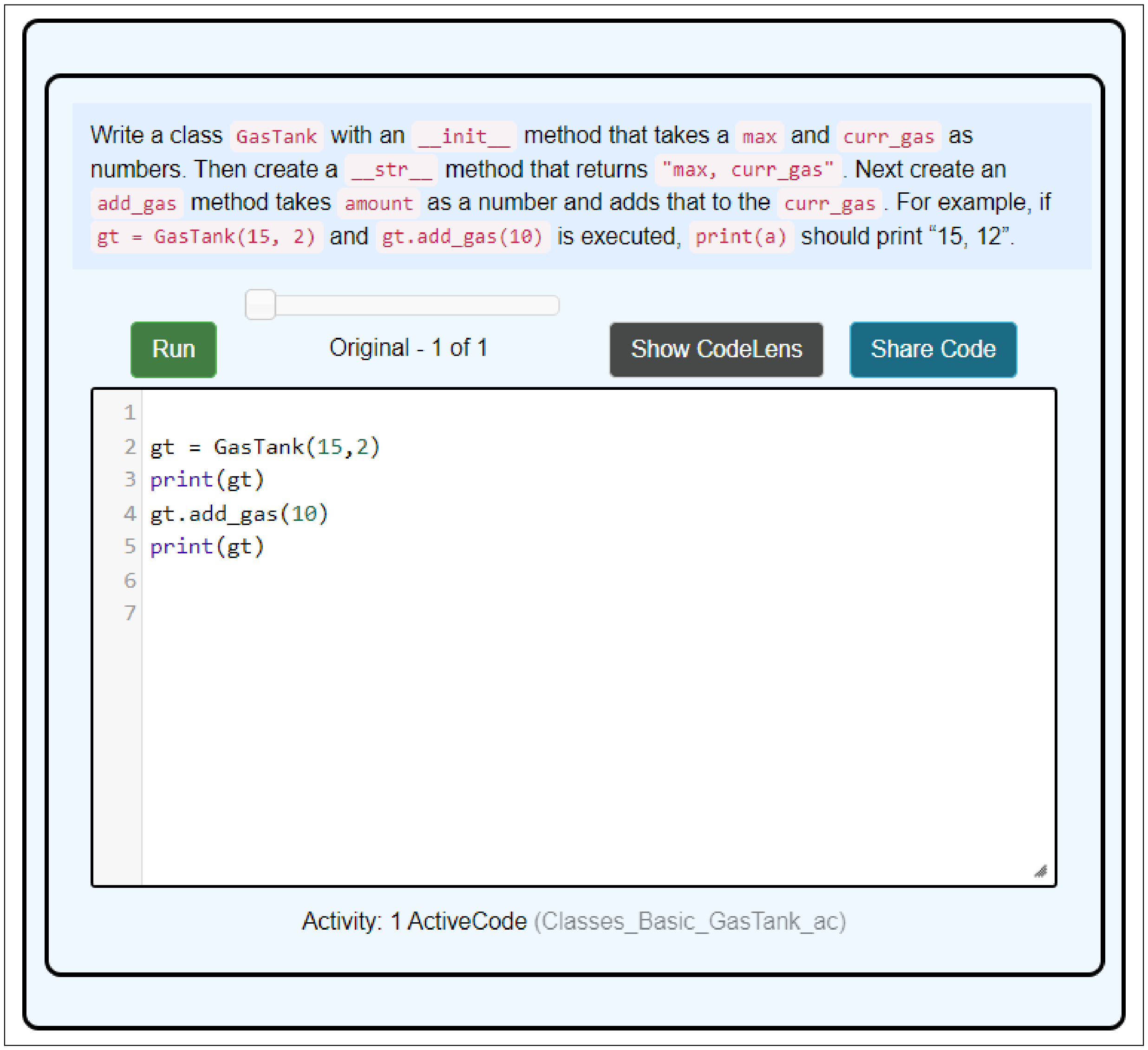


Write a class `Horse` with an `__init__` method that takes `name` as a string and `age` as a number and initializes these attributes in the current object. Next create the `__str__` method that returns `"name, age"`. For example if `h = Horse("Midnight", 10)` then `print(c)` should print `Midnight, 10`. Then define the `make_sound` method to return `"Neigh"`.

```

h = Horse("Midnight", 10)
print(h)
print(h.make_sound())


```

Activity: 1 ActiveCode (Classes_Basic_Horse_v2_ac)

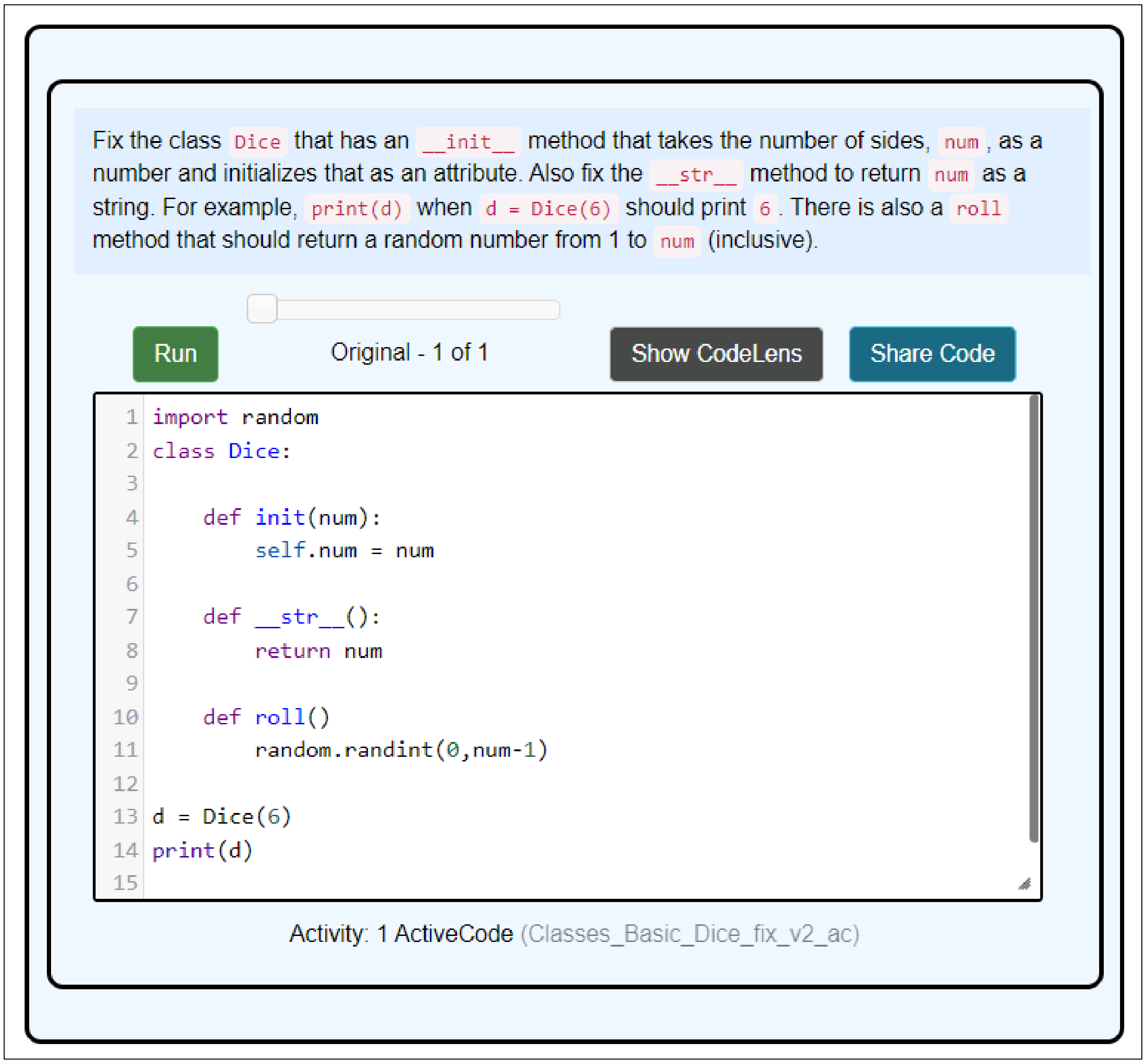
Figure 75: *class-tog* - Post-posttest task for defining a horse class and a method for making sound


Write a class `GasTank` with an `__init__` method that takes a `max` and `curr_gas` as numbers. Then create a `__str__` method that returns `"max, curr_gas"`. Next create an `add_gas` method takes `amount` as a number and adds that to the `curr_gas`. For example, if `gt = GasTank(15, 2)` and `gt.add_gas(10)` is executed, `print(a)` should print "15, 12".

```

gt = GasTank(15,2)
print(gt)
gt.add_gas(10)
print(gt)


```

Activity: 1 ActiveCode (Classes_Basic_GasTank_ac)

Figure 76: ***class-tog*** **- Posttest task for defining a gas tank class and a method for adding gas**



Fix the class `Dice` that has an `__init__` method that takes the number of sides, `num`, as a number and initializes that as an attribute. Also fix the `__str__` method to return `num` as a string. For example, `print(d)` when `d = Dice(6)` should print `6`. There is also a `roll` method that should return a random number from 1 to `num` (inclusive).

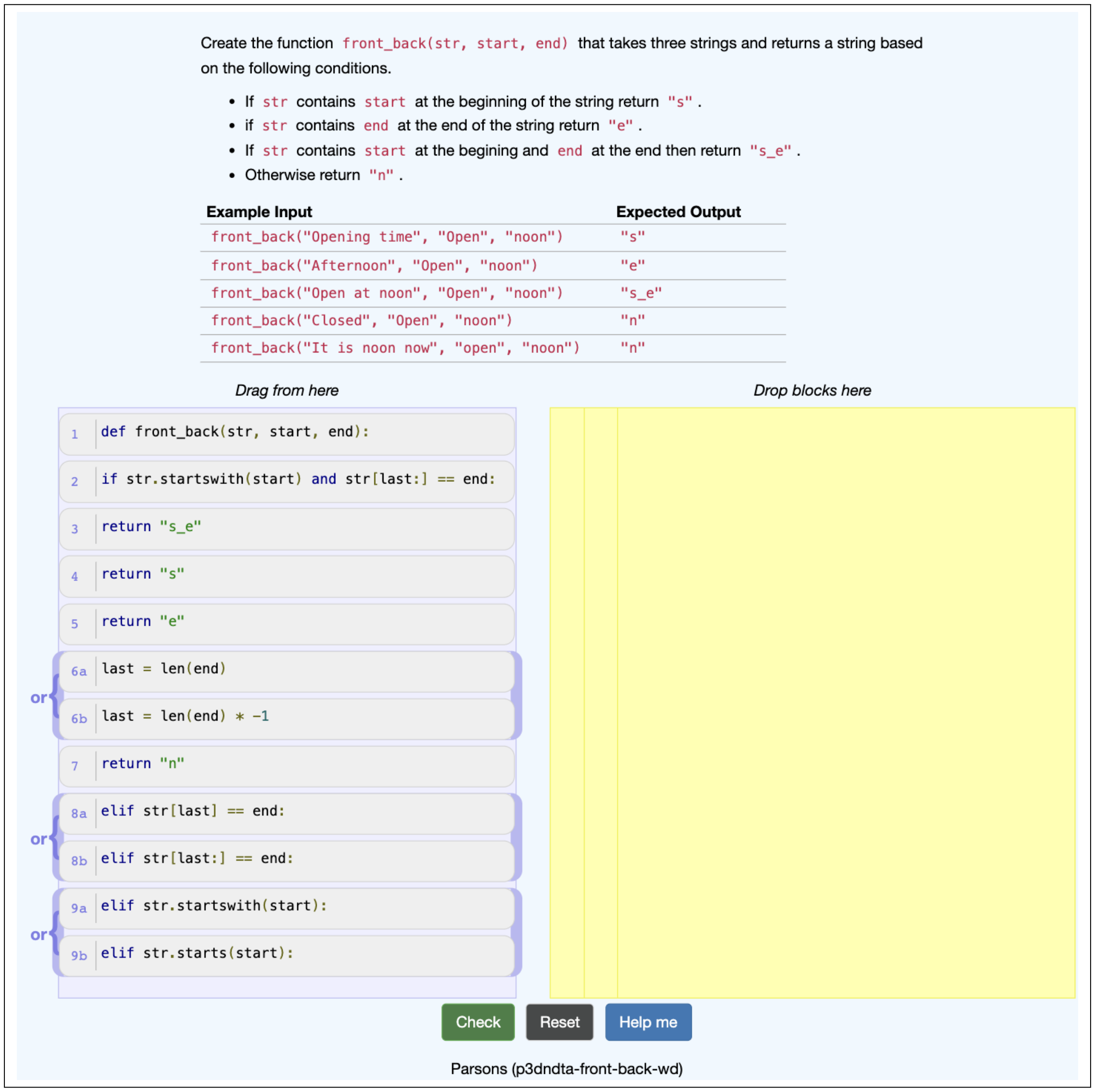
```
import random
class Dice:

    def init(num):
        self.num = num

    def __str__():
        return num

    def roll()
        random.randint(0,num-1)

d = Dice(6)
print(d)

```

Activity: 1 ActiveCode (Classes_Basic_Dice_fix_v2_ac)

Figure 77: *class-tog* - Posttest task for defining a dice class with a given number of sides and a roll method

A.8 *p3dnd*

The *p3dnd* study is similar to *p3pt* but is intended to have harder practice and posttest problems. The two conditions are Parsons problems with and without distractors.



Create the function `front_back(str, start, end)` that takes three strings and returns a string based on the following conditions.

- If `str` contains `start` at the beginning of the string return `"s"` .
- if `str` contains `end` at the end of the string return `"e"` .
- If `str` contains `start` at the begining and `end` at the end then return `"s_e"` .
- Otherwise return `"n"` .

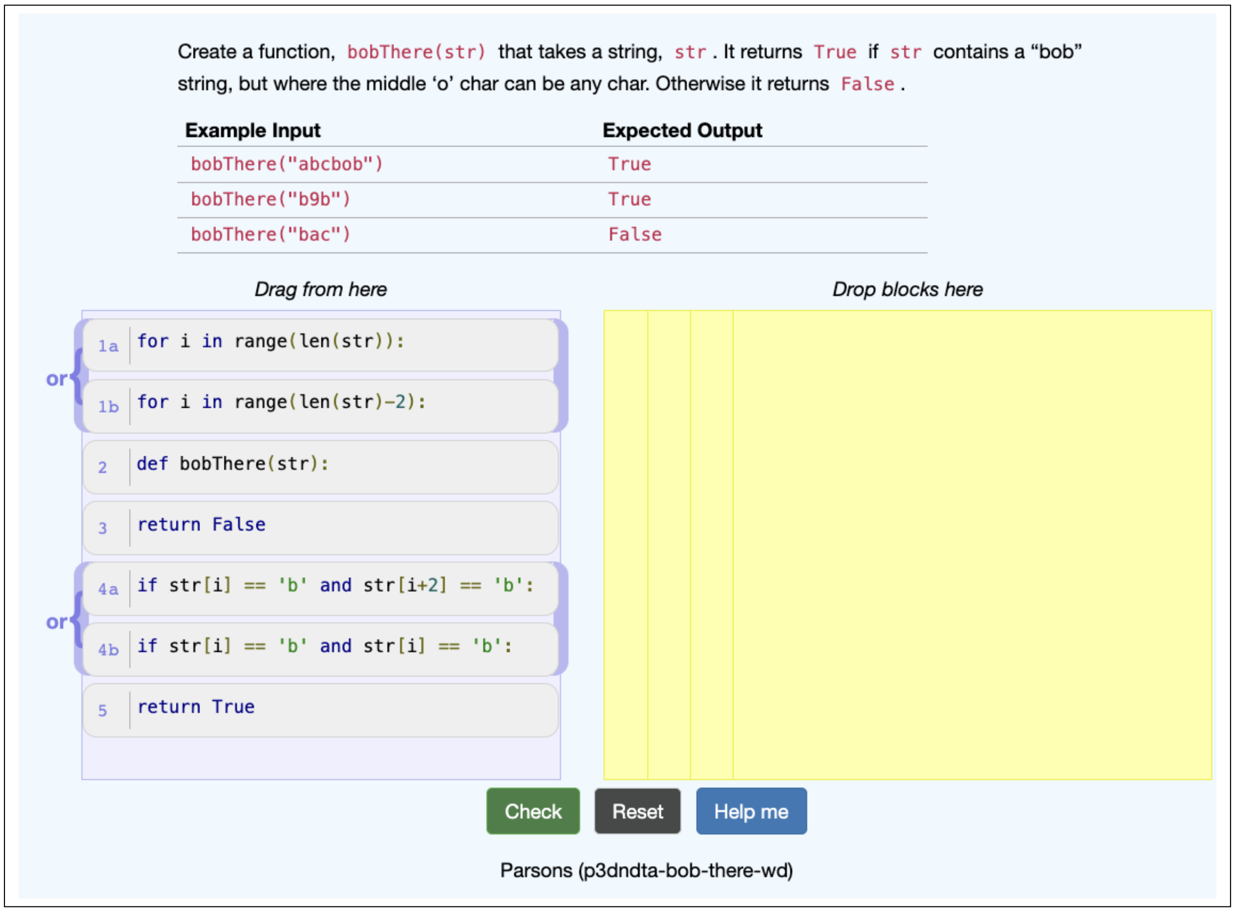
Example Input	Expected Output
`front_back("Opening time", "Open", "noon")`	`"s"`
`front_back("Afternoon", "Open", "noon")`	`"e"`
`front_back("Open at noon", "Open", "noon")`	`"s_e"`
`front_back("Closed", "Open", "noon")`	`"n"`
`front_back("It is noon now", "open", "noon")`	`"n"`

Drag from here

```
1   def front_back(str, start, end):
2   if str.startswith(start) and str[last:] == end:
3   return "s_e"
4   return "s"
5   return "e"
6a  last = len(end)
6b  last = len(end) * -1
7   return "n"
8a  elif str[last] == end:
8b  elif str[last:] == end:
9a  elif str.startswith(start):
9b  elif str.starts(start):
```

Parsons (p3dndta-front-back-wd)

Figure 78: *p3dnd* - return a character based on what is at the beginning and end of a string (with distractors)



Create a function, `bobThere(str)` that takes a string, `str`. It returns `True` if `str` contains a “bob” string, but where the middle ‘o’ char can be any char. Otherwise it returns `False`.

Example Input	Expected Output
`bobThere("abcbob")`	`True`
`bobThere("b9b")`	`True`
`bobThere("bac")`	`False`

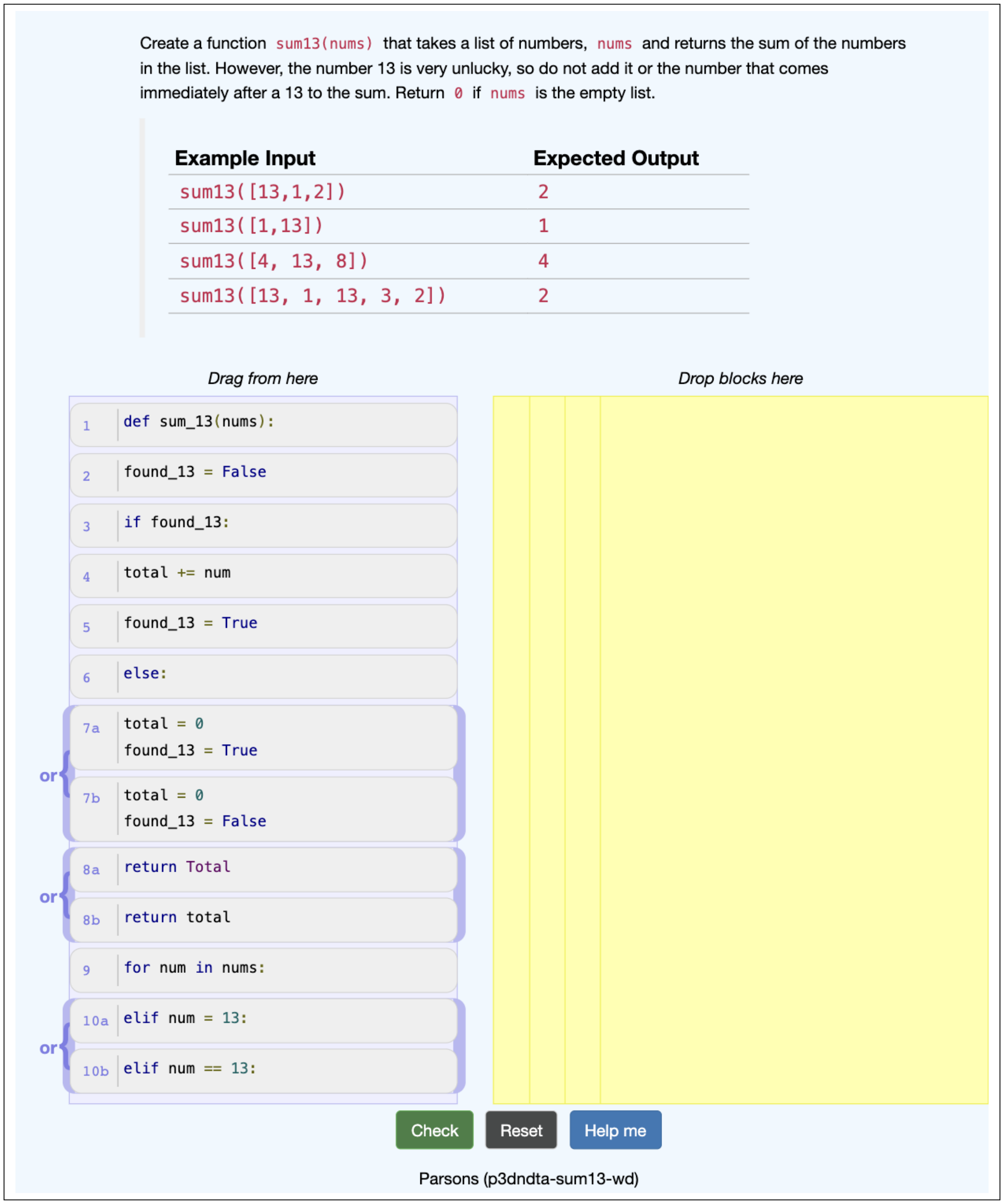
Drag from here

```
1a  for i in range(len(str)):
or
1b  for i in range(len(str)-2):
2   def bobThere(str):
3   return False
4a  if str[i] == 'b' and str[i+2] == 'b':
or
4b  if str[i] == 'b' and str[i] == 'b':
5   return True
```

Drop blocks here

Parsons (p3dndta-bob-there-wd)

Figure 79: *p3dnd* - return true if a string contains "b*b" where ’*’ can be any character (with distractors)



Create a function `sum13(nums)` that takes a list of numbers, `nums` and returns the sum of the numbers in the list. However, the number 13 is very unlucky, so do not add it or the number that comes immediately after a 13 to the sum. Return `0` if `nums` is the empty list.

Example Input	Expected Output
`sum13([13,1,2])`	2
`sum13([1,13])`	1
`sum13([4, 13, 8])`	4
`sum13([13, 1, 13, 3, 2])`	2

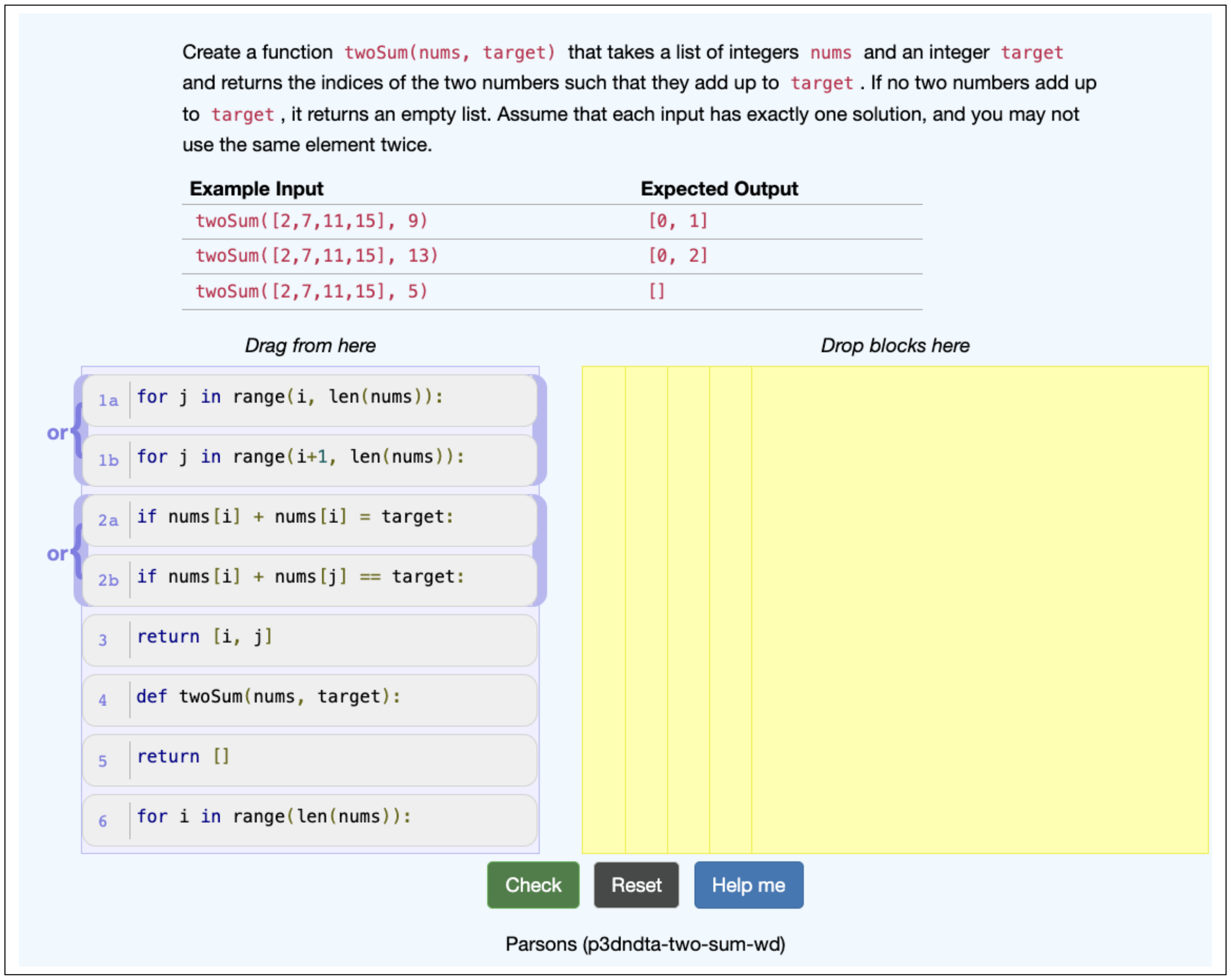
Drag from here

```
1    def sum_13(nums):
2    found_13 = False
3    if found_13:
4    total += num
5    found_13 = True
6    else:
7a   total = 0
     found_13 = True
  or
7b   total = 0
     found_13 = False
8a   return Total
  or
8b   return total
9    for num in nums:
10a  elif num = 13:
  or
10b  elif num == 13:
```

Drop blocks here

Parsons (p3dndta-sum13-wd)

Figure 80: *p3dnd* - return the sum of a list of numbers, but ignore 13 and any number after a 13 (with distractors)



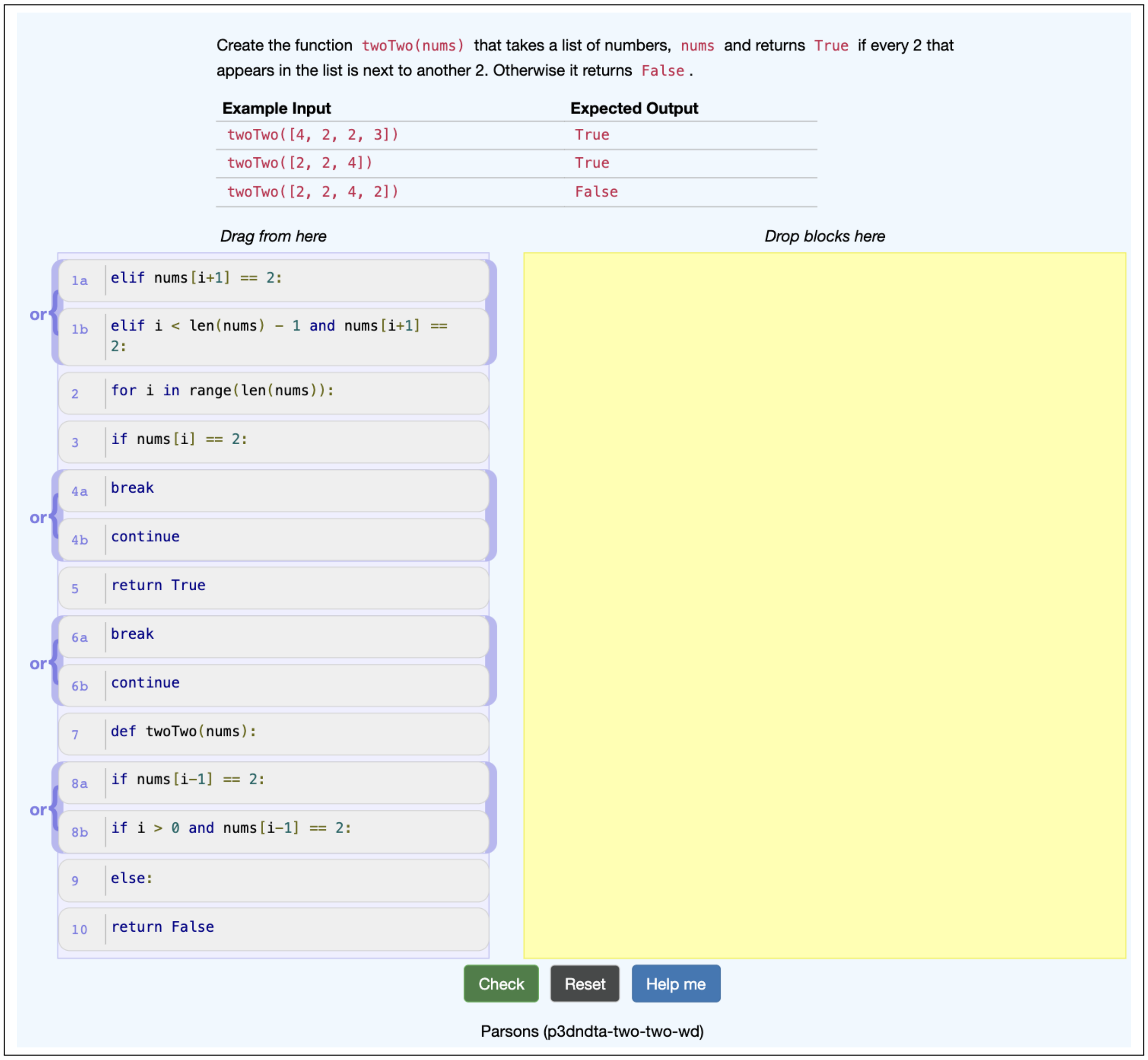
Create a function `twoSum(nums, target)` that takes a list of integers `nums` and an integer `target` and returns the indices of the two numbers such that they add up to `target` . If no two numbers add up to `target` , it returns an empty list. Assume that each input has exactly one solution, and you may not use the same element twice.

Example Input	Expected Output
`twoSum([2,7,11,15], 9)`	`[0, 1]`
`twoSum([2,7,11,15], 13)`	`[0, 2]`
`twoSum([2,7,11,15], 5)`	`[]`

```
1a for j in range(i, len(nums)):
or
1b for j in range(i+1, len(nums)):
2a if nums[i] + nums[i] = target:
or
2b if nums[i] + nums[j] == target:
3 return [i, j]
4 def twoSum(nums, target):
5 return []
6 for i in range(len(nums)):
```

Parsons (p3dndta-two-sum-wd)

Figure 81: *p3dnd* - returns the indices of two numbers in a list that add up to a passed target value (with distractors)


Create the function `twoTwo(nums)` that takes a list of numbers, `nums` and returns `True` if every 2 that appears in the list is next to another 2. Otherwise it returns `False` .

Example Input	Expected Output
`twoTwo([4, 2, 2, 3])`	`True`
`twoTwo([2, 2, 4])`	`True`
`twoTwo([2, 2, 4, 2])`	`False`

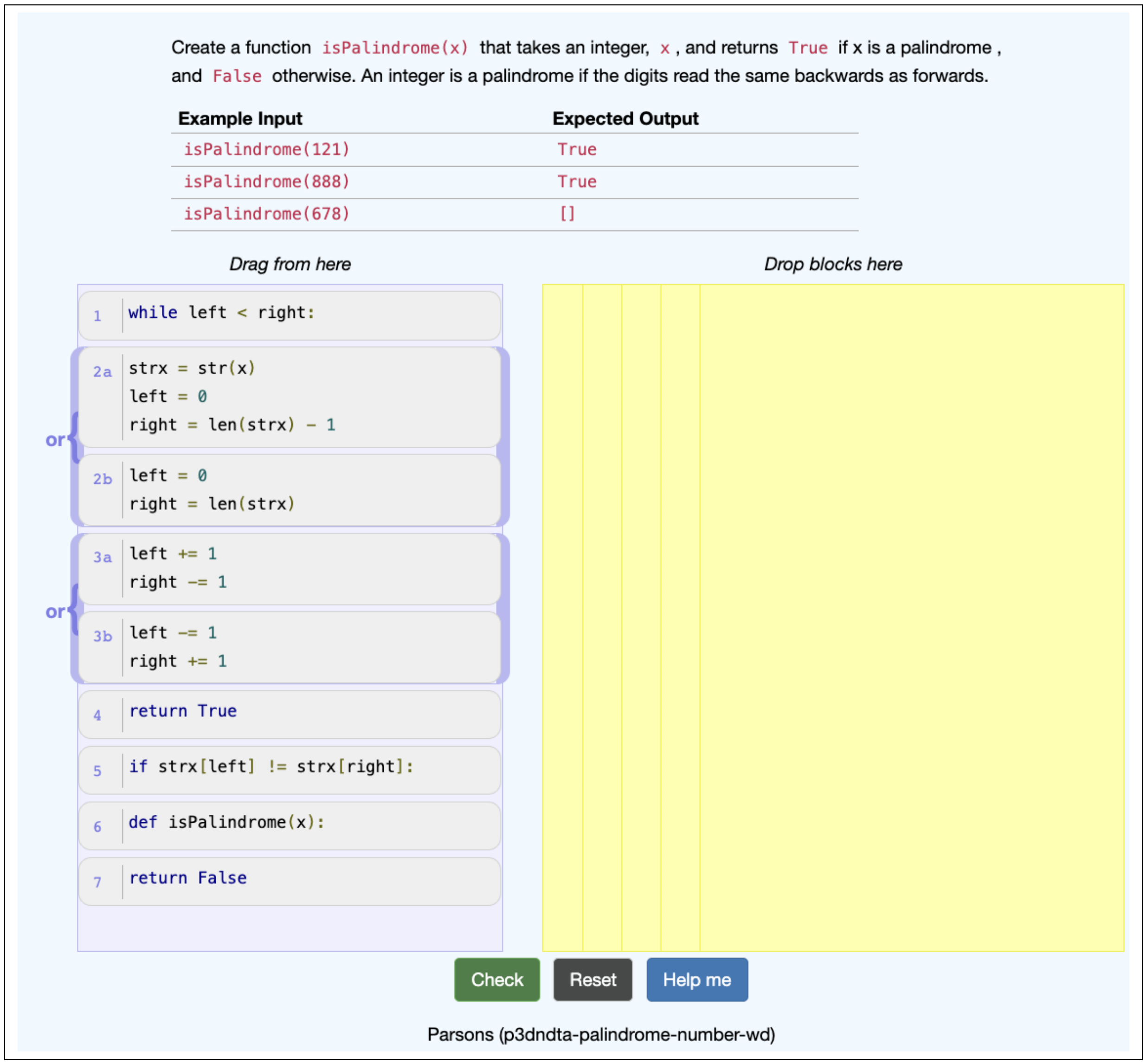
Drag from here

```
1a elif nums[i+1] == 2:
or
1b elif i < len(nums) - 1 and nums[i+1] ==
   2:
2  for i in range(len(nums)):
3  if nums[i] == 2:
4a break
or
4b continue
5  return True
6a break
or
6b continue
7  def twoTwo(nums):
8a if nums[i-1] == 2:
or
8b if i > 0 and nums[i-1] == 2:
9  else:
10 return False
```

Drop blocks here

Parsons (p3dndta-two-two-wd)

Figure 82: ***p3dnd*** **- return true if every two in a list of numbers is next to another two (with distractors)**


Create a function `isPalindrome(x)` that takes an integer, `x`, and returns `True` if x is a palindrome, and `False` otherwise. An integer is a palindrome if the digits read the same backwards as forwards.

Example Input	Expected Output
`isPalindrome(121)`	`True`
`isPalindrome(888)`	`True`
`isPalindrome(678)`	`[]`

Drag from here

```
1   while left < right:
```

or

```
2a  strx = str(x)
    left = 0
    right = len(strx) - 1
```

```
2b  left = 0
    right = len(strx)
```

or

```
3a  left += 1
    right -= 1
```

```
3b  left -= 1
    right += 1
```

```
4   return True
```

```
5   if strx[left] != strx[right]:
```

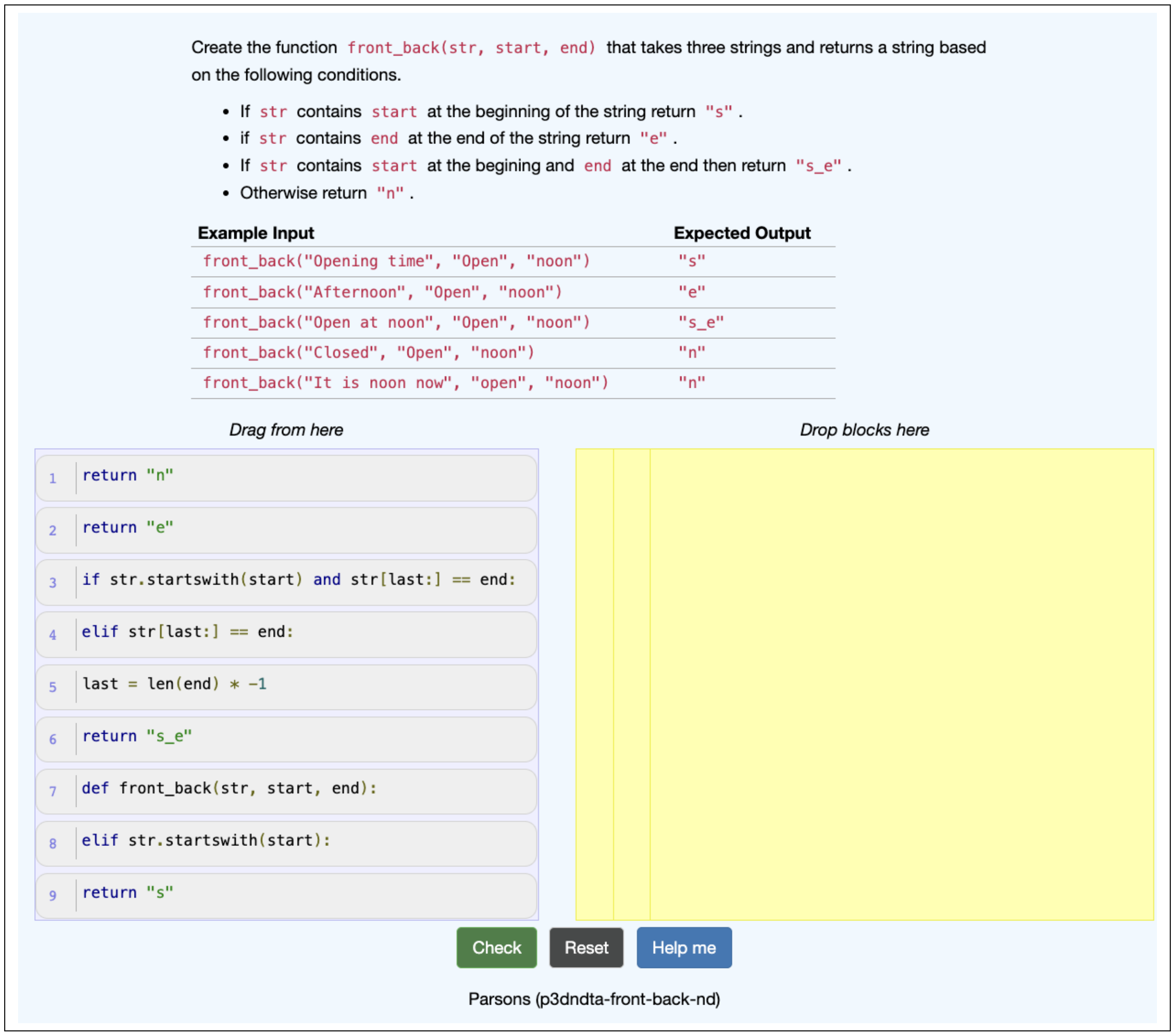
```
6   def isPalindrome(x):
```

```
7   return False
```

Drop blocks here

Parsons (p3dndta-palindrome-number-wd)

Figure 83: p3dnd - returns true if the digits in a number are a palindrome (with distractors)



Create the function `front_back(str, start, end)` that takes three strings and returns a string based on the following conditions.

- If `str` contains `start` at the beginning of the string return `"s"` .
- if `str` contains `end` at the end of the string return `"e"` .
- If `str` contains `start` at the begining and `end` at the end then return `"s_e"` .
- Otherwise return `"n"` .

Example Input	Expected Output
`front_back("Opening time", "Open", "noon")`	`"s"`
`front_back("Afternoon", "Open", "noon")`	`"e"`
`front_back("Open at noon", "Open", "noon")`	`"s_e"`
`front_back("Closed", "Open", "noon")`	`"n"`
`front_back("It is noon now", "open", "noon")`	`"n"`

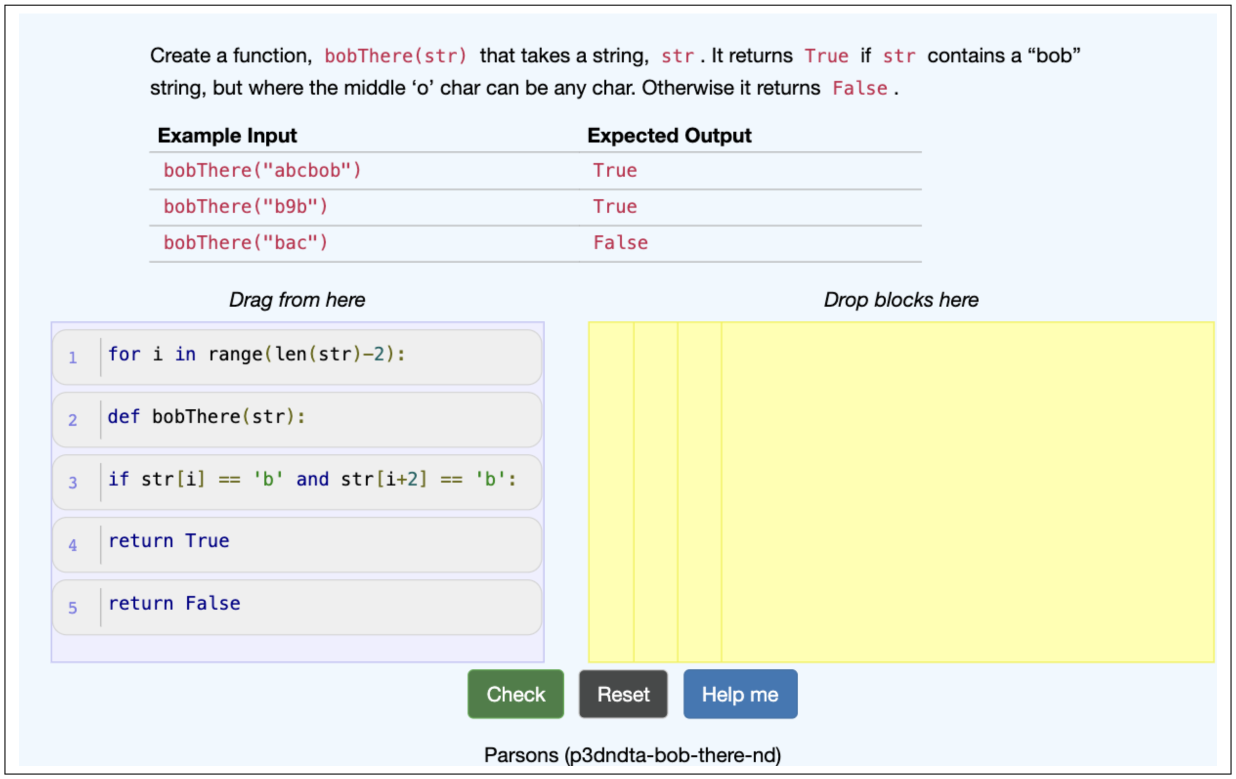
Drag from here

Drop blocks here

```
return "n"
```
```
return "e"
```
```
if str.startswith(start) and str[last:] == end:
```
```
elif str[last:] == end:
```
```
last = len(end) * -1
```
```
return "s_e"
```
```
def front_back(str, start, end):
```
```
elif str.startswith(start):
```
```
return "s"
```

Parsons (p3dndta-front-back-nd)

Figure 84: *p3dnd* - return a character based on what is at the beginning and end of a string



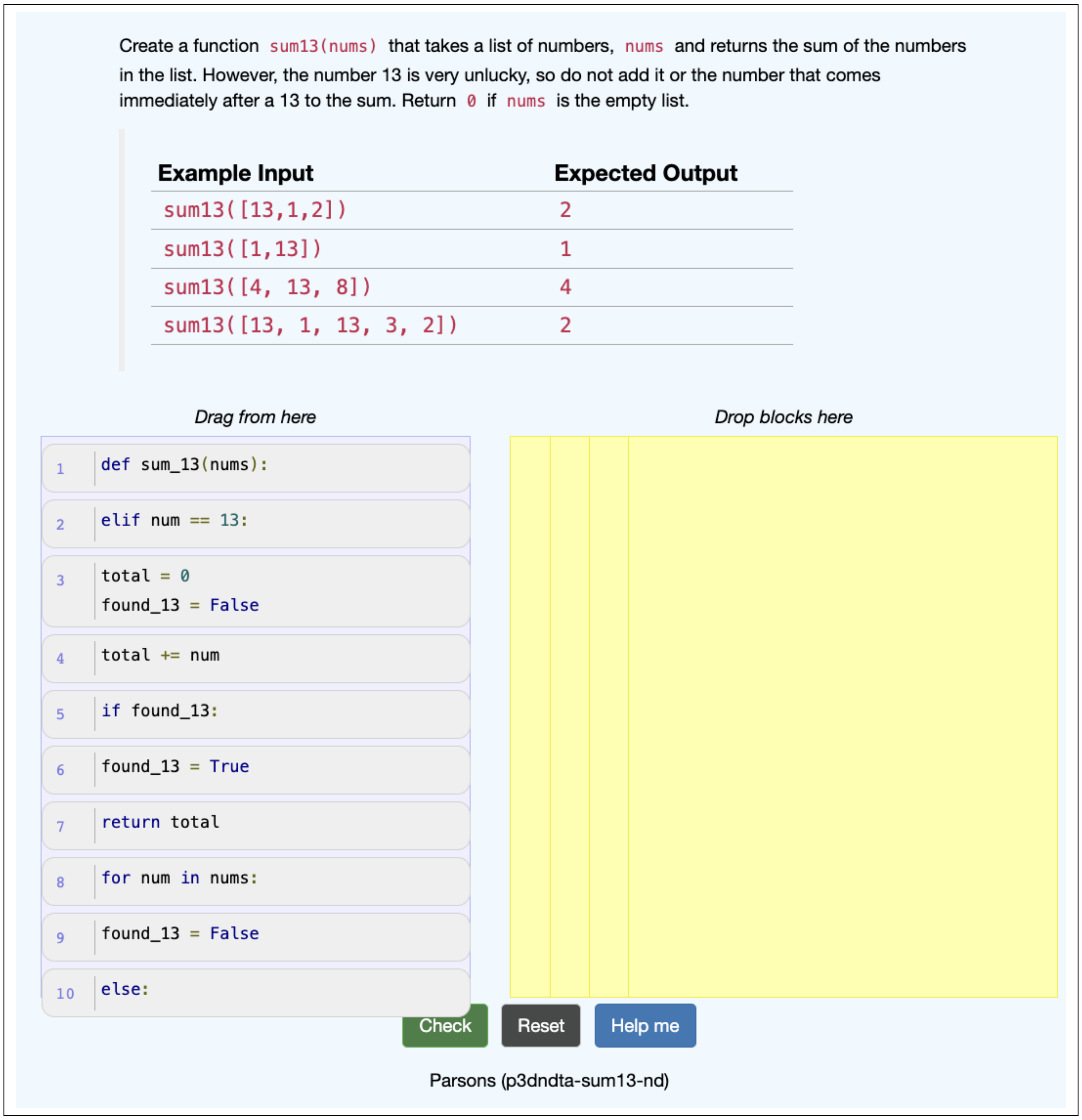
Create a function, `bobThere(str)` that takes a string, `str`. It returns `True` if `str` contains a "bob" string, but where the middle 'o' char can be any char. Otherwise it returns `False`.

Example Input	Expected Output
`bobThere("abcbob")`	`True`
`bobThere("b9b")`	`True`
`bobThere("bac")`	`False`

```
for i in range(len(str)-2):
```
```
def bobThere(str):
```
```
if str[i] == 'b' and str[i+2] == 'b':
```
```
return True
```
```
return False
```

Parsons (p3dndta-bob-there-nd)

Figure 85: *p3dnd* - return true if a string contains "b*b" where "*" can be any character



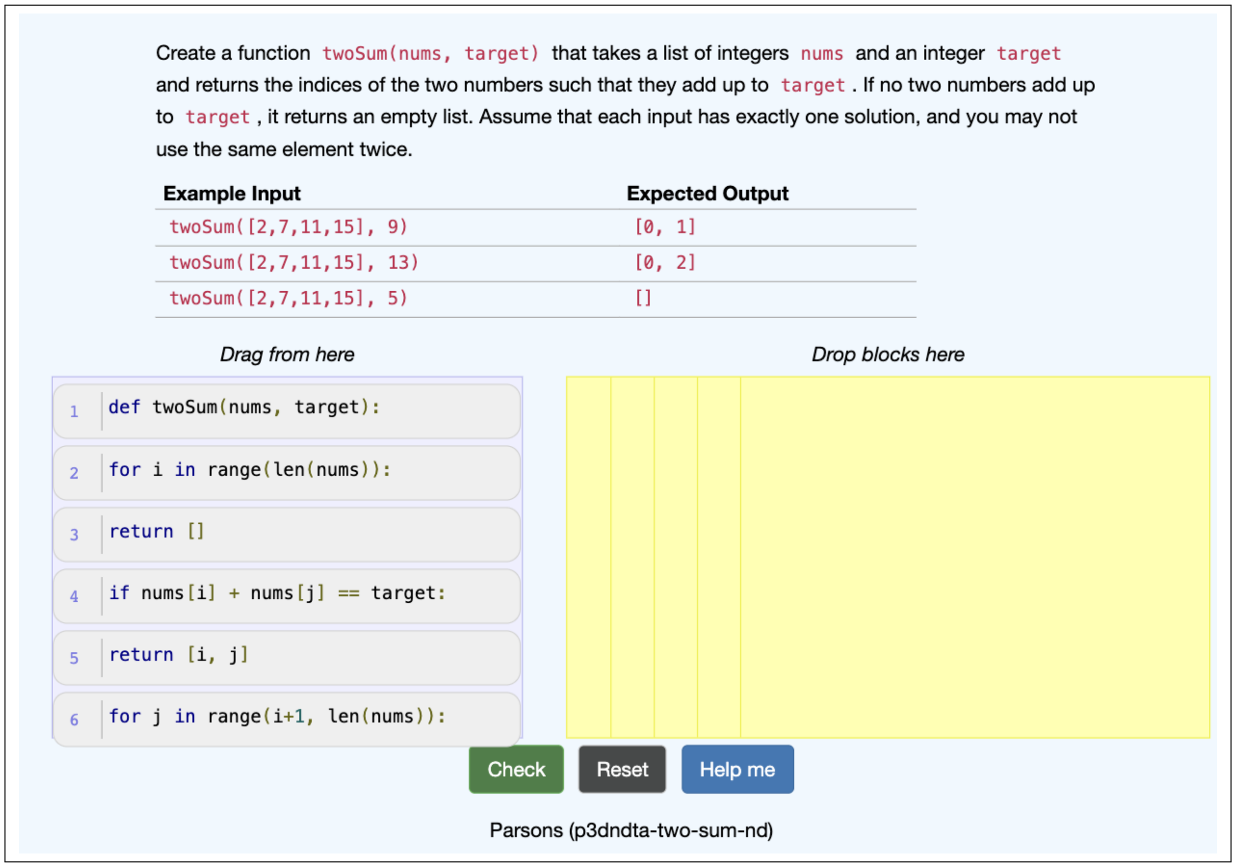
Create a function `sum13(nums)` that takes a list of numbers, `nums` and returns the sum of the numbers in the list. However, the number 13 is very unlucky, so do not add it or the number that comes immediately after a 13 to the sum. Return `0` if `nums` is the empty list.

Example Input	Expected Output
`sum13([13,1,2])`	2
`sum13([1,13])`	1
`sum13([4, 13, 8])`	4
`sum13([13, 1, 13, 3, 2])`	2

```
1  def sum_13(nums):
2  elif num == 13:
3  total = 0
   found_13 = False
4  total += num
5  if found_13:
6  found_13 = True
7  return total
8  for num in nums:
9  found_13 = False
10 else:
```

Parsons (p3dndta-sum13-nd)

Figure 86: *p3dnd* - return a sum of a list of numbers, but ignore 13 and any number after a 13



Create a function `twoSum(nums, target)` that takes a list of integers `nums` and an integer `target` and returns the indices of the two numbers such that they add up to `target`. If no two numbers add up to `target`, it returns an empty list. Assume that each input has exactly one solution, and you may not use the same element twice.

Example Input	Expected Output
`twoSum([2,7,11,15], 9)`	`[0, 1]`
`twoSum([2,7,11,15], 13)`	`[0, 2]`
`twoSum([2,7,11,15], 5)`	`[]`

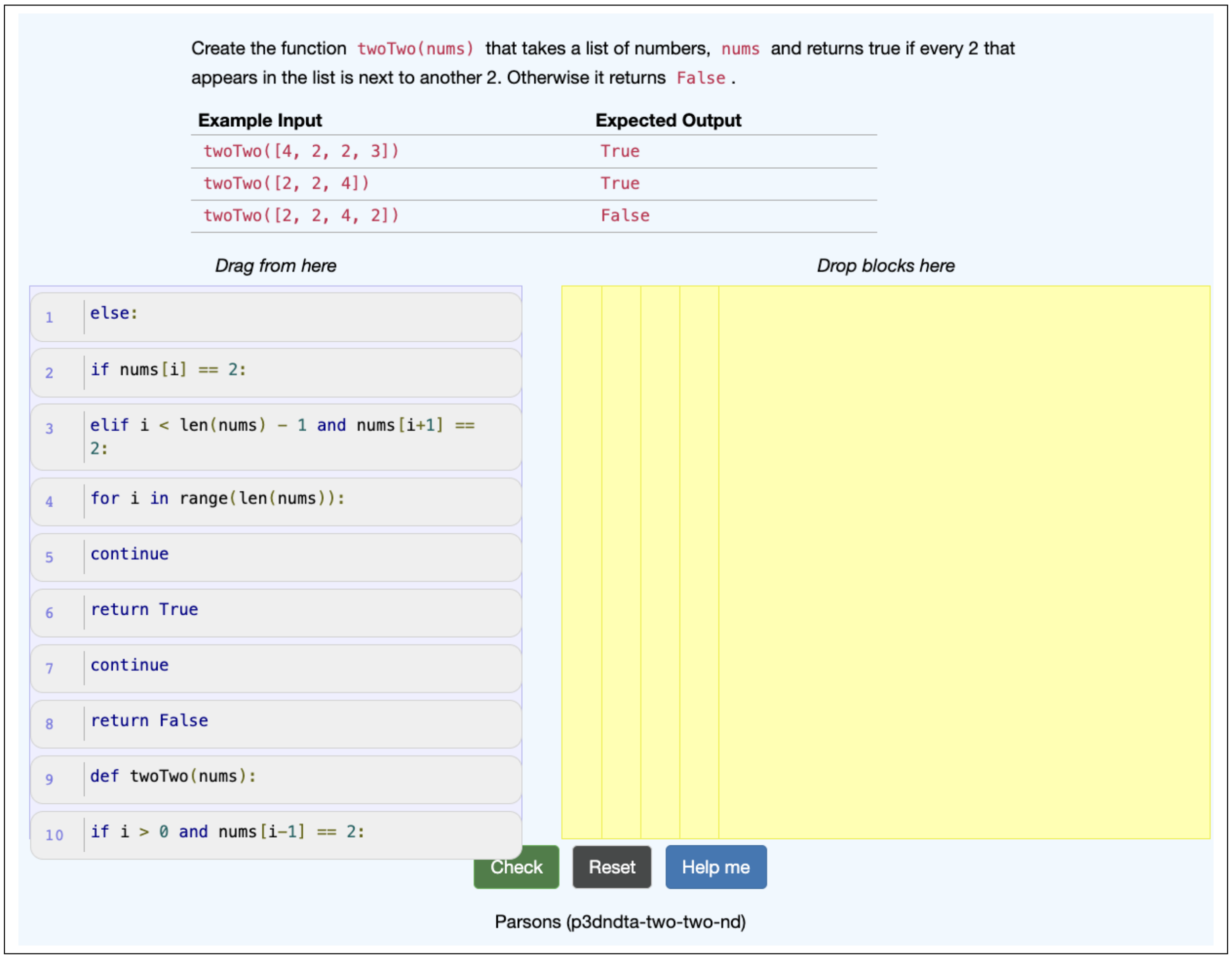
Drag from here

```
1 def twoSum(nums, target):
2 for i in range(len(nums)):
3 return []
4 if nums[i] + nums[j] == target:
5 return [i, j]
6 for j in range(i+1, len(nums)):
```

Drop blocks here

Parsons (p3dndta-two-sum-nd)

Figure 87: ***p3dnd*** **- returns the indices of two numbers in a list that add up to a passed target value**



Create the function `twoTwo(nums)` that takes a list of numbers, `nums` and returns true if every 2 that appears in the list is next to another 2. Otherwise it returns `False` .

Example Input	Expected Output
`twoTwo([4, 2, 2, 3])`	`True`
`twoTwo([2, 2, 4])`	`True`
`twoTwo([2, 2, 4, 2])`	`False`

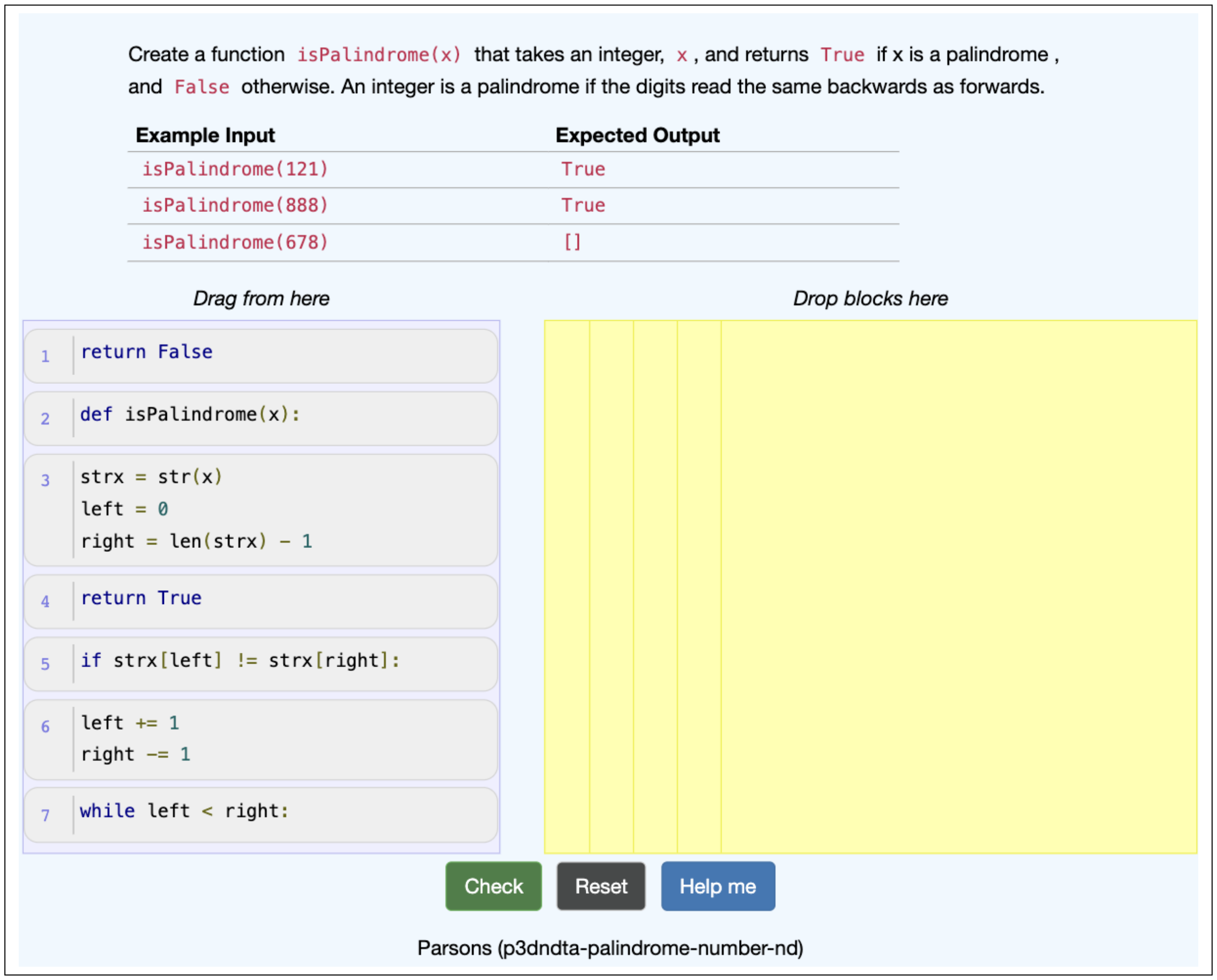
Drag from here

```
else:
```
```
if nums[i] == 2:
```
```
elif i < len(nums) - 1 and nums[i+1] ==
2:
```
```
for i in range(len(nums)):
```
```
continue
```
```
return True
```
```
continue
```
```
return False
```
```
def twoTwo(nums):
```
```
if i > 0 and nums[i-1] == 2:
```

Drop blocks here

Parsons (p3dndta-two-two-nd)

Figure 88: ***p3dnd*** **- return true if every two in a list of numbers is next to another two**


Create a function `isPalindrome(x)` that takes an integer, `x`, and returns `True` if x is a palindrome, and `False` otherwise. An integer is a palindrome if the digits read the same backwards as forwards.

Example Input	Expected Output
`isPalindrome(121)`	`True`
`isPalindrome(888)`	`True`
`isPalindrome(678)`	`[]`

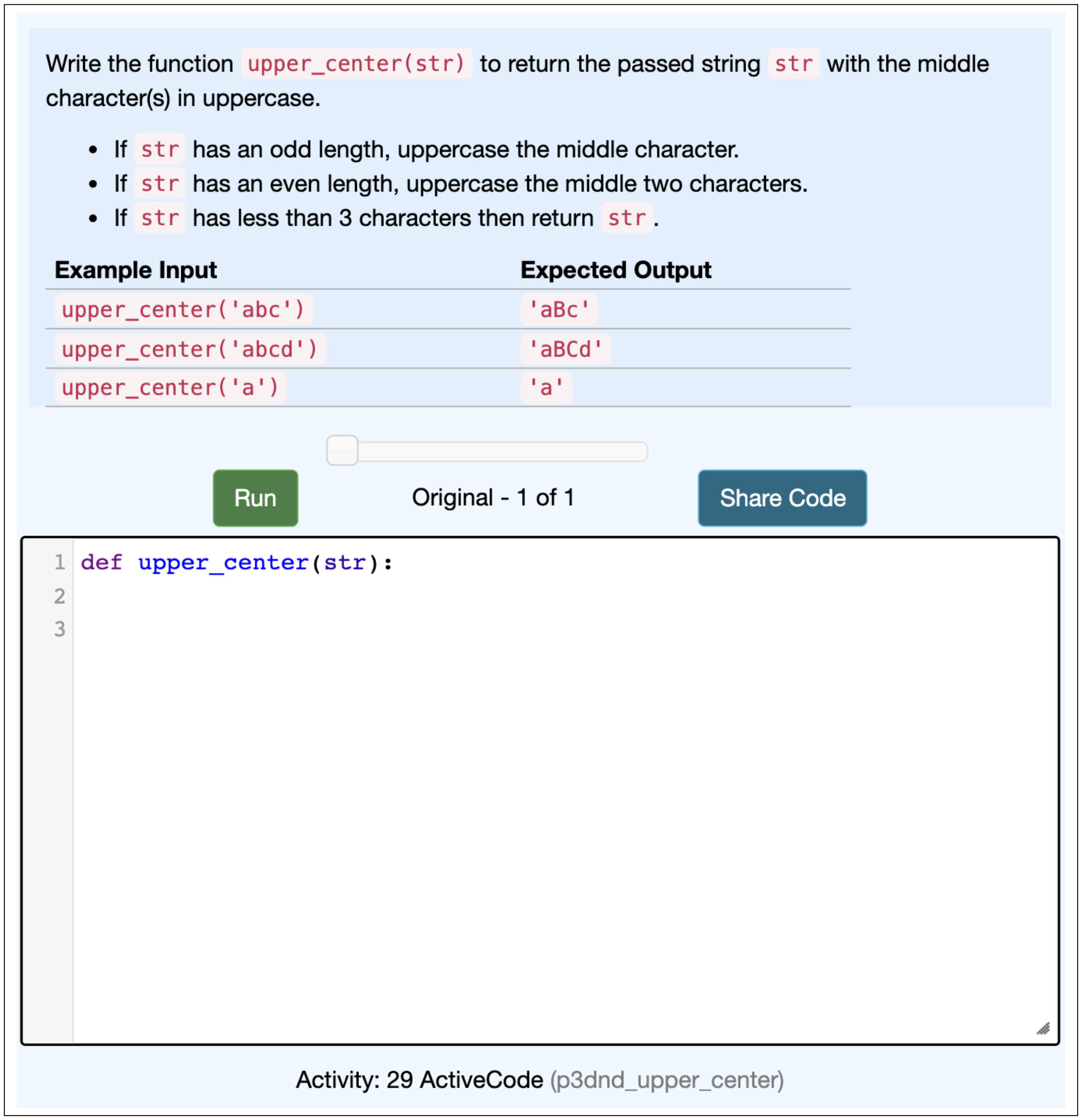
Drag from here

```
return False
```
```
def isPalindrome(x):
```
```
strx = str(x)
left = 0
right = len(strx) - 1
```
```
return True
```
```
if strx[left] != strx[right]:
```
```
left += 1
right -= 1
```
```
while left < right:
```

Drop blocks here

Parsons (p3dndta-palindrome-number-nd)

Figure 89: *p3dnd* - returns true if the digits in a number are a palindrome


Write the function `upper_center(str)` to return the passed string `str` with the middle character(s) in uppercase.

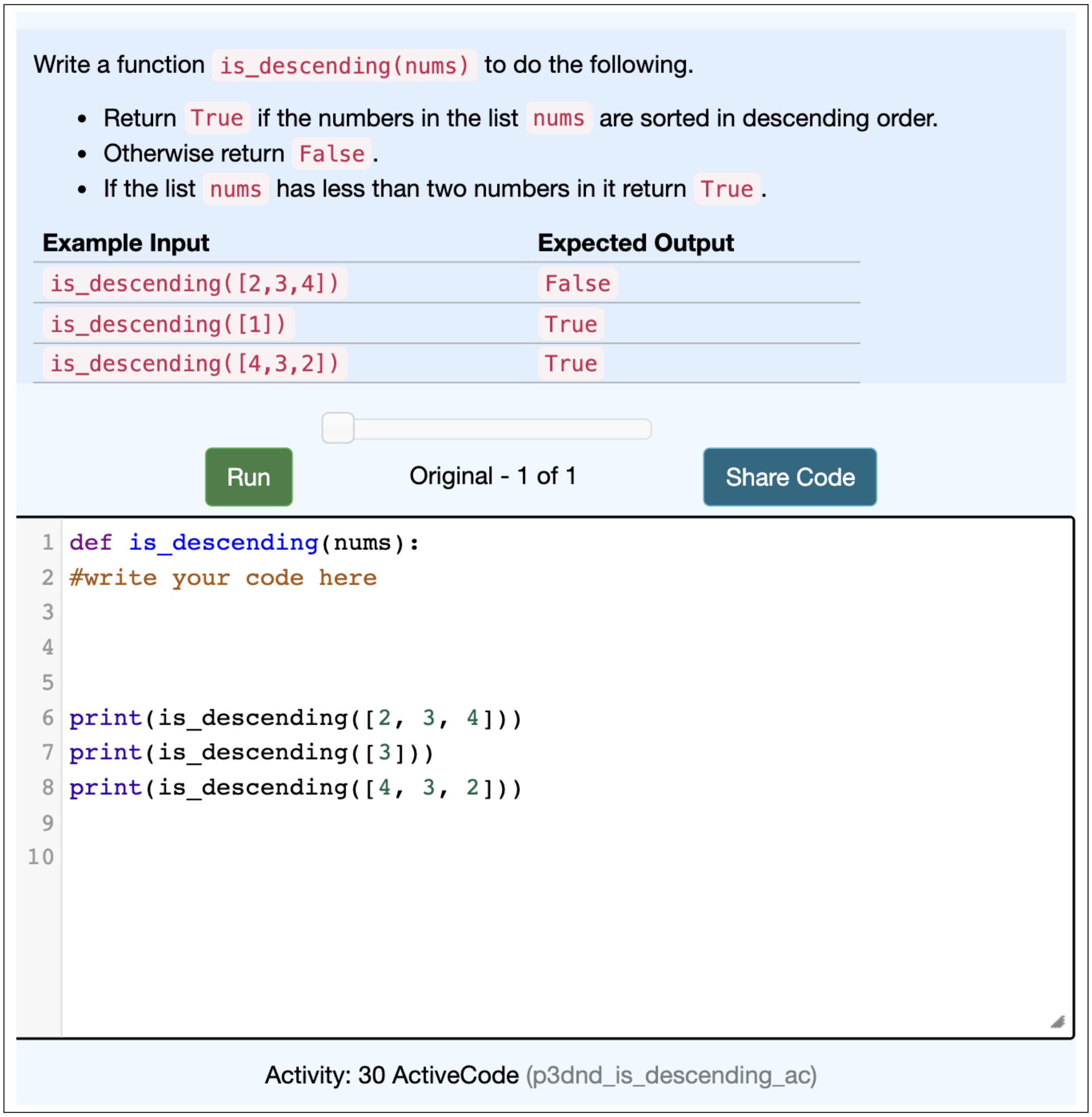
- If `str` has an odd length, uppercase the middle character.
- If `str` has an even length, uppercase the middle two characters.
- If `str` has less than 3 characters then return `str`.

Example Input	Expected Output
`upper_center('abc')`	`'aBc'`
`upper_center('abcd')`	`'aBCd'`
`upper_center('a')`	`'a'`

```python
def upper_center(str):


```

Activity: 29 ActiveCode (p3dnd_upper_center)

Figure 90: ***p3dnd*** **- return a string with the center characters in uppercase or just the string if the length is less than 3**



Write a function `is_descending(nums)` to do the following.

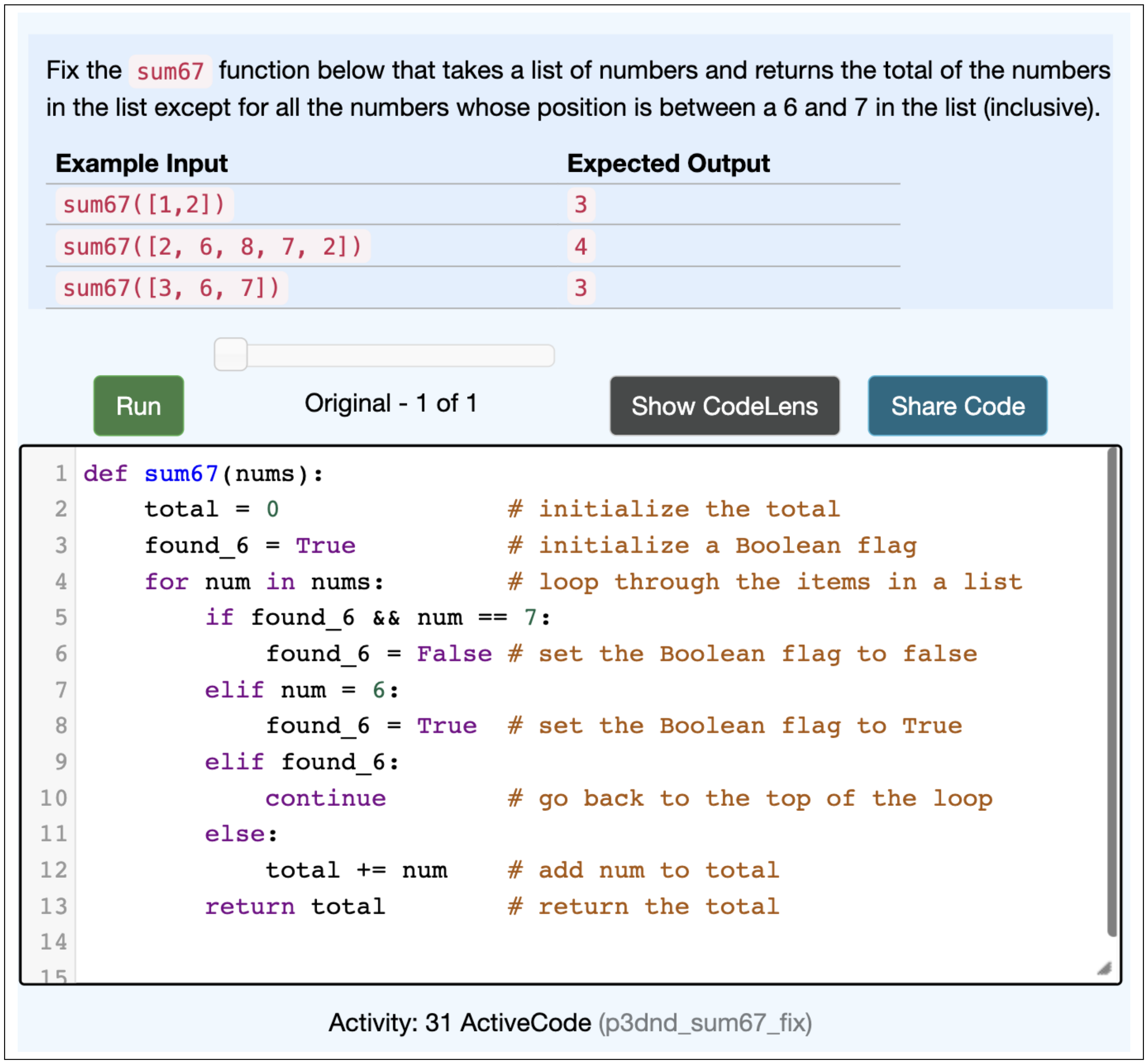
- Return `True` if the numbers in the list `nums` are sorted in descending order.
- Otherwise return `False`.
- If the list `nums` has less than two numbers in it return `True`.

Example Input	Expected Output
`is_descending([2,3,4])`	`False`
`is_descending([1])`	`True`
`is_descending([4,3,2])`	`True`

```
def is_descending(nums):
#write your code here



print(is_descending([2, 3, 4]))
print(is_descending([3]))
print(is_descending([4, 3, 2]))
```

Activity: 30 ActiveCode (p3dnd_is_descending_ac)

Figure 91: *p3dnd* - return true if the numbers in a list are sorted in descending order


Fix the `sum67` function below that takes a list of numbers and returns the total of the numbers in the list except for all the numbers whose position is between a 6 and 7 in the list (inclusive).

Example Input	Expected Output
`sum67([1,2])`	`3`
`sum67([2, 6, 8, 7, 2])`	`4`
`sum67([3, 6, 7])`	`3`

```
def sum67(nums):
    total = 0              # initialize the total
    found_6 = True         # initialize a Boolean flag
    for num in nums:       # loop through the items in a list
        if found_6 && num == 7:
            found_6 = False # set the Boolean flag to false
        elif num = 6:
            found_6 = True  # set the Boolean flag to True
        elif found_6:
            continue        # go back to the top of the loop
        else:
            total += num    # add num to total
        return total        # return the total

```

Activity: 31 ActiveCode (p3dnd_sum67_fix)

Figure 92: *p3dnd* - returns the total of all numbers in a list except those inclusively between a 6 and 7