Exploring the Use of Parsons Problems for Learning a New Programming Language

Mansi Shah

Technical Report No. UCB/EECS-2020-88
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-88.html

May 28, 2020

Acknowledgement

This work would not be possible without the support and help of my peers, friends, and mentors. In particular, I'd like to thank:

- My advisor Dan Garcia, for inspiring me to study CS Education, and for all of the support and guidance he's provided me.
- Michael Ball for his guidance and advice, and for his invaluable feedback as my second thesis reader.
- Nate Weinmann for helping me learn about Parsons Problems, and advising me on how to run this study.
- The entire CS10 staff, especially Maxson Yang, Niki Zarkub, Murtaza Ali, and Bryant Bettencourt, for supporting this work in CS10.
- Lara McConnaughey, Suraj Rampure, and Andrew Chang for all of their advice and support, and for their help editing my thesis.
- My family for their constant support, love, and encouragement.

UNIVERSITY OF CALIFORNIA, BERKELEY

MASTER'S PROJECT REPORT

Exploring the Use of Parsons Problems for Learning a New Programming Language

---

Author:
Mansi SHAH

Advisor:
Teaching Professor Dan GARCIA

A project submitted in fulfillment of the requirements for the degree of Master of Science

in the

Department of Electrical Engineering and Computer Sciences

May 29, 2020

Abstract

Parsons Problems are programming puzzles where students rearrange code blocks to construct a program. Parsons Problems have been found to be just as effective for learning computer science as problems that involve writing or fixing code, while taking students significantly less time to complete. Additionally, students find these problems more engaging than traditional problems. In this study, we explore the use of Parsons Problems for the specific learning goal of learning a new programming language. In particular, we study whether these problems maintain their benefits in this context, and how they should be designed to support this learning goal.

This study was carried out during the Spring 2020 semester of CS10, an introductory computer science course for nonmajors at UC Berkeley, to help students who had programmed for ten weeks using a block-based programming language, Snap*!*, to transition to programming in Python. Out of the 87 students who participated in this study, 83% found Parsons Problems helpful for transitioning to Python, 69% found them more enjoyable than writing freeform Python code, and 68% found them more time-efficient than writing freeform Python code; these trends were even stronger for students with no prior Python experience. Ultimately, we found that Parsons Problems are effective for learning syntax: students found them helpful, engaging, and more time-efficient, making them an especially promising learning exercise for introductory computer science courses.

Acknowledgements

This work would not be possible without the support and help of my peers, friends, and mentors. In particular, I'd like to thank:

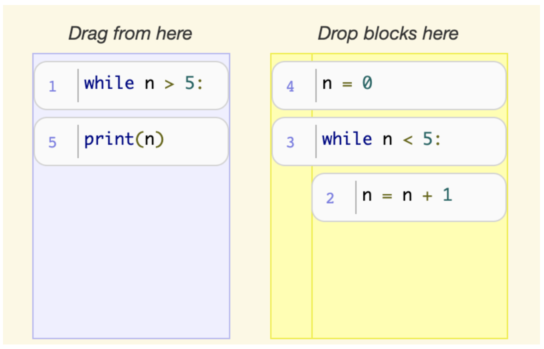
- My advisor Dan Garcia, for inspiring me to study CS Education, and for all of the support and guidance he's provided me across my five years at UC Berkeley.
- Michael Ball for his guidance and advice, and for his invaluable feedback as my second thesis reader.
- Nate Weinmann for helping me learn about Parsons Problems, and advising me on how to run this study.
- The entire CS10 staff, especially Maxson Yang, Niki Zarkub, Murtaza Ali, and Bryant Bettencourt, for supporting this work in CS10.
- Lara McConnaughey, Suraj Rampure, and Andrew Chang for their advice and support through this program, and for all of their help editing my thesis.
- My family for their constant support, love, and encouragement.

Contents

1 Introduction

1.1 Parsons Problems

Parsons Problems are programming puzzles in which students drag-and-drop pre-written code snippets to construct a program [18], as shown in Figure 1.1.



```
while n > 5:
print(n)
```

```
n = 0
while n < 5:
    n = n + 1
```

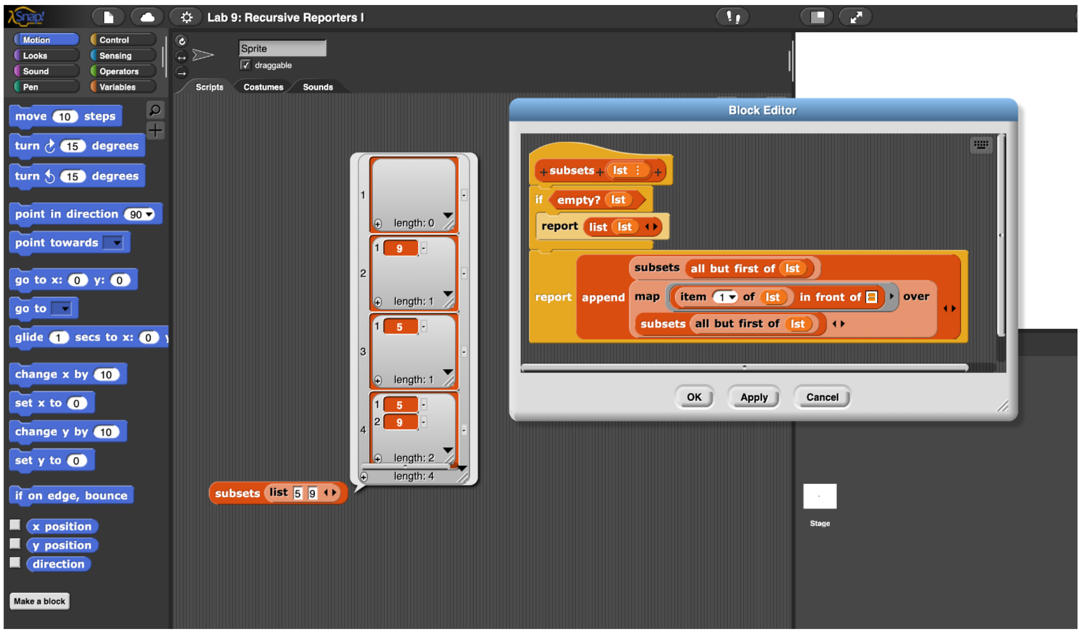
FIGURE 1.1: Sample Parsons Problem in Python. Students drag code snippets from the left pane to the right pane in the correct order. In this specific problem, which is a *2D* Parsons Problem, students also need to drag code snippets to the correct indentation level.

Parsons Problems offer a variety of benefits to learners. Due to their constrained problem space and their use of pre-written code snippets, Parsons Problems impose a lower cognitive load on students than traditional code-writing problems [21]. This also allows students to focus less on syntax and more on problem-solving, which may be especially helpful for novice programmers. Furthermore, because Parsons Problems generally run in a browser-based environment, students do not have to worry about learning to run their code; they can get continuous, rich feedback almost instantly, which is crucial for maintaining student engagement and in introductory computer science courses [2]. Altogether, Parsons Problems are highly effective learning tools: they have been found to increase students' engagement [4, 7, 18] and substantially reduce the overall time students spend on problems, while still being at least as effective for learning computer science as writing freeform code or fixing buggy code [8, 10, 12, 21].

1.2 CS10: The Beauty and Joy of Computing

CS10: Beauty and Joy of Computing, the course in which this study has been conducted, is an introductory computer science course at UC Berkeley for students with no previous coding experience. The course is designed to be extremely accessible to novice programmers and focuses on exposing students to big ideas in computer science, getting students excited about the subject, and teaching them how to program and apply computer science to larger problems [9].

In CS10, students are first introduced to programming using a visual blocks-based programming language called Snap*!* [13], as shown in Figure 1.2. In block-based programming languages, students drag and drop code and construct solutions from a limited set of code blocks, providing similar features and benefits as Parsons Problems. In fact, using block-based programming languages in introductory computer science courses has been shown to improve programming efficiency [19] and learning gains [20]. Through programming in Snap*!*, students develop strong foundations in computational thinking, and are able to solve problems involving recursion, higher order functions, and other core computer science concepts.



```
subsets lst
if empty? lst
report list lst
report append (subsets all but first of lst) (map (item 1 of lst in front of ☐) over (subsets all but first of lst))
```

FIGURE 1.2: Snap*!* Interactive Development Environment (IDE) with a tree recursive reporter block implementing subsets. Reporter blocks "speak" their reported (i.e., returned) values.

After teaching in Snap*!* for around 10 weeks (in a course of 15 weeks), CS10 transitions to teaching students Python, which is a text-based programming language. Although students feel comfortable with the computational ideas presented to them, there tends to be a higher learning curve for picking up the syntax, especially coming from a block-based programming language where syntax is relatively straightforward.

1.3 Study Goals

Currently, CS10 introduces students to Python using traditional programming exercises, in which students write freeform code. This study explores how using Parsons Problems could help scaffold students' transition between learning in Snap*!* and Python.

Parsons Problems are especially appropriate in this situation because students are used to programming in a block-based programming environment, which is similar to a Parsons Problem in that students drag and drop pre-written code blocks. Using Parsons Problems will then allow students to transition to a new programming language in an environment that is familiar to them from Snap*!*.

We hope that Parsons Problems will:

- Help students adjust to and feel more comfortable writing Python code
- Increase students' engagement and enjoyment of the course material
- Reduce the time students spend on the introductory Python assignment

In addition to exploring the above questions, we share various design considerations for tailoring Parsons Problems to learning syntax.

1.4 Related Work

While there has not been a lot of literature that specifically studies the effectiveness of Parsons Problems for learning syntax, there have been studies that evaluate Parsons Problems in the context of learning to program in general. Ericson et al.

[8] conducted a study in a CS1 course using Python that examined the effectiveness of Parsons Problems in the context of writing and fixing code. In the study, they partitioned students into three groups, all of whom were given the same problem in different formats: one group was given the problem as a Parsons Problem, another as a debugging exercise, and one as a code-writing exercise. To measure learning effectiveness, the researchers had students complete a pretest before their practice, a posttest directly after their practice, and a final posttest a week later. The pretests and posttests consisted of four different kinds of problems: multiple-choice, Parsons Problems, fixing code, and writing code. After the study, the researchers found no significant difference in scores across all three groups. They did, however find that the students assigned to the Parsons group finished their practice problems faster, suggesting that Parsons Problems are more efficient learning tools. Another study, conducted by Zhi et al. [21], studied the effectiveness of Parsons Problems in block-based learning environments. In this study, which used the block-based programming language Snap*!*, half of the assignments in the course were presented as Parsons Problems, and the other half were presented as code-writing problems. The researchers compared solution time and code quality across assignments and found that performance on each of the labs was similar, but the labs involving Parsons Problems took students about half as much time to solve. These findings are consistent with those from Ericson et al.'s study, suggesting that the effectiveness of Parsons Problems does not just stem from the elimination of syntax errors, but also the constrained problem space, which reduces cognitive load.

Another research area for Parsons Problems is in their design. One such design consideration is the use of two-dimensional Parsons Problems [15], in which students do not just have to place the blocks in the correct order, but also indent blocks correctly. While their effectiveness is still an active area of research, literature suggests that two-dimensional Parsons Problems have become the standard template for implementing Parsons Problems in Python, despite being more difficult to solve [5]. Another design consideration is the use of distractors, which are code blocks that are provided to students, but not used in the solution [3, 18]. The research in this area is also not conclusive, with some studies suggesting that distractors can be used to highlight and clear common misconceptions [18], while other studies suggest that they may actually decrease learning effectiveness [11]. It is worth noting that the

Ericson study that found Parsons Problems to be highly efficient for learning used Parsons Problems with distractors. [8]. Lastly, another area of research is the kinds of feedback given to students. One common form of feedback in Parsons Problems is line-based feedback [18], in which students are told which lines of their program are incorrect. However, some drawbacks of line-based feedback are that it requires problems to have a unique solution, and makes it easier for students to revert to trial-and-error based approaches to solve the problem [14]. Recently, the algorithm to assign line-based feedback was improved upon, and has not been evaluated in literature since [5, 16]. An alternative to line-based feedback is execution-based feedback, which gives students feedback similar to what they would get from compiling their code and running unit tests. Initial studies of this form of feedback show that students who receive execution-based feedback requested feedback less often and took more time to complete problems then students who received line-based feedback [14]. This suggests that execution-based feedback may make problems more difficult for students, but requires them to think more critically about the problem at hand.

2 Study Design

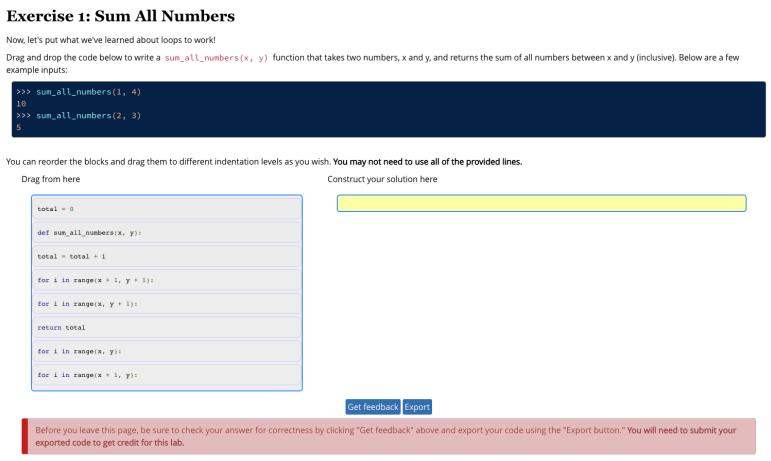
This study was conducted during the first Python lab in the Spring 2020 semester of CS10; a total of 83 students participated. Prior to the lab, the only in-class exposure students have had to Python is in one lecture, where students are introduced to some basic elements of the programming language. For the purposes of the study, we restructured the lab to introduce students to Python using Parsons Problems, transitioning to freeform code-writing only towards the end of the lab. Following the lab, we had students fill out surveys with their understanding of Python concepts, and their experiences with the new format.

2.1 CS10 Lab Context

In CS10, lab sections are the first place where students practice what they learn. During lab sections, students are given approximately two-hour long lab assignments, which they typically complete through pair-programming. Staff members are available to help students debug their code and understand concepts during lab time. Students who do not finish the lab during the allocated time are given one week to complete the assignment on their own [1].

At the time of this study, UC Berkeley had moved to online learning due to COVID-19. Lab sections were still being held online, but the teaching staff noticed a dramatic decrease in the number of students attended online lab sections, suggesting that students were completing labs on their own without working with their peers or soliciting as much help from staff members.

For this study, we tried to stay as true to the original lab as possible– the only change made to the content of the lab was the format of the problems. The lab had six questions, the first four of which we converted to Parsons Problems. The remaining two problems remained as is, asking students to write Python code in a text editor (with



Exercise 1: Sum All Numbers

Now, let's put what we've learned about loops to work!

Drag and drop the code below to write a `sum_all_numbers(x, y)` function that takes two numbers, x and y, and returns the sum of all numbers between x and y (inclusive). Below are a few example inputs:

```
>>> sum_all_numbers(1, 4)
10
>>> sum_all_numbers(2, 3)
5
```

You can reorder the blocks and drag them to different indentation levels as you wish. **You may not need to use all of the provided lines.**

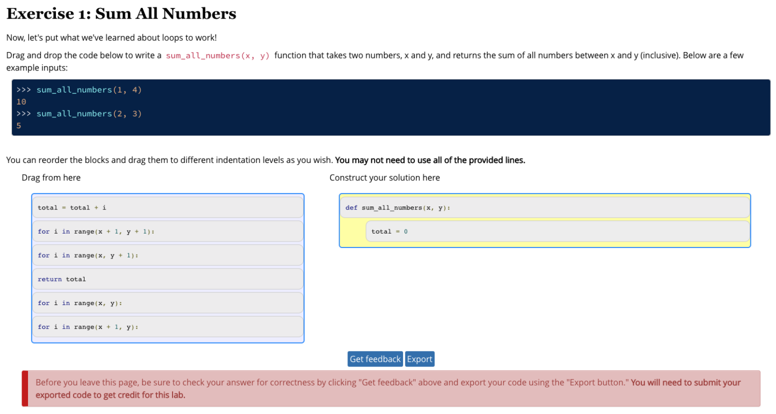
Drag from here

```
total = 0
def sum_all_numbers(x, y):
total = total + i
for i in range(x + 1, y + 1):
for i in range(x, y + 1):
return total
for i in range(x, y):
for i in range(x + 1, y):
```

Construct your solution here

Before you leave this page, be sure to check your answer for correctness by clicking "Get feedback" above and export your code using the "Export button." **You will need to submit your exported code to get credit for this lab.**

(A) Parsons Problem before student takes any actions.



Exercise 1: Sum All Numbers

Now, let's put what we've learned about loops to work!

Drag and drop the code below to write a `sum_all_numbers(x, y)` function that takes two numbers, x and y, and returns the sum of all numbers between x and y (inclusive). Below are a few example inputs:

```
>>> sum_all_numbers(1, 4)
10
>>> sum_all_numbers(2, 3)
5
```

You can reorder the blocks and drag them to different indentation levels as you wish. **You may not need to use all of the provided lines.**

Drag from here

```
total = total + i
for i in range(x + 1, y + 1):
for i in range(x, y + 1):
return total
for i in range(x, y):
for i in range(x + 1, y):
```

Construct your solution here

```
def sum_all_numbers(x, y):
    total = 0
```

Before you leave this page, be sure to check your answer for correctness by clicking "Get feedback" above and export your code using the "Export button." **You will need to submit your exported code to get credit for this lab.**

(B) Parsons Problem after student drags two code fragments to the solution space.

FIGURE 2.1: Parsons Problem for *Sum All Numbers* on lab.

no scaffolding) and run it through a Python shell. The goal of the lab is to help students translate their knowledge of Snap! to Python, without introducing any new forms of computational thinking.

The Parsons Problems were implemented using the `js-parsons` library, which was written by Ihantola et al. [15]. In addition to refactoring problems in the lab using this library, we also added more fine-grained logging, functionality to collect and aggregate logs, and a feature to allow students to export their code.

Below is a list of the problems on the assignment, presented in the order students worked on them. More details can be found in the Appendix, Section 5. Figure 2.1 shows a sample problem from the lab.

1. Parsons Problem for the "Hello World" program
2. Parsons Problem to return the sum of all numbers between x and y

3. Parsons Problem to write a function that calculates x^y for non-negative integer values of y
4. Parsons Problem to check if a word is a palindrome
5. Free-form problem to reverse a string
6. Free-form problem to draw a C-Curve fractal

2.1.1 Two-Dimensional Parsons Problems

Indentation is a key part of Python syntax. Since the goal of this lab is to help introduce students to Python syntax, and because indentation has semantic meaning in the Python language, we decided to use two-dimensional Parsons Problems, in which students have to arrange code both in the right sequence and with the correct indentation.

2.1.2 Use of Distractors

There is currently no strong consensus in the literature about whether the use of distractors in Parsons Problems is beneficial to student learning. Parsons and Haden [18], who first introduced Parsons Problems, suggested distractors can be helpful in addressing student misconceptions. In line with that guidance, we made use of distractors to address common mistakes students make with Python (e.g., missing colon, incorrect indexing, etc.).

For example, in the problem where students had to construct a program that added all of the numbers between x and y, inclusive, we included four versions of a `for` loop:

- `for i in range(x, y):`
- `for i in range(x, y + 1):`
- `for i in range(x + 1, y):`
- `for i in range(x + 1, y + 1):`

2.1.3 Feedback

We considered two options for feedback: line-based feedback, which tells students which lines of their code are incorrect, and execution-based feedback, which gives students feedback based on the results of executing their code. Execution-based feedback is helpful because it makes students think about the logic of their program as a whole, but it can't give students feedback about individual lines of code. Since the level of computational thinking in this lab is not new and we wanted to focus on teaching students syntax, we opted to use line-based feedback, as it helps students isolate exactly which lines of their code are incorrect, enabling them to more easily identify their misconceptions about Python syntax.

When students request feedback on their code, the system will first include a description of all of the student's errors in a pop up window, and then color all of their code fragments based on correctness, as shown in Figure 2.2. There are four possible classes of error messages a student can receive: *incorrect position*, *too few lines*, *too many lines*, and *incorrect indent*.

2.1.4 Data Collection

When students worked on problems, they were instructed to click the "Get Feedback" button shown in Figure 2.1 to verify the correctness of their solution, and then use the "Export" button to download their code for submission. We logged various student interactions with the problems, as described in the lists below, and we sent the logs to our database every time a student requested feedback. We then aggregated the following pieces of data per problem session, which we define as the time a unique user spent working on a problem before leaving the page:

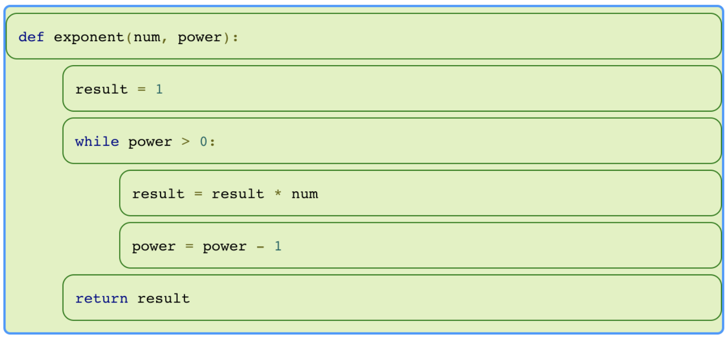
- Student actions (ex: dragged block into code, dragged block out of code, indented block, requested feedback, etc.)
- Start time of each action (we also used this to determine total time spent on the problem)
- Error messages
- Whether the student successfully solved the problem before they left the page

2.2 Checkoff Questions & Survey

Following the lab, we gave students an anonymous survey, in which they reflected on their experience with the lab, specifically regarding the different question formats. In addition to the survey, we also had students submit written checkoff questions (students are required to do these for all labs), which are questions we ask at the end of the lab to check students' understanding of the concepts. The checkoff questions ask about both general concepts and specific code.

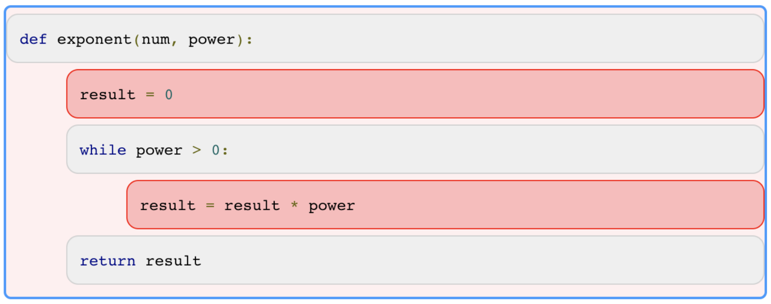
2.3 Limitations

The structure of this study had one major limitation: there was no formal control group. To the best of our abilities, we compared outcomes and attitudes from this lab to those from previous semesters, in which the presentation of Python material was almost identical (except the question formats). However, classes being forcibly transitioned to remote instruction due to COVID-19 changed the nature in which students interacted with labs and introduced some confounding variables to this study.



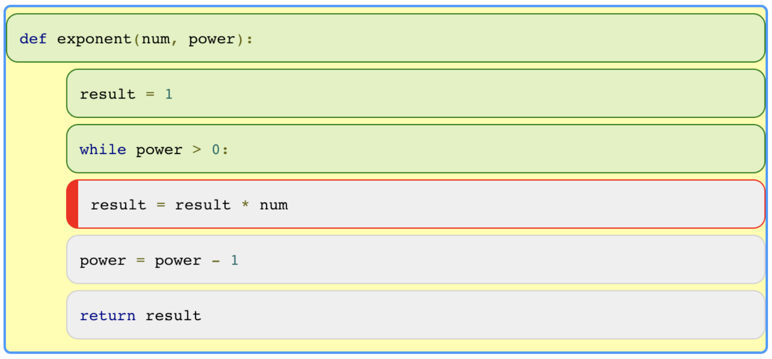
```
def exponent(num, power):
    result = 1
    while power > 0:
        result = result * num
        power = power - 1
    return result
```

(A) Example of feedback for correct solution.



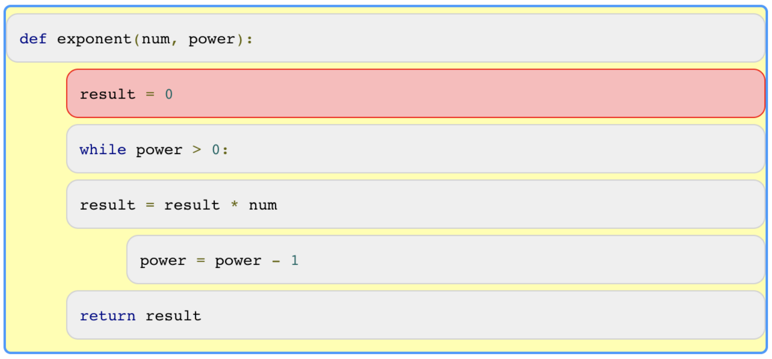
```
def exponent(num, power):
    result = 0
    while power > 0:
        result = result * power
    return result
```

(B) Example of feedback for solution with missing and incorrect lines. When a student requests feedback for this code, an alert will pop up saying "Code fragments in your program are wrong, or in wrong order. This can be fixed by moving, removing, or replacing highlighted fragments," and the incorrect code fragments will be colored in red. The logged errors for this code would be: incorrect position, too few lines.



```
def exponent(num, power):
    result = 1
    while power > 0:
    result = result * num
    power = power - 1
    return result
```

(C) Example of feedback for solution with incorrect indentation. When a student requests feedback for this code, an alert will pop up saying "The highlighted fragment 4 belongs to a wrong block (i.e. indentation)" ("fragment 4" here means the fourth code block). Note that the system will only highlight the first indentation issue. The logged errors for this code would be: incorrect indentation.



```
def exponent(num, power):
    result = 0
    while power > 0:
    result = result * num
        power = power - 1
    return result
```

(D) Example of feedback for solution with incorrect lines and indentation. When a student requests feedback for this code, an alert will pop up saying "Code fragments in your program are wrong, or in wrong order. This can be fixed by moving, removing, or replacing highlighted fragments." As in the last example, the system will only highlight the first issue. The logged errors for this code would be: incorrect position, incorrect indentation.

FIGURE 2.2: Examples of feedback for various attempts to solve the exponent problem on the lab.

3 Analysis

3.1 Overview

For our analysis, we combined data from our logs (597 entries) and the survey (87 responses) we administered after the lab. The logs included information about the amount of time students took, their number of attempts, their actions, the error messages they received, and whether or not they were ultimately successful. The survey included more qualitative data; it asked students to self-report their experiences and challenges with the different problems on the lab. In our analysis, we hoped to answer the following questions:

- How did students approach the Parsons Problems?
- What challenges did students face when working on the Parsons Problems, and how can that inform the design of future iterations of these problems?
- Did students find the Parsons Problems helpful in adapting to writing Python?
- Did students spend less time on the Parsons Problems, as the literature suggests?
- Did students enjoy working on the Parsons Problems?

Overall, we were pleased to find out that students overall found the Parsons Problems beneficial to their learning. Their challenges also gave us important insight into how we can improve future iterations of Parsons Problems for this use.

3.2 Student Approaches

To understand students' approaches to Parsons Problems, we first analyzed the number of attempts, which we defined as the number of times a student requested feedback on each problem, and the amount of time students spent on each problem, as shown in 3.1. Consistent with our expectations, the median number of attempts and amount of time spent was higher for problems later in the lab (which we believed were harder).

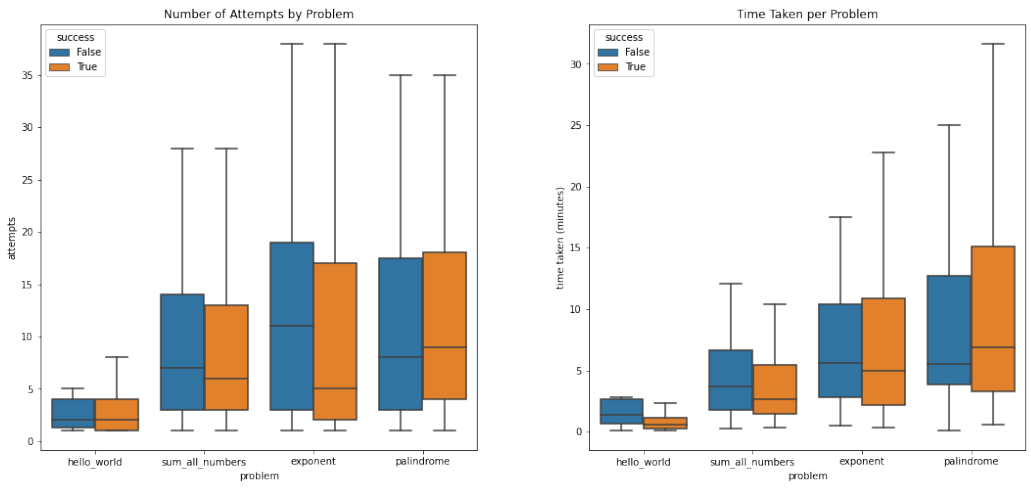


FIGURE 3.1: Graphs of the number of attempts and time taken per Parsons Problem

We also found no correlation between attempts (ρ = -0.11) or time taken (ρ = -0.08) and whether or not the student was ultimately successful in solving the problem in that sitting (about 20.4% of the total attempt sequences on a problem did not end with the student successfully solving the problem). However, a problem could also be marked unsuccessful if a student refreshed or left the page without finishing it, and then came back to finish it later (this would be marked as two different sequences, the first being unsuccessful and the second being successful). Most students submitted the lab with correct answers (correct code and correct answers to check-off questions), so we believe that most unsuccessful sequences in our logs were results of the latter scenario.

We then analyzed the average time taken per attempt for each of the Parsons Problems (excluding `hello_world`), as shown in Figure 3.2. We noticed that there seemed to be three main kinds of approaches:

- Students who made fewer attempts, but spent a lot of time on each attempt.
- Students who spent a moderate amount of time and attempts to solve the problem (this is by far the most common group).
- Students who made many attempts, and spent very little time on each attempt, implying a brute force or guess-and-check approach.

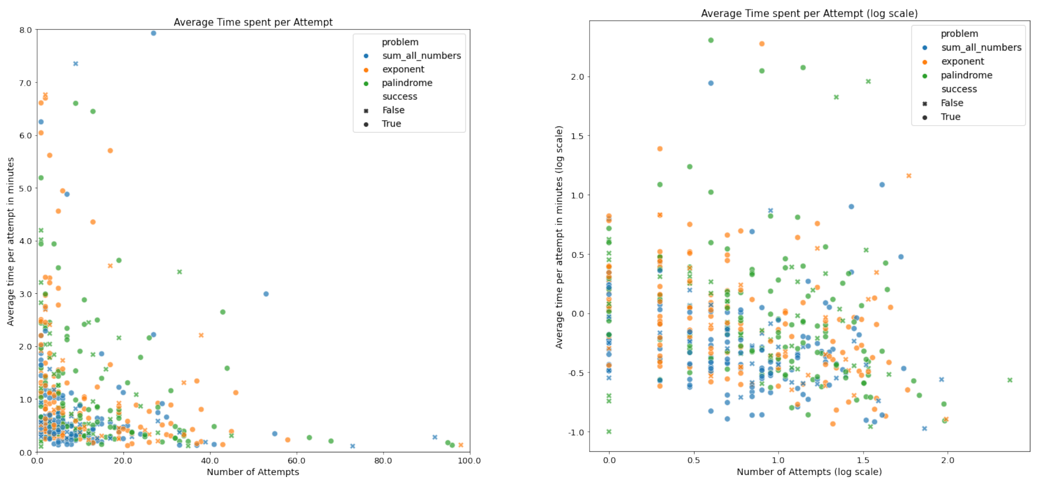


FIGURE 3.2: Graphs of the amount of time students took per attempt for the three hardest Parsons Problems. The graph on the left has no axis transformations (we also removed outliers), but clearly shows the right tail (high attempts, low time/attempt), and the upper tail (low attempts, high time/attempt). The graph on the right has a log transformation on each axis, and shows all of the data, including the outliers.

The last approach, which resembles a brute force strategy, could be a consequence of the use of line-based feedback, as it highlights each line students gets incorrect, allowing them to easily substitute different code blocks in their place [5].

Following this reasoning, we attempted to cluster the students to see if we could meaningfully group students based on their problem solving approaches. When clustering, we took into account the number of attempts, the actions a student took, the kinds of error messages they got, the amount of time it took, and whether they were ultimately successful, with various different preprocessing methods. We tried two different clustering algorithms, K-Means and Agglomerative Clustering (which is less sensitive to outliers), and two different dimensionality reduction methods to visualize the data (t-SNE and PCA). We did not find any meaningful clusters using these approaches– we believe that this could either be because we didn't have a large enough sample, or because there actually weren't distinct sets of approaches students take to solving these problems if we take into account their sequence of actions, not just their attempts.

3.3 Student Challenges

In the followup survey, we had students self-report their biggest difficulties working on the lab. The top answers for each problem type are summarized in Table 3.1. Table 3.2 describes potential areas of difficulty on each problem to better contextualize the data in Table 3.1.

Parsons Problems		Freeform Python	
Indentation	23 (26%)	Syntax	57 (66%)
Unclear feedback	16 (18%)	Running Python code in shell	12 (14%)
Algorithm	14 (18%)	Autograder/Unclear Feedback	8 (9%)
Syntax	11 (13%)	Algorithm	5 (6%)
Using Provided Code Blocks	8 (9%)	Indentation	2 (2%)
Distractors	4 (5%)	Lack of Help	2 (2%)

TABLE 3.1: Self-reported challenges for students as they did Parsons Problems and wrote free-form Python code.

Problem	Format	Description	Distractors?	Program Length
Hello World	Parsons	Function that prints "Hello, World!"	None	2 lines
Sum All Numbers	Parsons	Function that sums all numbers between x and y, inclusive	For loop conventions	5 lines
Exponent	Parsons	Function that computes x^y for non-negative integer y	General syntax, logic	6 lines
Palindrome	Parsons	Function that determines whether a string is a palindrome	General syntax, string syntax, logic	6 lines
Reverse String	Freeform Python	Function that reverses a string	n/a	5 lines
C-Curve	Freeform Python	Function that draws a recursive C-Curve using Turtle Graphics	n/a	9 lines

TABLE 3.2: Description of problems and potential areas of difficulty

3.3.1 Indentation

One challenge that students reported about Parsons Problems stood out in particular: indentation. While it is unclear how challenges with indentation compare across problem formats (since students may have reported issues with indentation under syntax as well), in previous iterations of the course students have been able to pick up indentation without reporting as much trouble.

One theory we have as to why students struggled more with indentation when using Parsons Problems is that when students type code in a text editor (which is how they wrote code for this lab), the text editor will correctly indent code following a `for` loop, `if` statement, or function definition. In the absence of a text editor, students may not know when to indent code, especially when they first start to program in a language like Python.

There is value to learning when to indent code, because even if the text editor automatically indents code for students, students still need to know when to unindent their code, like after the body of a `for` loop. We therefore do believe that there is value in teaching students indentation using Parsons Problems, though more attention may need to be paid to teaching students explicit indentation rules.

3.3.2 Feedback

Another interesting challenge students faced was around feedback, especially when compared across problem types. Overall, it seemed like students' reactions to the line-based feedback were mixed. While some students appreciated that the feedback tool highlighted exactly which lines of code were incorrect, others complained that the feedback they received was vague: *"it was unclear whether the highlighted errors referred specifically to the individual blocks or the entire section underneath them."* Interestingly, fewer students reported issues with feedback when completing the free-form Python exercises, which used a unit test-based autograder and provided feedback similar to execution-based feedback in Parsons Problems.

Another limitation of line-based feedback is that the problem needs to be set up in a way that allows for only one solution. Some students also expressed frustration with this: *"there are so so so many different answers to solving the same problem that I couldn't achieve with the blocks I was given in the [Parsons] problems."*

And finally, line-based feedback may have enabled more students to use a more brute-force approach to solving the problems when they were stuck, as described in the previous section. There currently is not much literature evaluating line-based feedback since the algorithm was improved upon [5, 16]. Some future directions could be to explore how the new line-based feedback algorithm affects students learning (both in terms of computational thinking and picking up syntax), and how it affects the prevalence of each of the three approaches mentioned in the previous section. Additionally, given that some students complained about the clarify of the feedback, it is worth exploring whether UI/UX changes in the feedback system could improve students' learning and experience with the system.

3.3.3 Comparison to Error Messages

Table 3.3 has a summary of the error messages that students received. Note that students could receive more than one error message in a single feedback request.

Error Message	Frequency
Incorrect Position	4095 (54.5%)
Too Few Lines	1861 (24.8%)
Too Many Lines	861 (11.5%)
Incorrect Indent	689 (9.2%)

TABLE 3.3: Error messages for Parsons Problems (6296 total feedback requests)

The error messages suggest that indentation may not have been as frequent of an issue as students self-reported. One hypothesis for why there is a mismatch between self-reported challenges and error messages is that coming from a block-based programming language, indentation is a new kind of error that students faced, which is why it was brought up more. However, even if indentation wasn't the most common error students faced, it was still the challenge they most commonly reported, and adequate steps should be taken to help scaffold learning indentation in the future. One possible way to do this could be to first introduce students to Python using 1D Parsons Problems, in which blocks of code are already pre-indented. Once they familiarize themselves with indentation patterns, then students can be introduced to 2D Parsons Problems, like the ones in this lab.

3.4 Helpfulness

Despite the challenges students faced, students' self-reported data suggests that students found the Parsons Problems quite helpful in adjusting to writing Python code ($p \approx 0$). This was especially true for students who had no previous Python experience, as shown in Figure 3.3.

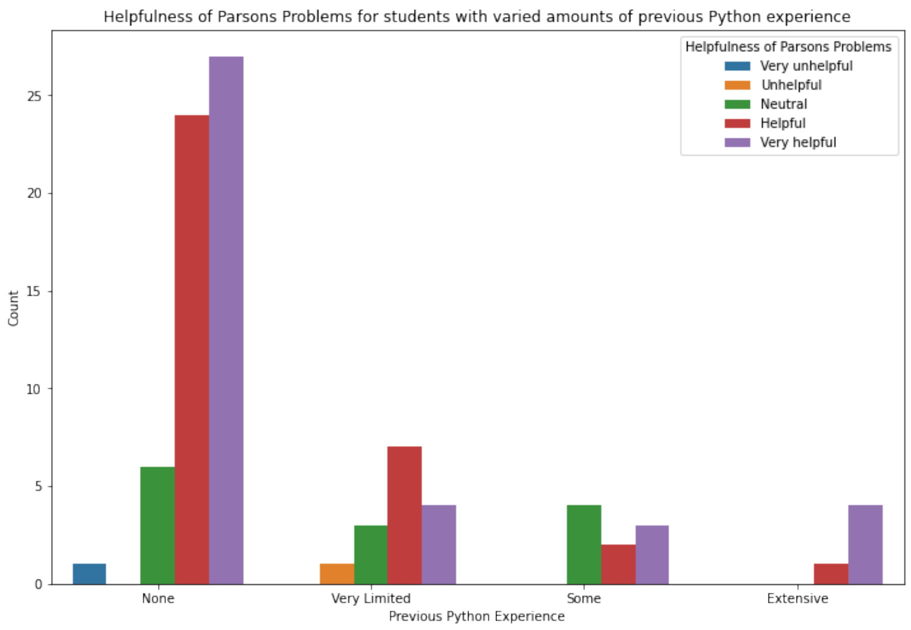


FIGURE 3.3: Graph of how helpful students found the Parsons Problems, augmented by their Python experience.

3.5 Efficiency

Students also confirmed our hypothesis that they would spend less time working on Parsons Problems as opposed to the freeform Python code ($p = 1.2 \times 10^{-9}$). This is especially promising, as high time commitment has long been a complaint from CS10 students, and has been noted as a reason for higher attrition in CS1 courses [17].

3.6 Enjoyment

One of our last goals for using Parsons Problems to introduce students to Python was to increase their engagement with and enjoyment of the material. Our data suggests that students enjoyed the Parsons Problems more than they enjoyed the freeform Python problems. ($p = 3 \times 10^{-10}$). The data for self-reported enjoyment is summarized in Figure 3.4.

This data is exciting, as it reaffirms that Parsons Problems are a more engaging and enjoyable way to introduce students to Python. However, at the same time, it opens up a lot of questions about why students found Parsons Problems more enjoyable. Some studies suggest that Parsons Problems are more enjoyable because of their

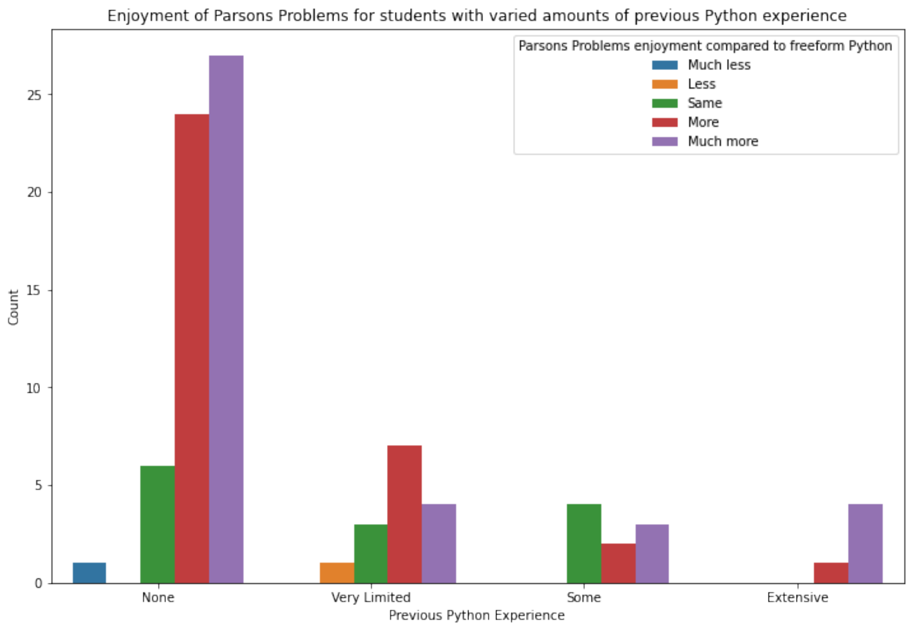


FIGURE 3.4: Graph of how much students enjoyed Parsons Problems as compared to free-form Python problems, augmented by experience.

inherent puzzle-like structure, which gamifies the programming experience [18]. However, we believe that another contributor to the observed difference in enjoyment may be due to students' familiarity with concepts: while students are used to the drag-and-drop programming environment present in Parsons Problems, they initially faced challenges picking up Python syntax, which became especially apaprent when they wrote freeform Python code. Students reported having less trouble with Parsons Problems, which may be a reason they enjoyed them more.

3.7 Experience Level

As we saw in Figures 3.3 and 3.4, Parsons Problems are particularly beneficial for students with no previous Python programming experience (which describes the majority of CS10 students, as shown in Figure 3.5). Out of the 87 students who participated in this study, 83% found Parsons Problems helpful for transitioning to Python, 69% found them more enjoyable than writing freeform Python code, and 68% found them to be more time-efficient than writing freeform Python code. However, out of the 58 students who had no previous Python programming experience, 88% found Parsons Problems helpful for transitioning to Python, 74% found them more enjoyable than writing freeform Python code, and 71% found them to be more

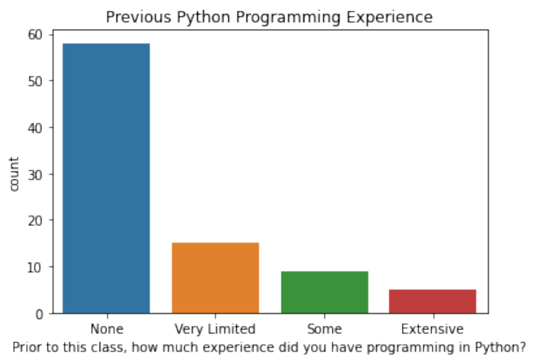


FIGURE 3.5: Students' experience with Python before taking CS10.

time-efficient.

Given the research about Parsons Problems, it makes sense why they would be especially beneficial for beginners: they help scaffold students' learning so they can master certain elements of Python programming before writing programs from scratch. For example, for this lab, using Parsons Problems helped students familiarize themselves with syntax and indentation patterns in Python before jumping into freeform problems, which require a stronger grasp of indentation and syntax.

4 Conclusion

4.1 Future Work

To date, there exists relatively little literature on the effectiveness of Parsons Problems [5]. This study presents some findings about the use of Parsons Problems to help students learn a new programming language, while opening up various directions for future studies.

One limitation of this study was the lack of a control group, making it difficult to isolate the effect of Parsons Problems from other confounding variables, especially given classes were no longer held in person. Additionally, much of the analysis relied on self-reported data; a future direction could be to use methods like pre/post tests, freefrom problem logging, and interviews to collect similar types of data, and examine its consistency with the data from this study.

This study raised many questions about Parsons Problems in general. One open direction is an investigation of different feedback types in Parsons Problems, which hasn't been done since the algorithm for line-based feedback was updated. This study uses the line-based feedback method– while some students found it helpful, it also had various consequences, including opening up the possibility of using a brute force approach to solve the problem, and limiting the solution space of a single solution. Future studies could ask how the feedback type affects how students approach problems, and what they later take away from them.

Another future direction, as mentioned in the analysis section, is to further investigate why students find Parsons Problems more engaging and enjoyable than freeform Python problems. Is it because of the inherent structure of Parsons Problems, as suggested by Parsons and Haden [18], or could it be because of students' familiarity with syntax, as this paper hypothesizes? Does students' enjoyment of the different kinds of problems evolve with their experience, and how can we take elements of Parsons Problems to make traditional Python problems more enjoyable?

Additionally, given that students struggled with indentation, future studies could include a more scaffolded approach to introducing students to indentation. Early problems could start as one-dimensional Parsons Problems, where blocks are pre-indented and students only need to worry about putting code in the correct order. Slowly, indentation-based scaffolding could be removed, perhaps by first introducing distractors and then changing the format to two-dimensional Parsons Problems. Eventually, the problems could transition from focusing on syntax to focusing on logic, until students are ready for freeform Parsons Problems. Adaptive Parsons Problems, which adjust in difficulty and scaffolding based on student performance, could also work well in this context [6].

One challenge in CS10 students' transition from Snap*!* to Python that this study does not investigate is confidence: in previous semesters, some students noted that their challenges adjusting to programming in Python made them feel less confident in their programming abilities as a whole. This study suggests that Parsons Problems help students adjust to writing Python code; could it also affect their confidence when it comes to their programming abilities? If so, how does this inform how we should use Parsons Problems in introductory computer science courses?

4.2 Closing Summary

This study investigated how Parsons Problems could help students learning a new programming language. It confirmed the findings of a lot of existing literature: Parsons Problems take students less time and are more engaging than their traditional freeform counterparts, suggesting that they are an effective and efficient learning tool. It also provided promising results about how Parsons Problems can aid students transition to a new programming language: a majority of students found it helpful, and were able to finish lab exercises correctly. Parsons Problems are an effective means of scaffolding student learning; in this case they allowed students to transition to Python more slowly, focusing on individual elements of Python programming before writing entire programs in it. Based on our conclusions from this study, we recommend the use of Parsons Problems to help students familiarize themselves with new syntactic structures, as they are more effective, efficient, and enjoyable than their traditional counterparts.

5 Appendix

5.1 Lab Questions

The full version of the lab can be accessed here.

5.1.1 Parsons Problems

Hello World

Question: Write a function that prints the text "Hello, World!"

Learning Goals: Familiarize students with Parsons Problem interface (dragging and dropping code, indentation, requesting feedback).

Code Provided: (Order is random)

- `def hello_world():`
- `print("Hello, World!")`

Solution:

```
def hello_world():
     print("Hello, World!")
```

Sum All Numbers

Question: Write a function that takes in as input two numbers, x and y, and returns the sum of all numbers between x and y (inclusive).

Learning Goals: Familiarize students with for loops.

Code Provided: (Order is random)

- `def sum_all_numbers(x, y):`
- `total = 0`
- `for i in range(x, y + 1):`
- `for i in range(x, y):` (distractor)

- `for i in range(x + 1, y):` (distractor)
- `for i in range(x + 1, y + 1):` (distractor)
- `total = total + i`
- `return total`

Solution:

```
def sum_all_numbers(x, y):
    total = 0
    for i in range(x, y + 1):
        total = total + i
    return total
```

Exponent

Question: Write an `exponent(num, power)` function that takes two arguments (a number and an exponent) and returns the computed result. (Also includes sample input/outputs)
Learning Goals: Familiarize students with while loops.
Code Provided: (Order is random)

- `def exponent(num, power):`
- `def exponent(num, power)` (distractor)
- `result = 1`
- `result = 0` (distractor)
- `while power > 0:`
- `result = result * num`
- `result = result * power` (distractor)
- `result = result * 2` (distractor)
- `power = power - 1`
- `return result`

Solution:

```
def exponent(num, power):
    result = 1
    while power > 0:
        result = result * num
        power = power - 1
    return result
```

Palindrome

Question: Write a function that takes in an input string and returns `True` if the string is a palindrome and `False` otherwise.
Note: Students have already written this exact function in Snap*!*, so the logic should not be new to them.
Learning Goals: Familiarize students with `if/elif/else`, string operations, and writing recursive code in Python.
Code Provided: (Order is random)

- `def palindrome(string):`
- `if len(string) < 2:`
- `if len(string) < 2` (distractor)
- `return True`
- `return False` (distractor)
- `elif string[0] != string[-1]:`
- `elif string[0] != string[1]:` (distractor)
- `return False`
- `return palindrome(string[1:-1]`
- `return palindrome(string[0:-1])` (distractor)

Solution:

```
def palindrome(string):
    if len(string) < 2:
        return True
    elif string[0] != string[-1]:
        return False
    return palindrome(string[1:-1]
```

5.1.2 Free-Form Questions

Reverse String

Question: Write a function that takes in a string and returns the same string in reverse order.

Learning Goals: Familiarize students with loops and string operations in Python.

Solution: (There may be many possible solutions)

```
def reverse_string(string):
    new_string = ""
    for letter in string:
        new_string = letter + new_string
    return new_string
```

C-Curve

Question: Write a function that draws the fractal pattern shown in Figure 5.1.

Note: Includes introduction to turtle graphics before the question. Students have had extensive practice with drawing fractals in Snap*!*

Learning Goals: Familiarize students with turtle library. **Solution:** (There may be many possible solutions)

```
import turtle as t
def c_curve(level, size):
    if level == 1:
        t.forward(size)
    else:
        t.left(45)
        c_curve(level - 1, size / 2)
        t.right(90)
        c_curve(level - 1, size / 2)
```

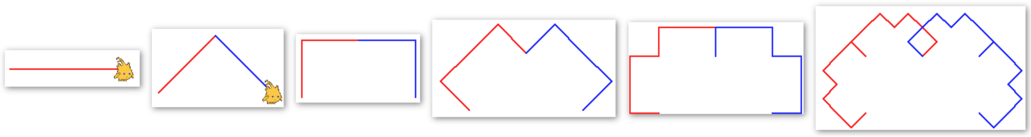


FIGURE 5.1: C-Curve fractal for students to recreate

```
t.left(45)
```

5.2 Survey Questions

1. Prior to this class, how much experience did you have programming in Python?
    - No experience at all
    - Very limited exposure
    - Some exposure
    - Extensive exposure (an entire class, for example)
2. In this lab, we tried out a new question format, in which we had you drag and drop Python code. How helpful did you find this kind of question when adjusting to writing Python? (Answers were on a 5-point Likert scale)
3. How much did you enjoy working on the drag and drop Python problems as compared to the regular Python problems (like reverse_string)? (Answers were on 5-point Likert scale)
4. How did you feel about the balance of problems on this lab? (Answers were given from 1-5, where 1 was "There were too many drag and drop problems, I wish we had more freeform Python problems" and 5 was "There were too many freeform Python problems, I wish we had more drag and drop problems")
5. How did the amount of time you spent compare between the drag and drop problems and the freeform Python problems?
    - I spent more time on the drag and drop problems.
    - I spent about the same time on each of the kinds of problems.
    - I spent more time on the freeform Python problems.

6. About how much time total did you spend on this lab?
    - <1 hour
    - 1-2 hours
    - 2-3 hours
    - 4-6 hours
    - 6+ hours
7. What were the greatest challenges you faced when completing the drag and drop exercises?
8. What were the greatest challenges you faced when completing the freeform Python exercises?
9. Do you have any other comments/feedback about the lab?

5.3 Logs

We first cleaned the raw logs, and organized them in a table with the schema below. Each row represented a single session, which we define as the time a student spent on a single page before refreshing or leaving the page. We had a total of 597 entries in our log table.

- `ID`: Unique identifier for single session (time spent on page)
- `problem`: Name of problem
- `attempts`: Total number of attempts before leaving page
- `start_time`: Start time on page
- `end_time`: End time on page
- `actions`: Sequence of actions. Possible actions are:
    - add output: Add line to solution
    - feedback: Request feedback
    - move input: Line moved within input space
    - move output: Move line in solution

  - remove output: Remove line from solution

- `errors`: Errors student received, in order. Possible errors are:
  - incorrect indent
  - incorrect poisition
  - too few lines
  - too many lines
- `success`: Whether the student successfully solved the problem before leaving the page.
- `total time`: Total time spent on page

Bibliography

[1] CS10: The Beauty and Joy of Computing, Spring 2020. UC Berkeley EECS. https://cs10.org/sp20.

[2] BEAUBOUEF, T., AND MASON, J. Why the high attrition rate for computer science students: Some thoughts and observations. *SIGCSE Bull. 37*, 2 (June 2005), 103–106.

[3] DENNY, P., LUXTON-REILLY, A., AND SIMON, B. Evaluating a new exam question: Parsons problems. In *Proceedings of the Fourth International Workshop on Computing Education Research* (New York, NY, USA, 2008), ICER '08, Association for Computing Machinery, p. 113–124.

[4] DICHEVA, D., AND HODGE, A. Active learning through game play in a data structures course. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education* (New York, NY, USA, 2018), SIGCSE '18, Association for Computing Machinery, p. 834–839.

[5] DU, Y., LUXTON-REILLY, A., AND DENNY, P. A review of research on parsons problems. In *Proceedings of the Twenty-Second Australasian Computing Education Conference* (New York, NY, USA, 2020), ACE'20, Association for Computing Machinery, p. 195–202.

[6] ERICSON, B., MCCALL, A., AND CUNNINGHAM, K. Investigating the affect and effect of adaptive parsons problems. In *Proceedings of the 19th Koli Calling International Conference on Computing Education Research* (New York, NY, USA, 2019), Koli Calling '19, Association for Computing Machinery.

[7] ERICSON, B. J., GUZDIAL, M. J., AND MORRISON, B. B. Analysis of interactive features designed to enhance learning in an ebook. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research* (New York, NY, USA, 2015), ICER '15, Association for Computing Machinery, p. 169–178.

[8] ERICSON, B. J., MARGULIEUX, L. E., AND RICK, J. Solving parsons problems versus fixing and writing code. In *Proceedings of the 17th Koli Calling International Conference on Computing Education Research* (New York, NY, USA, 2017), Koli Calling '17, Association for Computing Machinery, p. 20–29.

[9] GARCIA, D., HARVEY, B., AND BARNES, T. The beauty and joy of computing. *ACM Inroads 6*, 4 (Nov. 2015), 71–79.

[10] GARCIA, R., FALKNER, K., AND VIVIAN, R. Scaffolding the design process using parsons problems. In *Proceedings of the 18th Koli Calling International Conference on Computing Education Research* (New York, NY, USA, 2018), Koli Calling '18, Association for Computing Machinery.

[11] HARMS, K., CHEN, J., AND KELLEHER, C. Distractors in parsons problems decrease learning efficiency for young novice programmers. pp. 241–250.

[12] HARMS, K. J., ROWLETT, N., AND KELLEHER, C. Enabling independent learning of programming concepts through programming completion puzzles. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (2015), pp. 271–279.

[13] HARVEY, B., AND MÖNIG, J. Snap! (build your own blocks). `https://snap.berkeley.edu`.

[14] HELMINEN, J., IHANTOLA, P., KARAVIRTA, V., AND ALAOUTINEN, S. How do students solve parsons programming problems? – execution-based vs. line-based feedback. In *2013 Learning and Teaching in Computing and Engineering* (2013), pp. 55–61.

[15] IHANTOLA, P., AND KARAVIRTA, V. Two-dimensional parson's puzzles: The concept, tools, and first observations. *Journal of Information Technology Education 10* (2011), 119–132.

[16] KARAVIRTA, V., HELMINEN, J., AND IHANTOLA, P. A mobile learning application for parsons problems with automatic feedback. pp. 11–18.

[17] KINNUNEN, P., AND MALMI, L. Why students drop out cs1 course? In *Proceedings of the Second International Workshop on Computing Education Research* (New York, NY, USA, 2006), ICER '06, Association for Computing Machinery, p. 97–108.

[18] PARSONS, D., AND HADEN, P. 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* (AUS, 2006), ACE '06, Australian Computer Society, Inc., p. 157–163.

[19] PRICE, T. W., AND BARNES, T. Comparing textual and block interfaces in a novice programming environment. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research* (New York, NY, USA, 2015), ICER '15, Association for Computing Machinery, p. 91–99.

[20] WEINTROP, D., AND WILENSKY, U. Comparing block-based and text-based programming in high school computer science classrooms. *ACM Trans. Comput. Educ. 18*, 1 (Oct. 2017).

[21] ZHI, R., CHI, M., BARNES, T., AND PRICE, T. W. Evaluating the effectiveness of parsons problems for block-based programming. In *Proceedings of the 2019 ACM Conference on International Computing Education Research* (New York, NY, USA, 2019), ICER '19, Association for Computing Machinery, p. 51–59.