Examining Novice and Experienced Programmers' Use of Python in a Combinatorial Context

Rebekah Kuss
Oregon State University

Elise Lockwood
Oregon State University

We explore the use of computational tools and techniques to enrich student's combinatorial reasoning by contrasting novice and experienced programmers in their exploration of combinatorial tasks in the context of Python. Drawing upon Lockwood's model of combinatorial thinking, we present data from interviews with students who were engaging with Python modules designed to help students solve counting problems. We contrast pairs of students who had different levels of prior experience with coding, and we highlight the experienced coders' solution that reflects how their experience with coding arose in their combinatorial solution.

Keywords: combinatorics, discrete mathematics, computing, programming.

Introduction

The use of computational tools and techniques has long supported mathematical activity (e.g., Papert, 1980), and there have been recent calls for studies that explore the role of computing in enriching students' mathematical reasoning (e.g., Geraniou et al., 2024; Lockwood & Morken, 2020). There is a growing body of work that highlights the potential of computational settings and activities to support student's mathematical thinking in a variety of contexts (e.g., Buteau et al., 2019; Sinclair & Patterson, 2018). In this report, we seek to add to this literature by contrasting novice and experienced programmers in their exploration of combinatorial tasks in the context of Python. The work in this report contributes by illuminating some aspects of programming that more experienced programmers may draw upon in such contexts.

Literature and Theoretical Perspectives

Counting Processes, Sets of Outcomes, and Formulas/Expressions

To characterize what we mean by combinatorial thinking, we draw on Lockwood's (2013) model of students' combinatorial thinking, which includes three main components of counting processes (the processes one engages with in solving a counting problem), formulas/expressions (the numbers, variables, and operations that often give the answer to a counting problem), and sets of outcomes (the set of objects being counted by a counting problem; the cardinality is the solution to the counting problem). In this model and in subsequent work, Lockwood and others (e.g., Soto et al., 2022) have made a case for the value of giving students experiences to connect counting processes they employ with the sets of outcomes those processes generate. It is desirable to us for students to make connections among these components, and the use of the computer is a motivation to foster connections between counting processes and outcomes.

Notably, in some prior work we have explored students' programming in the context of counting problems (e.g., De Chenne & Lockwood, 2022; Lockwood & De Chenne, 2020), but such work has focused mostly on novice programmers with no prior programming experience. Some work has examined an experienced coder with no combinatorial experience (De Chenne & Lockwood, 2023), but we have not yet reported on such tasks with students who had both combinatorial and coding experience. There is a need in this literature for more insights into the work of what more experienced coders bring to these combinatorial tasks.

Programming and Machine-based Computing

In this report, we focus on *machine-based computing*, as defined by Lockwood and Morken (2021): "the practice of developing and precisely articulating algorithms that may be run on a machine" (p. 405). We are interested in approaches that foreground the development of an algorithm as opposed to only its performance or implementation. In our study, this machine-based computing was programming, which entailed both creating code and editing existing code (in alignment with Weintrop et al., 2016). In the context of coding, we conceive of the programs as representing counting processes (in the sense of Lockwood 2013), as they are representative of the process of generating outcomes. The output that the computer then generates from that code represents the sets of outcomes. This is in line with other work (e.g., De Chenne & Lockwood, 2022; Lockwood & De Chenne, 2021) that has connected the computer to components of combinatorial thinking and activity in this way. We seek to answer the following research question: *What kinds of computational activity did experienced coders engage in as they solved counting problems in a computational context?*

Methods

Data Collection

The work described in this paper was part of data collected during a whole-class implementation of materials involving Python in the context of combinatorics. We developed a set of modules in which students engaged with Python and solved counting problems. These were implemented in a classroom setting of an upper-level mathematics course designed for pre-service teachers (the second author was also the instructor for the course). The modules covered six 80-minute class sessions, and they began with an introduction to Python and progressed to increasingly advanced combinatorics problems. The entire class worked through the modules, and we video- and audio-recorded two pairs of students for those six sessions. In those recordings, we both recorded the students as they worked, and we recorded their screens; we spliced together those inputs for analysis. As they worked in pairs, the instructor and another researcher would occasionally check in on how they were doing and would ask questions about their thinking and their work.

For this report, we focus particularly on one of the pairs of students we recorded, Zach and Levi (all names are pseudonyms). As a contrasting case, we briefly describe the solution of another pair Taylor and Kirk. Taylor, Kirk, and Levi were math majors enrolled in the secondary education program, Zach was a math major not enrolled in the secondary program. They were all senior-level mathematics majors with strong mathematics backgrounds. Taylor and Kirk had taken discrete mathematics but no advanced combinatorics; Zach and Levi students were taking an upper-division combinatorics class at the time of the interviews.

Taylor had no prior experience; Kirk had seen VPython, Matlab, and RStudio but said, "I don't remember how to do anything." Levi did not articulate prior experience with programming, and his work suggested he was a relative novice but felt comfortable trying to program. Zach had a strong background in programming, having coded previously in Python. At the time we did not ask what specific courses he had taken, but he talked about and demonstrated prior experience with coding in Python. This is borne out in their work, and we consider a relative comparison of *novice* for Taylor, Kirk, and Levi, and *experienced* for Zach; we are not making a claim about Zach's relative expertise to expert coders, but Zach was certainly more experienced than the other students and students we have reported on previously (e.g., Lockwood & De Chenne, 2021). Practically, this meant that he drew upon techniques (such as using functions and structures) that suggested familiarity with Python programming.

Data Analysis

For data analysis for this report, we began by examining Zach and Levi's work to investigate ways in which a pair of students that had more experience (in Zach) engaged with the problems. Specifically, the authors first watched the spliced videos of the interviews with Zach and Levi and regularly met to discuss broad themes that emerged from them. Then, the first author reviewed all of the interviews again and identified and collected episodes where the students specifically engaged with the code in a way that reflected their counting process. From these instances, the authors collaboratively wrote the episodes and together chose a representative example that highlighted an interesting solution to a counting problem that reflected the student's advanced knowledge of coding. The first author then reviewed the example and analyzed the students' thought processes. Finally, the first author analyzed Taylor and Kirk's solution to the same counting problem to see how they worked through the problem.

Results

We present data on the Language Books problem, which states: *There are 3 different French books, 4 different Norwegian books, and 5 Spanish different books. Create a computer program to list all of the possible book combinations of two books, both of which are not in the same language.* The students were asked to write code to solve the problem. We begin by only briefly (due to space) presenting Taylor and Kirk's solution, which is the solution we anticipated when we wrote the problem. We show this mainly to highlight a contrast with Zach and Levi's work, which we present in more detail. Then, we present Zach and Levi's solution, which demonstrates what the additional coding experience facilitated for their work. The point of the comparison is not to criticize the novice coders, but rather to give some sense of the kinds of tools and resources that might arise if such tasks are presented to more experienced students.

Novice Coders' Solution to the Language Books Problem

The first solution that Taylor and Kirk tried was a single nested for loop. This solution loops through all the French, Norwegian, and Spanish books, but it will repeat pairs causing the code to overcount. It appears in Figure 1a below, and the partial output shows some repeated pairs (such as F1 N1 appearing multiple times). The students were not sure how to fix their code in order to keep a single for loop and still avoid the overcounting that arose. In order to combat this overcounting, the novice coder group decided to use three separate for loops (Figure 1b). This code uses three separate, consecutive for loops that separately handle pairs of French and Norwegian, French and Spanish, and Spanish and Norwegian books. This case breakdown provides a cleaner way to organize the problem that avoids the overcounting that previously arose, without having to have the code check to see if there are repeats. As a result, Taylor and

```
French = ['F1','F2','F3']
Norwegian = ['N1','N2','N3','N4']
Spanish = ['S1','S2','S3','S4','S5']

books = 0
for i in French:
    for j in Norwegian:
        for k in Spanish:
            print(i,j)
            print(i,k)
            print(j,k)
            books = books+1
print(books)
```

```
F1 N1
F1 S1
N1 S1
F1 N1
F1 S2
N1 S2
F1 N1
F1 S3
N1 S3
F1 N1
F1 S4
N1 S4
F1 N1
F1 S5
N1 S5
```

```
French = ['F1','F2','F3']
Norwegian = ['N1','N2','N3','N4']
Spanish = ['S1','S2','S3','S4','S5']

books = 0
for i in French:
    for j in Norwegian:
        print(i,j)
        books = books+1
for i in French:
    for k in Spanish:
        print(i,k)
        books=books+1
for j in Norwegian:
    for k in Spanish:
        print(j,k)
        books=books+1
print(books)
```

Figure 1a and 1b. Figure 1a shows Kirk and Taylor's initial attempt at a single for loop. Figure 1b shows their solution for code that uses three consecutive for loops.

Kirk are utilizing sets of outcomes from Lockwood's model (2013). By thinking about how to do this process, they relied on knowing that the final set of outcomes would have pairs of French

and Norwegian, French and Spanish, and Spanish and Norwegian books. This is a correct answer to the problem, and indeed it was the solution we expected when we designed the task.

The output of the code in Figure 1b is shown in Figure 2. Note that each nested for loop creates one line of these lists (the first nested for loop created pairs of French and Norwegian books, the second created pairs of French and Spanish books, and the third created pairs of Spanish and Norwegian books). To reflect formulas/expressions of the model, we can consider an expression to reflect this process. For each nested for loop, we multiply the number of items in the outer array and the number of items in the inner array, which is also how Taylor and Kirk thought of it. Since we are counting up by one for each time the loop goes through, we end up adding up all the times each of the nested-for loops completes a loop, and so we sum up the products. This code thus reflects the sum of products, 3*4 + 3*5 + 5*4, which is a result that Taylor and Kirk found after they had written their code using the three consecutive loops.

```
F1 N1  F1 N2  F1 N3  F1 N4  F2 N1  F2 N2  F2 N3  F2 N4  F3 N1  F3 N2  F3 N3  F3 N4          12

F1 S1  F1 S2  F1 S3  F1 S4  F1 S5  F2 S1  F2 S2  F2 S3  F2 S4  F2 S5  F3 S1  F3 S2  F3 S3  F3 S4  F3 S5          15

S1 N1  S1 N2  S1 N3  S1 N4  S2 N1  S2 N2  S2 N3  S2 N4  S3 N1  S3 N2  S3 N3  S3 N4  S4 N1  S4 N2  S4 N3  S4 N4  S5 N1  S5 N2  S5 N3  S5 N4          20
```

Figure 2. The output of the code from Figure 1b. This was printed horizontally to save space.

Experienced Coder's Solution to the Language Books Problem

We now share results from Zach and Levi as they solved this problem. As noted, Zach especially was an experienced programmer. The students attempted a different solution than we had anticipated. Zach and Levi also attempted to create a single nested for loop, and we highlight what aspects of coding they needed to draw upon in order to do this successfully.

Zach and Levi create a single nested for loop. The first thing when creating their for loop was making sure that the code "looked forward" in the list; this was a term Zach used. By this, we infer that he meant that if X is the 5th item in the list, Y would only be the 5th or higher item in the list. The excerpt below shows Zach's reasoning about this, writing the code in Figure 3.

Zach: So, I was gonna do for X in range, uh, 1, 13.

Levi: Oh, yeah.

Zach: And then for, um... 'Cause I don't wanna have F1, R1, and then, uh, at... back at R1 go back to F1. So, I can actually do for Y in range X13. Right, so now if I'm... if I... so now if I start with like three- I'm not gonna look backwards. [...] It can only look forwards.

```
books = ['F1','F2','F3','R1','R2','R3','R4','S1','S2','S3','S4','S5']

for x in range(1,13):
    for y in range(x,13):
```

Figure 3. Zach's use of the range function in the code.

To do this, Zach used the range function so that the code would look at the first element of the list ("for x") and then the second element of the list ("for y") by putting an "x" into the range so that it would look at the number that the first for loop is on and only after that number. We suggest that this approach is indicative of someone who is more advanced in coding. While more novice coders may know that the range function can be used to create a list of numbers, we argue that Zach is using more advanced knowledge of coding here. In particular, he uses a nested for loop with range functions in order to force the second for loop to only look at the element that x is on and after that element. The use of range(x,13) suggests that he understood conceptually how the range function works, as well as understanding that this will use the x created in the outer for loop as a variable within the range function.

However, Levi and Zack recognized that this code would not ensure that they would not repeat the same language. As seen in the excerpt below, Levi recognizes that "F1, F2" is still a possibility. They discuss this below, and Zach tries to use nested indexing (which we would argue is a more advanced coding practice) to check that he can isolate the first elements of the string. In Figure 4 we see his initial work on comparing the first letters of each pair.

Levi: But then you can still get F1, F2.

Zach: Right. And then I can say, uh,

Levi: If-

Zach: If books at X at one is equal to... is... I'm sorry, is not equal to books at Y at one. All right, this should be a zero, shouldn't it? This is a zero.

Levi: Oh yeah, yeah, yeah.

Zach: So, this is like, the X inden- Uh, indices [...] Actually, this should be zero as well, shouldn't it? [...] So then it's gonna probably pull the first things of that string. So, when... if I pull F1 and F2-

```
books = ['F1','F2','F3','R1','R2','R3','R4','S1','S2','S3','S4','S5']

for x in range(0,12):
    for y in range(x,12):
        if books[x][0] != books[y][0]
```

Figure 4. Zach using nested indexing to look at the first letter of the x-th element of the array and making sure it is not the same as the first letter of the y-th element of the array.

Zach then checked this work to test whether the code is in fact isolating the first letter. He successfully "pulls" just the first element of the string, as seen in the column of Fs in Figure 5.

```
books = ['F1','F2','F3','R1','R2','R3','R4','S1','S2','S3','S4','S5']

for x in range(0,12):
    for y in range(x,12):
        print(books[x][0])
```

```
F
F
F
F
F
F
F
F
```

Figure 5. Zach testing the nested indexing function to see if it does what he thinks it does.

Zach: [...] It's gonna read it as, um, FF. So that... So, books... well, here. You could do like, print books at X at zero. Let me see what that does. So that- that's printing F. So that's seeing it as looking at the string- [...] First then look- so books X looks at the string and then zero pulls the first element of the string, which is just F.

Levi: Oh, okay.

Zach and Levi conclude their work on the problem in Figure 6, and in the following excerpt Zach explains his reasoning. Having successfully isolated the first letter, he can include a conditional if statement that only counts pairs where the first letter (the books[x][0] and books[y][0] are not equal). Here, they first make sure that the language of the books would not be the same by having the code look at the first letter and only keep going if the first letter of the first book selected and the second book selected were not the same, as Zach explains.

Zach: So, if we have book... so, as long as the letters aren't the same, books X-[...] at zero, as long as it's not equal to books, our yth component at zero... We gotta be careful with that one. I'll do plus one there I think […] Yeah. 'Cause like, as long as I have this, this is gonna say F1 and then for Y in range zero, it's gonna look at F1 again [...] It's gonna compare it to itself, but obviously it- it can't be the same as itself.

Then we do count equals, uh, zero. And then if they're not the same, the letters aren't the same […]

```
books = ['F1','F2','F3','R1','R2','R3','R4','S1','S2','S3','S4','S5']
count = 0
for x in range(0,12):
    for y in range(x+1,12):
        if books[x][0] != books[y][0]:
            count += 1
            print(books[x],books[y])
print(count)
```

Figure 6. Zach and Levi's final code.

In arriving at this final code, Zach also did a few fixes to make sure that the code was operating correctly. He first had to change the range to go from 0 to 12 instead of 1 to 13 since the indexing of the array starts at 0 and not 1. He then made the range for the y go from x+1 instead of starting at x so that it won't have a repeated book and it will just start at the next one. These practices highlight Zach's prior experience with coding. There are also examples of them utilizing the set of outcomes to obtain their final code. For example, they pay close attention to repeats like F1F2 and Zach utilizes a function of the code for reading the first letter to make sure that their set of outcomes does what they want it to do.

Zach and Levi's explanation of how the code works. In the excerpt below, Zach and Levi explain their work, and the explanations demonstrate how well that Zach understands how the code works. Notably, Levi also seems to agree and is able to articulate aspects of what the code is doing as well.

Levi: For X in range between 0 and 12 for Y and range X plus 1 to 12, and then, uh, that's just if that... if the F... if the letters don't match up.

Zach: Yeah.

Levi: Then you can continue [...].

Zach: If the letters aren't the same then they're the same... if they are the same then, they're like, the same language so I don't wanna count it.

Levi: Mm-hmm (affirmative).

Zach: And then the for range X plus 1 I'm not... I'm not gonna count, look behind me anymore, which you... it removes duplicates. So, I'm not gonna get R1 F1 and F1 R1 […]

Interviewer: Okay, good. And then what is making it so you don't get F1 R1 and also R1 F1?

Levi: The range on Y because once you... 'cause the range on Y is, it only looks farther down the line-[...] on the books, so...

Interviewer: I see, because you're just... you're in the same set. You have one set here. [...] So by looking at Y is only gonna be further down […]

Zach: Yep, so X picks like the fourth one and then my Y is only gonna be from the fifth on.

Relating the counting process to an expression. In terms of the model, we see the students connecting the code to expressions for the numerical total. In Figure 7, we see Zach wrote out the total of 27+20 = 47. To represent how the counting process was reflected in their code, they started with F being the first letter, and they noted that each of the "Fs" has 9. With 3 "Fs", there would be a total of 27. Then, looking at the "Rs", since we have already accounted for the pairs with French and Norwegian, each "R" is going to have 5 pairs. With 4 "Rs", there would be a total of 20. There are 47 combinations. In this way, they connected the process in their code to the outcomes being counted, and to mathematical expressions that reflected that code.

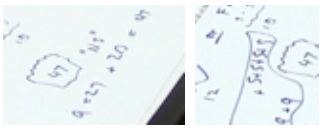

Figure 7. Zach's numerical solutions of how he counted the number of combinations of language books.

Discussion and Conclusion

By leveraging a couple of more advanced programming ideas (specifically, a sophisticated use of the range function and nested indexing), Zach arrived at a creative and efficient program that solved the Language Books problem in a single for loop. By using these techniques, Zach was able to account for making sure that they would not repeat the same language and ensured a complete list of desirable outcomes. We also highlight computational practices that Zach employed that suggest he drew upon his coding experience. Zach used the code to troubleshoot while making a prediction of what it would do. He wanted the code to specifically look at the first letter so that he could later compare the letters. To do this, he tested the code so that it would print out what he hoped was the first element of the code (this is reflected in Figure 5). We interpret that not only is the use of function itself using knowledge that an experienced coder would draw upon, but his testing of it to make sure that it does what he wants it to also highlights a practice of a more experienced programmer.

In comparing Taylor and Kirk's code and Zach and Levi's code, the overall structure of their sets of outcomes and final formula were different. Because Zach and Levi combined the code into one for loop, it forced French to be paired up with Spanish and Norwegian all at the same time, creating a group of where Spanish and Norwegian are almost like its own array. Meanwhile, Taylor's and Kirk's method forced French and Spanish and French and Norwegian to be completely separated groups. This is also reflected in their formulas, since Taylor and Kirk's group had three separate additions (3*4 + 3*5 + 5*4) while Zach and Levi's code had two separate additions (3*9+4*5). In all, they created the same result, but their Zach's experience as a programmer, they were able to see a different set of outcomes and formula/expressions.

In terms of components of the model of students' combinatorial thinking, both pairs of students were able to relate their counting processes with a structure of the set of outcomes and a corresponding mathematical expression. We note that their explanations suggest Zach and Levi understood the counting processes, and they ultimately expressed mathematical expressions that reflected the process of their creative, unconventional code. This highlights the fact that alternative approaches may meaningfully be connected to expressions that suggest a process.

In the undergraduate mathematics education literature, there is relatively little work on ways in which experienced programmers engage in mathematical tasks in computational settings. The episodes highlight ways that experienced coders may leverage certain advanced coding practices in their work. Both pairs successfully solved the problem, but the advanced techniques allowed for Zach to program the problem in one for loop, something Taylor and Kirk attempted but did not do. Zach's advanced coding practices emphasize why it is not surprising that Taylor and Kirk were not able to use a single for loop. Future research could explore whether there are mathematical implications for more advanced programmers, and subsequent studies could design tasks that could utilize certain computational tools that might not be available to novice coders.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1650943.

References

Buteau, C. & Muller, E. (2017). Assessment in undergraduate programming-based mathematics courses. *Digital Experiences in Mathematics Education, 3*, 97-114. doi:10.1007/s40751-016-0026-4

De Chenne, A. & Lockwood, E. (2022). A task to connect counting processes to lists of outcomes in combinatorics. *Journal of Mathematical Behavior*, *65*. https://doi.org/10.1016/j.jmathb.2021.100932

De Chenne, A. & Lockwood, E. (2023). A Case of a Computational Setting Facilitating Empirical Re-Conceptualization. In Cook, S., Katz, B. & Moore-Russo D. (Eds.). *Proceedings of the 25th Annual Conference on Research in Undergraduate Mathematics Education*. Omaha, NE. 494-501.

Geraniou, E., Faggiano, E., Medová, J. *et al.* Guest Editorial for Special Issue "Digital Experiences in University Mathematics Education: Advances and Expectations". *Int. J. Res. Undergrad. Math. Ed.* 10, 683–689 (2024). https://doi.org/10.1007/s40753-024-00260-4

Lockwood, E. (2013). A model of students' combinatorial thinking. *Journal of Mathematical Behavior, 32,* 251-265. doi:10.1016/j.jmathb.2013.02.00.

Lockwood, E. & De Chenne, A. (2020). Using conditional statements in Python to reason about sets of outcomes in combinatorial problems. *International Journal of Research in Undergraduate Mathematics Education, 6,* 303-346. doi:10.1007/s40753-019-00108-2

Lockwood, E. & Mørken, K. (2021). A call for research that explores relationships between computing and mathematical thinking and activity in RUME. *International Journal of Research in Undergraduate Mathematics Education, 7*(3), 404-416. doi: 10.1007/s40753-020-00129-2.

Papert, S. (1980). *Mindstorms: children, computers, and powerful ideas.* New York: Basic books.

Sinclair, N. & Patterson, M. (2018). The dynamic geometrisation of computer programming, *Mathematical Thinking and Learning, 20*(1), 54-74, doi:10.1080/10986065.2018.1403541

Soto, O., Siy, K. & Harel, G. (2022). Promoting a set-oriented way of thinking in a U.S. High School discrete mathematics class: a case study. *ZDM Mathematics Education,* 54, 809–827. https://doi.org/10.1007/s11858-022-01337-7

Weintrop, D., Beheshti, E., Horn, M., Orton, K. Jona, K., Trouille, L., & Wilensky, U. (2016). Defining computational thinking for mathematics and science classrooms. *Journal of Science and Education Technology, 25*, 127-147. Doi: 10.1007/s10956-015-0581-5.