PRECURSOR SKILLS TO WRITING CODE.

Laura Zavala
Medgar Evers College | CUNY
1650 Bedford Ave. | Brooklyn, NY 11225
718-270-6128 | rzgutierrez@mec.cuny.edu

Benito Mendoza
New York City College of Technology | CUNY
300 Jay Street | Brooklyn, NY 11201
718.260.5885 | bmendoza@citytech.cuny.edu

ABSTRACT

Practice in programming courses is most of the time equated with programming and more specifically with writing code. However, we believe that more emphasis needs to be put in the precursor skills to writing code. Specifically, that there are three phases that students should sequentially master in the process of learning computer programming: code comprehension, code manipulation, and code writing. Further, there are other companion skills, such as understanding technical documentation (i.e., APIs), that should also be stressed in all of the three phases and that are usually under-looked in CS1 courses. In this paper we present a study conducted in both, an introductory and an advanced programming courses in which we evaluate students in these competencies. The results obtained are in line with our intuition that code comprehension, code manipulation, and code writing are phases that students should sequentially master in the process of learning to write programs.

INTRODUCTION

The need for considerable practice in CS1 courses is indisputable and widely acknowledged. High failure rates abide in introductory computer programming courses (CS1) all over the world with a pass rate estimated to be around 67.7%. The literature abounds with research on pedagogies and innovative approaches for CS1 courses, specifically, active and collaborative learning approaches such as pair-programming, peer-lead instruction, flipped classrooms, and live coding [1][2][3][7][8][9]. The advantages of these innovative approaches cannot be dismissed. However, the emphasis in programming courses is mainly in writing code. This might be adequate for advanced, upper-level courses. However, for CS1 courses, the fact that code comprehension skills are a precursor to code writing skills should not be ignored; practice of the former should be ensured before asking students to do the latter.

Some practitioners have discussed the need for a reading before writing approach in CS1 courses [4][10][11][12]. The underlying idea is inspired in the observation that children learn to write after they have spent several years of reading (or rather, being read to) and speaking the language. By the time they are asked to write, they have been exposed to the syntax and semantics of the language, as well as different models of writing. The same underlying idea has been explored in other problem solving oriented disciplines. For example, Sweller and Cooper [5] compared the efficacy of a worked-examples approach versus a problem-solving approach. In their experiments, some students learnt certain algebraic processes by solving problems, while other students learnt the same algebraic processes by studying complete solutions to those problems (i.e., worked examples). They found that students who learned algebra with worked examples performed significantly better than those who were asked to solve problems instead.

The analogous reading before writing approach in CS1 courses is more of a reading before manipulating and manipulating before writing approach. Students should be able to not only read code before they can write code, but also manipulate code that is given to them (i.e., modify or use). Formally, there are three phases that students should sequentially master in the process of learning to write programs: code comprehension, code manipulation, and code writing. This paper presents the results of a study conducted at two urban colleges with the goal of corroborating this intuition.

Although it can be argued that code comprehension and manipulation work is covered in CS1 courses through sample solutions, design roadmaps, implementation hints, and starter code, the emphasis is, by far, on code writing. Providing students with all these resources does not really constitute practice on code comprehension and manipulation; it is not the main focus and students are rarely or never assessed on these competencies. In addition to these competencies, there are other skills, which we call companion skills that should also be stressed in all three phases, many of which are usually under-looked in CS1 courses. Namely, identifying programming constructs (variables, data types, expressions, function definitions, function calls, parameters, etc.), explaining code (being able to verbally explain what a piece of code does), understanding technical documentation (i.e., APIs), and refactoring existing code.

BACKGROUND

Linn and Clancy [12] emphasize the importance of learning patterns of program design. They propose the use of programming case studies to teach students program design skills. The authors reported significant improvement in students' design skills. In a similar effort, Kirschner, Sweller, and Clark [6] recommend providing students with worked examples and process worksheets, which should provide a description of the steps they should go through when solving the problem as well as hints or rules of thumb that may help to successfully complete each step.

In [4], an assessment developed to evaluate the programming competency of students that have completed their first one or two courses in computer science is presented. They established that many students do not know how to program at the conclusion of their introductory courses. In the search for a cause to this problem, Lister et al. [13] tested students from seven countries at the conclusion of their CS1 courses on two abilities: (i) predicting the outcome of a program, and (ii) completing a near-complete program. The authors noted that poor performance in one or both abilities might be an explanation for the fact that many students do not know how to program at the conclusion of their introductory courses. This work is closely related to that presented in this paper. An important difference is that they only test students on code comprehension skills and speculate on a relation of those skills to the ability of students to write code. The work presented in this paper examines that assumption given that students were tested on both, code comprehension skills as well as code writing skills of similar complexity.

PROBLEM FORMULATION

The high-level goal for the study presented in this paper was to collect initial data to shed light into the belief that (1) code comprehension, (2) code manipulation, and (3) code writing are phases that students should sequentially master in the process of learning to write programs. The specific goal was to answer the following questions:

a) Do (most or all) students that show proficiency in a given skill also possess proficiency in the skills that precede it in the sequence?
b) Do (most or all) students who show deficiency in, or lack of, a given skill, also lack the skills that follow it in the sequence?

METHODOLOGY

Students from both, an introductory and an advanced programming courses were evaluated on their code comprehension, code manipulation, and code writing competencies, as well as their companion skills. Code comprehension is the ability of students to understand a code fragment or a program. Ways to assess this skill include asking students to (i) select the piece of code, from a set of choices, that performs a specific task; (ii) describe what a program does, (iii) indicate what is the value returned by calling a function; (iv) indicate what a program would display on screen when it runs; (v) indicate what the value of one or more variables is after executing a program; and (vi) identify the part of a program where a specific action is carried. Code manipulation involves using or modifying existing code. The skill can be assessed with exercises where students are asked to: (i) complete a piece of code (either by writing the missing code, or choosing it out of a set of choices), (ii) write function calls to provided functions so that a specific result is obtained; (iii) build a program from a set of fragments of code, not all of which might be part of the solution; (iv) reorder a scrambled program (a.k.a. Parson's programming puzzles [13]). Code writing is the ability of students to write code for a given task. The default method of assessment is to provide students with the specification of a problem, for which they have to write a program.

Companion skills include: (i) identifying programming constructs (variables, data types, expressions, function definitions, function calls, parameters, etc.), (ii) explaining code (being able to verbally explain what a piece of code does), (iii) understanding technical documentation (i.e., APIs), and (iv) refactoring existing code.

Students in the introductory course were evaluated on conditional statements in the Python language. Students in the advanced course were evaluated on conditional statements, arrays, and loops. Additionally, students were also tested on their ability to identify elements of a program (terminology), use APIs, and refactor an existing program.

A person is eligible to be a US Representative who is at least 25 years old and has been a US citizen for at least 7 years.

The program below asks the user for his/her age and years of citizenship to display the corresponding message related to the eligibility for US Representative. The program has been scrambled, though. **Reorder the program so that it performs as described.**

```
    print("You are eligible to be US Representative.")
citizenship = int(input("For how many years have you
been US citizen? "))
else:
age = int(input("What is your age? "))
    print("You are NOT eligible to be US
Representative.")
if (age >= 25 and citizenship >= 7):
```

(a)

A person is eligible to be a US Senator who is at least 30 years old and has been a US citizen for at least 9 years.

Write a Python program that will ask the user to enter his/her age and length of citizenship. Then, it will print the appropriate message, one of the following:

- You are eligible for the Senate
- You are NOT eligible for the Senate.

(b)

FIGURE 1. QUESTIONS GIVEN TO THE INTRODUCTORY PROGRAMMING COURSE: (A) CODE MANIPULATION; (B) CODE WRITING

What is the output of the following program? In other words, what is printed to the screen when you run it?

Assume the file *numbers.txt* contains the numbers 1 through 30

```
import java.util.Scanner;
import java.io.File;
try {
    Scanner reader = new Scanner(new
                  File("numbers.txt"));
    while (reader.hasNext()) {
        int foo = reader.nextInt();
        if ((foo % 2)>0)
            System.out.print(foo + "\t");
    }
} catch (IOException e) {
    System.out.print("Error with input file")
}
```

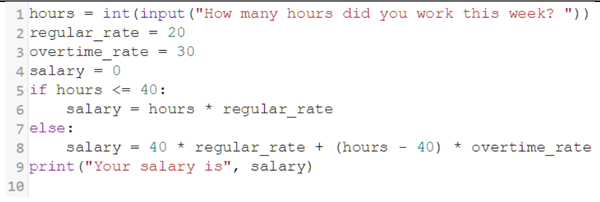
A file called *cs355students.txt* contains a list of students and their corresponding GPA's. Each line in the file contains the name of the student followed by his/her GPA.

Write a Java program that will print the best students from that list (the ones with a high GPA). Your program should read the entries in the file and print to screen the names of only those students with a GPA greater than or equal to 3.0

(a)	(b)

FIGURE 2. QUESTIONS GIVEN TO THE ADVANCED PROGRAMMING COURSE: (A) CODE COMPREHENSION; (B) CODE WRITING

The assessment was administered by phases: first, the code writing and use of APIs exercises, then the code manipulation and refactoring exercises, and at the end, the code reading and terminology questions. Students worked on each phase one at a time and had no access to the questions on the other sections. Figures 1 and 2 show some of the questions given to the introductory programming course and the advanced programming course, respectively. Figure 3 shows some of the companion skills questions.



```
 1 hours = int(input("How many hours did you work this week? "))
 2 regular_rate = 20
 3 overtime_rate = 30
 4 salary = 0
 5 if hours <= 40:
 6     salary = hours * regular_rate
 7 else:
 8     salary = 40 * regular_rate + (hours - 40) * overtime_rate
 9 print("Your salary is", salary)
10
```

From the code above provide the line number(s) containing the program constructs indicated. If there is more than one line, please separate the numbers with commas (i.e., 1, 2, 3).

i) a Boolean expression
ii) an arithmetic expression
iii) assignment statement with a literal

(a)

Assume a text file called *words.txt* contains a string on each line. Write a piece of code to read the information on the file and print to screen only the words that are palindromes. A palindrome is a word that reads the same backward or forward. For example: stressed, rewarder, noon, civic, radar.

`public String readLine()`

A member function of `BufferedReader`, reads a line of text and returns a `String` containing the contents of the line, or null if the end of the stream has been reached

`public int length()`

A member function of `String`, returns the length of a string, that is, the number of characters in the string.

`public StringBuilder reverse()`

This method returns a reference to this object with its character sequence reversed.

`StringBuilder(String str)`

Constructs a string builder initialized to the contents of the specified string.

(b)

FIGURE 3. EXAMPLES OF COMPANION SKILLS QUESTIONS: (A) IDENTIFYING PROGRAMMING CONSTRUCTS AND (B) UNDERSTANDING TECHNICAL DOCUMENTATION (I.E., APIS).

RESULTS

The stacked line charts in Figures 4 and 5, in which the cumulative scores by level are plotted, show that code reading is the most dominant skill, followed by code manipulation, while code writing is the weakest one. The same pattern is observed in both, the introductory and the advanced courses.

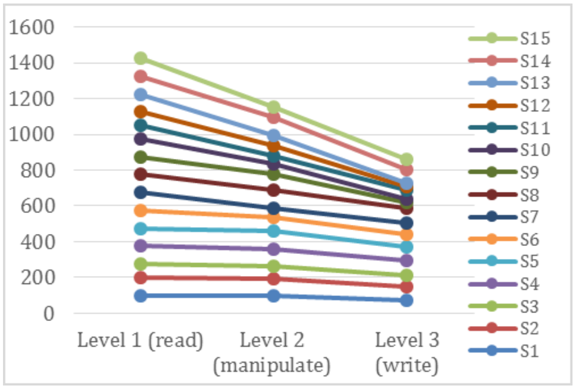


FIGURE 4. INTRODUCTORY PROGRAMMING COURSE CUMULATIVE SCORES BY LEVEL

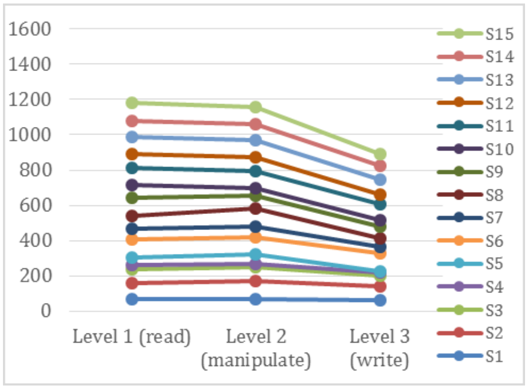


FIGURE 5. ADVANCED PROGRAMMING COURSE CUMULATIVE SCORES BY LEVEL

CONCLUSIONS

Based on the expressed need of a reading before writing approach for CS1 courses, an analogous approach has been presented that is more of a reading before manipulating and manipulating before writing approach. Students should be able to not only read code before they can write code, but also they should be able to manipulate code that is given to them. To shed light on the validity of this claim, a small-scale study was conducted where students' code comprehension, code manipulation, and code writing skills in two different colleges were evaluated. The results obtained are in line with the original premise that code comprehension, code manipulation, and code writing are phases that students should sequentially master in the process of learning computer programming. The experiment was conducted with computer science students in both, an introductory and an advance course at different institutions. The fact that the same pattern is observed in both further supports our claims.

REFERENCES

[1] Leo Porter and Beth Simon, "Retaining nearly one-third more majors with a trio of instructional best practices in CS1," *in Proceedings. ACM Computer Science Education (SIGCSE '13). 44th Annual Technical Symposium* (2013): 165-170, ACM.

[2] A. Robins, J. Rountree, and N. Rountree. "Learning and teaching programming: A literature review," *Computer Science Education* 13, no. 2 (2003): 137–172.

[3] A. Sanwar, "Effective teaching pedagogies for undergraduate computer science," *Mathematics and Computer Education* 39, no. 3 (2005): 243–257.

[4] McCracken, M., V. Almstrum, D. Diaz, M. Guzdial, D. Hagen, Y. Kolikant, C. Laxer, L. Thomas, I. Utting, T. Wilusz, "A Multi-National, "Multi-Institutional Study of Assessment of Programming Skills of First- year CS Students," *SIGCSE Bulletin* 33, no. 4 (2001): 125- 140.

[5] Sweller, J., & Cooper, G. A., "The use of worked examples as a substitute for problem solving in learning algebra," *Cognition and Instruction* 2, no. 1 (1985): 59–89.

[6] Kirschner, P.A., Sweller, J., and Clark, R.E., "Why minimal guidance during instruction does not work: an analysis of the failure of constructivist, discovery, problem-based, experiential, and inquiry-based teaching," *Educational Psychologist* 41, no. 2 (2006): 75-86.

[7] Judith D. Wilson, Nathan Hoskin, and John T. Nosek, "The benefits of collaboration for student programmers," in Proceedings. *ACM Computer Science Education (SIGCSE '93). 24th Annual Technical Symposium* (1993): 165-170, ACM.

[8] Henry M. Walker. 2011, "A lab-based approach for introductory computing that emphasizes collaboration," *in Proceedings. Computer Science Education Research Conference* (2011): 21-31, Open Universiteit, Heerlen.

[9] Marc J. Rubin. 2013, "The effectiveness of live-coding to teach introductory programming," *in Proceedings. ACM Computer Science Education (SIGCSE '13), 44th Annual Technical Symposium* (2013): 651-656, ACM.

[10] Mark Guzdial and Judy Robertson, "Too much programming too soon?," *Communications of the ACM* 53, no. 3 (March 2010): 10-11, ACM.

[11] Mark Guzdial. 2015, "What's the best way to teach computer science to beginners?," *Communications of the ACM* 58, no. 2 (January 2015): 12-13, ACM.

[12] Marcia C. Linn and Michael J. Clancy. 1992, "The case for case studies of programming problems," *Communications of the ACM* 35, no. 3 (March 1992): 121-132, ACM.

[13] Dale Parsons and Patricia Haden, "Parson's programming puzzles: a fun and effective learning tool for first programming courses," *in Proceedings of the 8th Australasian Conference on Computing Education (ACE '06)*, Denise Tolhurst and Samuel Mann (Eds.), Vol. 52 (2006): 157-163, Australian Computer Society, Inc., Darlinghurst, Australia, Australia.