Journal of Information Systems Education

Volume 31
Issue 2
Spring 2020

Teaching Tip
Teaching Programming to the Post-Millennial Generation: Pedagogic Considerations for an IS Course

Madhav Sharma, David Biros, Surya Ayyalasomayajula, and Nikunj Dalal

Recommended Citation: Sharma, M., Biros, D., Ayyalasomayajula, S., & Dalal, N. (2020). Teaching Tip: Teaching Programming to the Post-Millennial Generation: Pedagogic Considerations for an IS Course. *Journal of Information Systems Education,* 31(2), 96-105.

Article Link: http://jise.org/Volume31/n2/JISEv31n2p96.html

Initial Submission:	2 January 2019
Accepted:	3 October 2019
Abstract Posted Online:	3 March 2020
Published:	4 June 2020

ISSN: 2574-3872 (Online) 1055-3096 (Print)

Teaching Tip
Teaching Programming to the Post-Millennial Generation: Pedagogic Considerations for an IS Course

Madhav Sharma
David Biros
Surya Ayyalasomayajula
Nikunj Dalal
Department of Management Science and Information Systems
Spears School of Business
Oklahoma State University
Stillwater, OK 74075, USA
madhav.sharma@okstate.edu, david.biros@okstate.edu, surya.ayyalasomayajula@okstate.edu, nik@okstate.edu

ABSTRACT

Teaching introductory programming to IS students is challenging. The educational, technological, demographic, and cultural landscape has changed dramatically in recent years. The post-millennial generation has different needs and expectations in an era of open resources. Learning to program is perceived as difficult, teaching approaches are diverse, and there is little research on what works best. In this paper, we share our experiences in developing, testing, and implementing a new design for teaching introductory IS programming at the undergraduate level. We describe pedagogic considerations and present teaching tips for a blended course that combines best practices with experimentation. Our approach recognizes the changing nature of the student body, the needs of an IS major in the current environment, and the worldwide shift in education from instructor-centered to student-centered learning.

Keywords: Teaching tip, Introductory programming, Introductory course, Instructional pedagogy, Blended learning

1. INTRODUCTION

An introductory programming course is an important foundation for Information Systems (IS) students. As the first exposure to programming for IS majors and a prerequisite to other advanced courses, it is important that this course be an effective learning experience to lay the groundwork for the future.

Historically, educators have found teaching programming to be a challenging experience. Learning to program is perceived as "difficult" even by students enrolled in IT-related majors (Ali and Smith, 2014). There is considerable diversity in teaching approaches and in the presentation of various textbooks. However, there is little research about effective pedagogies for teaching programming, let alone the question of whether IS students may have different needs from computer science students. A widely-cited survey on the curriculum, pedagogy, and languages for teaching introductory programming (Pears et al., 2007, p. 204) concludes "that despite the large volume of literature in this area, there is little systematic evidence to support any particular approach."

Compounding these formidable challenges is that the educational, technological, demographic, and cultural landscape has changed dramatically in recent years. One big change is the new generation of students, also called Generation Z, loosely defined as people born from the mid-1990s to the early 2000s, who make up 25% of the U.S. population, a larger cohort than the Baby Boomers or Millennials (Dill, 2015). We have to question whether classical methods of teaching are still as relevant today for the post-millennial generation of students, who are essentially digital natives living in a wired world of gadgets and open resources.

In this paper, we share our experiences in developing, testing, and implementing a new design for teaching introductory IS programming at the undergraduate level. We designed and delivered an introductory programming course in the IS curriculum at a comprehensive state university in Fall 2018 based on several considerations, among which are the changing nature of the student body, the needs of an IS major in the current environment, and the worldwide shift in education from instructor-centered to student-centered learning. We made significant changes in the curriculum to

adjust the course to student behaviors so that students stayed motivated and learned core concepts of programming. From our vantage point as educators teaching undergraduate business students in an IS curriculum, we started with a relatively clean slate design, posing some fundamental questions: What would be the differences between classical teaching methods and methods for teaching post-millennials? What should be the differences, if any, between teaching IS students and Computer Science students? How would we teach introductory business programming differently from advanced programming? How much of the course should focus on teaching a programming language versus teaching more general computational thinking skills? Upon reviewing the research, it became increasingly clear that there were no conclusive answers to these questions, and what was needed was a fresh approach guided by available evidence of best practices combined with ample experimentation.

2. THEORETICAL BACKGROUND

Computational thinking, a term envisioned by Wing (2006), has been proposed as a fundamental skill for problem-solving in the new age of technology. It is broadly defined as a set of cognitive skills and problem-solving processes that includes but is not limited to decomposition, pattern recognition, data representation, abstraction, generalization, and algorithms. While this type of learning is typically first introduced at the K-12 stage, once the student is in college, the introductory programming course is important as a gateway to imparting, reinforcing, and strengthening computational thinking skills using symbolic representation, iteration, decision structures, and logical, algorithmic operations. Emphasis on computational thinking and problem-solving is somewhat different from programming instruction focused mainly on the mechanics of a language. How best should computational thinking skills be imparted at the college level?

With the help of a committee of seasoned instructors and practitioners, we came up with four learning goals and corresponding objectives. First, students should have a basic understanding and appreciation for programming as a problem-solving approach. Second, students should understand the logic of common programming constructs and structures used to build programs such as if-then-else, for-loop, while-loop, and others. Third, students should be able to write and debug the procedural code with moderate proficiency. Fourth, students should have an introductory knowledge of the concepts of object-oriented programming. We ended up having to balance the theoretical overarching pedagogic goals of developing computational and algorithmic thinking skills in the students with the practical need to introduce a vocationally useful programming language and environment.

In the traditional model of classroom instruction, the lecture is the main event of the class, and the educator is the primary disseminator of knowledge. "Homework" assignments are done out of class independently by students. In the modern age, where learning resources are readily available and digitally-plugged in students are used to looking online for answers, the flipped classroom model reverses the traditional approach by delivering lecture-type content online before class and by placing activities such as discussion and exercises in the classroom. The flipped class model is designed to be learner-centered, where students actively take responsibility for their learning. In the context of teaching programming, Mok (2014) has implemented a flipped class model and observed that students found the experience effective, helpful, engaging, and empowering.

For this introductory programming class, the instructors recognized the need for close guidance to shape students' programming ability offered by the traditional model of the classroom and exposure to the open resources available online offered by the flipped classroom. Between the two ends of a classical lecturer-centered model and a completely flipped classroom model, one can envision hybrid courses that fall at various places on a "blended" spectrum where online lectures replace a portion of traditional face-to-face instruction. The instructors also saw a high variance in student skills in the initial classes and, hence, had to make adjustments to fulfill the student needs. Our design choice of where to fall on the spectrum is discussed in the next section in the context of various pedagogic considerations and adjustments made to the curriculum to accommodate skill variance.

3. TEACHING TIPS

In this section, we present teaching tips based on several pedagogic considerations. We started this class with the flipped classroom approach, where we shared some videos related to core concepts before the class and expected the students to view them. The classroom activity contained an in-class exercise. In the first class, we asked the students to submit a small introduction as an assignment where they were asked about their exposure to computer programming. We noticed a high variance in exposure to programming across students. As the class proceeded, we noticed that the gap between student skills continued to increase. A few students were able to follow the videos and reproduce what they learned in the in-class exercise with the instructor's supervision. However, other students struggled to do the same. In order to assess this issue, we conducted a formative survey to get some feedback from students about the class. According to those results, we made certain adjustments to our curriculum (as shown in the following tips). Some of the considerations are common sense and obvious, but we discuss them here in a more specific and subtler context of teaching introductory programming to IS undergraduate students.

3.1 Aligning Instruction to Student Expectations

Our students were largely post-millennial digital natives who had grown up with smartphones and computers. Given the ease-of-use of graphical user interfaces (GUI) and touch screens, some students saw programming as an unnecessarily complicated and tedious use of computers. As instructors, we had to prepare the student to realize the importance of programming in the development of practical applications and their potential role as a contributor to the process.

A challenge that we faced in our class was the varying levels of programming knowledge at the beginning of the semester. Based on the student introduction (submitted as an assignment), we identified three levels of programming expertise: the absolute beginners or novices, the continuing learners, and the budding experts. The absolute beginners were students who used computers for mostly hedonic purposes such

as gaming, social media, and web-browsing. These students had used spreadsheets (Excel) but had little or no exposure to any programming language. The continuing learners were students who had been exposed to a programming language in middle/high school. These students were enthusiastic about programming. The budding experts were students who were somewhat experienced in programming. In the first few weeks of the class, the budding experts were able to complete the in-class exercises with ease. As we could monitor their activity on our learning management system, we could see that these students were able to do the initial exercises without watching the videos. The continuing learners were able to finish these exercises, albeit with some effort. The absolute beginners were not able to do these exercises and were not motivated to watch the videos either. Observing these changes, we decided to conduct a formative survey where we asked open-ended questions gauging students' perceptions of these classes. We got a mixed response as exhibited by responses from three different students:

- "To be honest, I feel like the course is moving too slow."
- "More time and more instruction on in-class exercises."
- "It is a me thing, I need to learn, and the videos are not so helpful."

Based on these responses, and in order to accommodate the students' high skill variance, we decided to change the curriculum from a flipped class approach. We needed a curriculum architecture that could fulfill the following two needs: first, the need for close step-by-step guidance for the absolute beginners and continuing learners; second, the need to provide students with exposure to various online resources and the ability to pick up other languages. These needs made us consider a blended teaching approach.

3.2 Consider a Blended Teaching Approach

Given widely different levels of programming knowledge and the nature of the students, a classical lecture-based format delivering identical instruction to all was not seen as appropriate. A flipped classroom was considered due to its merits, such as pacing and effectiveness with courses requiring information literacy (Davies, Dea, and Ball, 2013; Mok 2014). However, findings from our initial formative survey (at the end of week 4) showed that only 38% of students were watching the lecture videos shared before class. As a result, students often found practice-based instruction to be fast and hard to follow.

Because of the student behaviors, we adopted a blended instructional approach (as shown in Figure 1) at the end of week 4. The blended instructional approach combines the use of online videos to be watched before class, with a condensed lecture that includes the concepts from the videos and clarification, followed by in-class exercises in a face-to-face class, and finally ending with a weekly homework assignment.

The class sessions also involved the experimental use of best practices, such as live-coding, the use of authentic tasks, peer instruction, and a semi-structured project to normalize the skills gap between students. More specifically, the blended teaching approach we used was as follows. Students were asked to view the video for a concept (such as data types or loops) before coming to class. During class, instructors explained the concept and responded to questions. Next, they explained an authentic task or problem which was followed by a discussion of a solution in the form of an algorithm using inputs from class participation. After this, the instructor coded the program live while students followed. The instructor took the time to code and explained the reasoning behind the code with profuse commenting. An in-class exercise based on the same concept and code skeleton was then given to students to do on their own. The goal of this exercise was to make students think about the concepts taught in class in a slightly different programming context. These exercises were less complex than the code covered by the instructor but required significant modifications

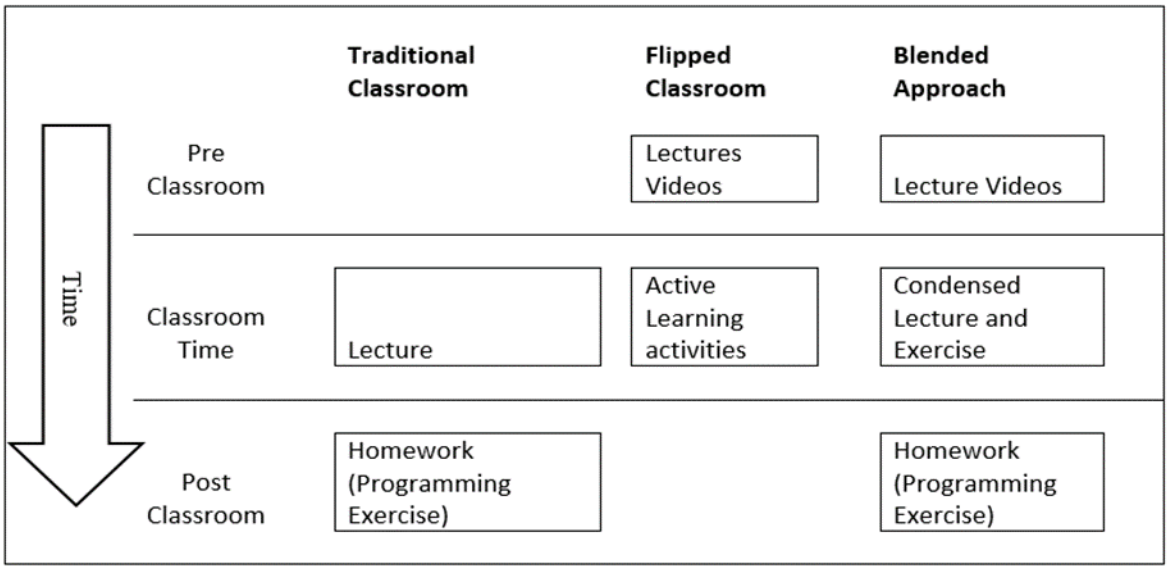


Figure 1. A Blended Instructional Approach

for the proper output. The instructors went around the class to help students debug their programs and ask any questions they might have. Finally, students were given a weekly homework assignment which built upon the cumulative work done so that students could continually practice all the constructs they learned in previous weeks, in contrast to in-class exercises which are specific to a single construct. Thus, in a typical week, the student was involved in working with three programs independently: two in-class exercises and a homework assignment.

This approach turned out to be more effective. The students who watched the videos were able to practice the code in the class, and the students who were not motivated to watch the videos before the class could catch up with instructions very well. The instructors observed that the complaints about instruction being too fast were reduced, and students were able to follow the adjusted pace of the class. The relatively more complex homework programs served as a practice for students rather than a first-time attempt. Students also responded more positively to this structure; for example, one of the responses was: "The Homework assignments were a good comprehensive way to utilize what we learned during the class period."

3.3 Integrating Existing Sources from the Internet

The basic structure of the course was adapted from a tried-and-tested source. Initially, various textbooks and online resources were considered during the course redesign process. We found helpful ideas in many textbooks, but they did not align with our goals. Books about programming are version-specific, leaned heavily on granularities of the language, and adopting them as official textbooks risked losing the critical and computational thinking dimensions. Our goal, on the other hand, was to introduce students to programming as a thinking process such that they can read and understand programs, customize code, build algorithms, and execute tasks using any computer language or platform.

Given the blended course environment, we organized the flow of concepts according to the acclaimed Microsoft Visual Academy's (MVA) C# for Absolute Beginners course by Bob Tabor (https://mva.microsoft.com/en-us/training-courses/c-fundamentals-for-absolute-beginners-16169). According to the course creator, this free open online course was in its sixth iteration since 2005 and had incorporated feedback from thousands of students. The course website consisted of short 15 to 30-minute videos laid out in sequence, transcripts of the videos, short quizzes, program resources, and discussion forums. According to the instructors, the videos seemed to have an organic build-up of concepts on an as-needed basis. For example, it makes sense to not do all the loop constructs all at once, unlike what textbooks typically do. Rather, introduce students to one kind of loop construct (FOR loop) for a specific problem, let them chew on it and digest it, and then two weeks later, introduce them to the DO/WHILE loop, showing why it is necessary for a slightly different problem.

With the rise of massive open online courses (MOOCs), such as MVA, Udemy, and Coursera, and ever-expanding crowdsourced programming communities like StackOverflow and GitHub, students have many free resources at their disposal to learn any programming language of their choice (Hew and Cheung, 2014; Zagalsky et al., 2015). Our course's organization aimed at not only introducing students to programming but also at making them aware of these resources and how to use them. We shared the videos on the course website and asked students to view them before covering the same topics in class using simple slides and experiential in-class exercises. The videos provided detailed explanations on code blocks targeted to novices, letting instructors focus in class on underlying concepts of the code blocks and how they can be used to solve problems with critical thinking. The MVA videos provided a tried and tested de-facto structure for us to cover all major code blocks in a sequence consistent with our learning goals and objectives. The videos could be re-watched, and students had the opportunity to consult additional resources and participate in a larger discussion forum. Besides MVA, we also shared selective open-source content such as textbook chapters, videos, and web-links for students' reference.

Using these resources had two major desirable outcomes. First, students did not have to buy a textbook. Textbooks on programming are comprehensive and can offer good guidance to students if used properly. However, research has repeatedly recognized motivating students to read textbooks as challenging (Murden and Gillespie, 1997; Burchfield and Sappington, 2000). Textbooks for C# programming cost $50 and up, and, thus, were considered not worth the cost. Second, students got exposure to a breadth of online resources for programming. Students in the IS curriculum are expected to have a working knowledge of programming. MOOCs and online resources are becoming an integral part of the programmer's ecosystem. Understanding how to learn about new languages, libraries, and customizing pre-made codes will help students immensely in their careers.

3.4 Communicate Frequently and Offer Optional Lab Hours

Although obvious, this teaching tip has particular applicability for undergraduate programming students enrolled in a hands-on, blended course. We provided clear and frequent communication using multiple channels. Students were contacted and kept informed about the dates of exams, assignments, and lab and office hours via classroom announcements, e-mail, and the class learning management system (LMS).

Students can lose interest if they have trouble running a program and experience a lack of opportunities to get help in keeping up with the rest of the class. Instructors were able to interact one-on-one with at least 80% of the students who asked for help. Though students were strongly encouraged to ask questions using email or set up additional appointments if necessary, some students were reluctant to ask questions to the instructor directly.

From week 4, adjusting to student behaviors, the instructors arranged for the availability of optional lab hours that students could come to and work on their homework, project, or in-class exercise in case they needed to do so. These lab hours were completely optional for students, and at least one instructor was around to help students debug programs or answer any questions they had. Though they were intended to be time for extra help, the instructors observed a steady increase in the attendance of students in these lab hours. In the initial four weeks, only 10-15% of students showed up; after that point, the attendance grew to about 75% of the class. Students often came in groups with similar problems in their code. Such students

benefitted from receiving help from peers, a process that was encouraged by the instructors. Working in groups is an important part of programming experience. In the industry, where the students are expected to work later in their careers, seeking help from peers in debugging programs is deemed essential (Vidgen and Wang, 2009; Porter et al., 2013). This type of peer help was a win-win situation as it gave knowledgeable students a chance to participate in class (after their programs were completed) and helped them improve their understanding of the concepts. As the following student response shows, the lab hours were also found to be very effective by the students: "The work hours on Friday were very helpful to me because it gave me a chance to catch up on any [In-Class exercise] that I didn't get done that week and also helped me with the HW."

3.5 Use Authentic Tasks

Students must see programming as a useful tool for problem-solving rather than as a series of tedious tasks. This problem-solving focus is a recognition that students' understanding of programming at the introductory level should be logic-oriented and not dependent on any specific programming language. Early in the semester, we discussed a theoretical problem-solving model in class – showing students why and how programming was essentially a problem-solving process, and how it fits into the overall context of the systems development life cycle (SDLC).

One of the recognized practices used in this class was giving students authentic problems to solve (Falkner and Palmer, 2009; Thomas, Ge, and Greene, 2011). Authentic tasks are problems with real-world relevance with which students are familiar and can devise a solution with minimal explanation (Herrington, 2006). We designed in-class exercises for every class session and a homework assignment for every week based on the logical concept covered in class during the week. The in-class and homework exercises were developed from familiar practical problems. Recurring themes included: calculating GPA (as shown in Appendix 1), planning supplies for a get-together, making payments in a grocery store, and toll booth payments. Students solved these problems using the code blocks they were learning. The problem complexity grew with their knowledge of advanced concepts and code blocks. For example, students were introduced to a common problem, such as calculating their GPA, in the early weeks. An earlier solution to this problem involved the use of the 'if statement.' The same problem was continued later with solutions based on arrays, for-loop, and while-loop. The students were able to see how the same problem can be solved in different ways.

We provided students with several practical problems in their in-class exercises and homework over the semester. The first part of every homework assignment was about building an algorithm to solve the given problem. The algorithm was implemented using comment-first coding (Sengupta, 2009), a scaffolding strategy in which the programming logic is specified via plain English (or any other natural language) inside comments. Once the logic is completely specified, the code is then written using those comments. This approach creates a necessary separation between the thinking needed for programming logic and the mechanics of writing code in a specific language (Barr and Stephenson, 2011).

3.6 Provide a Semi-Structured Project

Continuing the theme of challenging students with algorithmic thinking problems, the second half of the semester involved work on a semi-structured project. We define a semi-structured project as a project where we assign a broad problem to students to solve using programming without giving them specifics about deliverables. The students were familiar with most code blocks used in C# by this point in time. The project involved a practical problem scenario (not simply a task) with which students were familiar – coming up with a system to manage a parking lot on campus. The students were tasked to observe the parking lot, understand the problem in-depth, and specify issues. They were further asked to separate the issues resolvable by programming solutions. In the next phase, they were tasked to propose logical designs to address those issues. In the final phase, they had to develop a small application in C# to implement the logical design. To better manage and understand the development process, the project was broken down into four deliverables, each corresponding to a phase. For the final deliverable, students could employ the instructor's help in debugging their programs and improving their application. The use of a semi-structured project, where some components were left to the students' initiative, helped them to create unique applications and appreciate creativity in the programming process.

According to the first and second deliverables, the instructors suggested potential ways students can execute their proposed solutions using programming. The instructors observed that students got increasingly engaged in this project. As a student noted in our survey: "I think the project we are working on right now has been very effective in learning different aspects of programming."

3.7 Use Live Coding

Live coding is defined as "the process of designing and implementing a project (involving programming) in front of the class during the lecture period" (Paxton, 2002, p. 52). It is a renowned best practice that was adopted for this class to slow down the teaching and focus on the problem-solving aspects. Students, though familiar with typing for short sentences, have variable typing speeds. Live-coding gives ample time for students to catch up with the instructor in writing the code. Additionally, the process of live-coding was also able to generate significant classroom participation. Students more knowledgeable in programming were able to provide some peer help to others during live-coding sessions. Many instructors have used this technique for teaching programming (Paxton, 2002; Rubin, 2013). We can confirm that it is extremely effective by the following student responses:

- "I really liked when the days when we were given examples for the problems and were able to type the code with (the instructor)."
- "I think the best part of this class was that we had computers in front of us to follow along with programming as the lecturer was doing it on the projector."

4. EFFECTIVENESS

Looking at effectiveness, we have to consider two factors: the course design and its implementation. The two factors need to be assessed separately because what, in principle, is a good course component may still suffer from a weak implementation. Based on our experiences, the course design was found to be operationally feasible, robust, flexible, evolutionary, and drawing upon best practices and the needs of the students. This was borne out by data collected from two anonymous surveys: a formative survey at the end of the fourth week of instruction and a survey at the end of the semester. In terms of implementation, our efforts were largely successful though we faced some challenges. Considering the exploratory design and evolving adaptive delivery of instruction, a pre-post survey is not appropriate to analyze our overall approach. However, we can make several observations based on our experiences and student feedback obtained through anonymous surveys.

Students tend to have an expectation from every class based on their peers' prior experiences and information from their advisors. Due to our modifications to the course, students took time to adapt to the class. Though most key concepts were covered in the lecture, they were expected to watch the videos in advance. In weeks leading up to the first survey, students were introduced to coding in IDE, displaying and inputting messages and values using different data types and the 'if statement.' The first formative survey suggested over 80% of students (n=41) found the in-class exercises and homework assignments useful in understanding problem solving and programming with C#, which was an encouraging signal for the two main components of the course. Responses on some questions were mixed, suggesting skill variance among students. For example, when it came to the pace of instruction, two responses were:

- "Slowing (sic) down during instruction, sometimes it goes too fast, and most of us cannot keep up."
- "To be honest I feel like the course is moving too slow."

Based on the early formative survey, we made some changes. For example, issues faced by students with the speed and format of live-coding were addressed by fine-tuning the pace and form. Student feedback on live coding and exercises were two parameters to assess effectiveness. The students practiced three to five problems a week in building logic, writing code, and executing code. Two of the programs were examples that instructors coded live along with an accompanying presentation of concepts. The other three programs were included in the in-class exercises and homework. The structure of the class enabled us to strategically increase the complexity of programs and employ concepts that build on each other. Students were autonomously able to build logic and write the initial skeleton of codes. The instructors and their peers helped them decode it for execution. Students gradually warmed up to these exercises as reflected in students' comments such as:

- "The [In-class exercises] I found effective because they gave us practical application of what we are learning in the course. The homework assignments were a good comprehensive way to utilize what we learned during the class period."
- "The homework assignments were also extremely effective in my opinion. They built upon what you were taught and created with the [In-class exercises] and increased the understanding of what was taught."

Although the course design was strong, the implementation was not without its challenges. First, not all students watched the MVA videos regularly in advance, despite our urging. While the MVA videos were informative, some of the later videos discussed granularities of Microsoft's libraries and the .NET framework which were not directly relevant to the goals of the course. Students had to learn what to focus on while watching videos. We addressed this aspect by selectively showing portions of a few videos in the first week, though the videos were used to complement the class sessions and never as a replacement. Also, we provided alternative, supplementary options such as references to specific book chapters. The second challenge was an uneven attendance after the first half of the course. We believe this was a general trend with many other classes and it was due to some forces beyond our control. A potential solution to counter this problem would be to allocate a part of the grade to attendance.

This introductory programming class was deemed to be one of the early steps in shaping the students into well-rounded IS practitioners. Our objective for this class was to introduce students to programming as a problem-solving process so that they can read and understand code, build algorithms, and write programs for basic problems. For testing a student's ability to read and understand code, a test for recognition is preferable (Simkin and Kuechler, 2005). The midterm exam constituted 25 multiple-choice questions. As the students gained more hands-on experience, we decided to test them on their ability to build algorithms and write code. The final exam constituted 15 multiple-choice questions and a code writing question. Consistent with the findings of Kuechler and Simkin (2003), we found that the student scores on multiple-choice questions and code writing were correlated. The tests constituted slightly less than one-third of their total grade. Besides tests, continuous assessment took place in the form of iterative in-class and homework exercises. These in-class and homework exercises contained two components: a thinking component and a coding component. The thinking component usually constituted writing an algorithm for a task or defining a concept in their own words. We believe the in-class exercises, homework, project, and exams collectively were able to test the logical understanding of students along with their basic programming skills.

Though there has been considerable debate over the relevance and requirement of programming in an IS program (Saulnier and White, 2011; Bell, Mills, and Fadel, 2013), the industry expects IS graduates to have the ability to solve problems using computational thinking and comprehend code to some degree (Wilkerson, 2012). Through this method of course delivery, we were able to teach students how programming can be used to solve authentic everyday problems. In doing so, we also noticed that students understood

the need for learning programming. In our second formative survey at the end of the semester, 90% of the students agreed that programming is a useful tool for problem-solving.

5. CONCLUSIONS

Our goal was to design and deliver an introductory programming course suitable for the post-millennial generation of undergraduate IS students. Toward this end, we have described an innovative blended course design and its implementation that combines best practices with experimentation. The design is flexible in how it may be implemented, keeping in mind the needs of a specific student body. Given the paucity of research on effective pedagogy to teach programming and the different needs and the differing expectations of post-millennials in an era of open resources and changing technologies, we believe the pedagogical considerations and teaching tips described in this article will be helpful to course designers and instructors of programming in undergraduate IS curricula.

6. REFERENCES

Ali, A. & Smith, D. (2014). Teaching an Introductory Programming Language in a General Education Course. *Journal of Information Technology Education: Innovations in Practice,* 13, 57-67.

Barr, V. & Stephenson, C. (2011). Bringing Computational Thinking to K-12: What is Involved and What is the Role of the Computer Science Education Community? *Inroads*, 2(1), 48-54.

Bell, C., Mills, R., & Fadel, K. (2013). An Analysis of Undergraduate Information Systems Curricula: Adoption of the IS 2010 Curriculum Guidelines. *Communications of the Association for Information Systems*, 32(1), 73-94.

Burchfield, C. M. & Sappington, J. (2000). Compliance with Required Reading Assignments. *Teaching of Psychology*, 27(1), 58-60.

Davies, R. S., Dean, D. L., & Ball, N. (2013). Flipping the Classroom and Instructional Technology Integration in a College-Level Information Systems Spreadsheet Course. *Educational Technology Research and Development*, 61(4), 563-580.

Dill, K. (2015). 7 Things Employers Should Know About The Gen Z Workforce. *Forbes*. Retrieved May 27, 2020, from https://www.forbes.com/sites/kathryndill/2015/11/06/7-things-employers-should-know-about-the-gen-z-workforce/#20de51f0fad7.

Falkner, K. & Palmer, E. (2009). Developing Authentic Problem-Solving Skills in Introductory Computing Classes. *ACM SIGCSE Bulletin,* 41(1), 4-8.

Herrington, J. (2006). Authentic E-Learning in Higher Education: Design Principles for Authentic Learning Environments and Tasks. In *E-Learn: World Conference on E-Learning in Corporate, Government, Healthcare, and Higher Education* (3164-3173).

Hew, K. F. & CheunG, W. S. (2014). Students' and Instructors' Use of Massive Open Online Courses (MOOCs): Motivations and Challenges. *Educational Research Review*, 12, 45-58.

Kuechler, W. L. & Simkin, M. G. (2003). How Well do Multiple Choice Tests Evaluate Student Understanding in Computer Programming Classes? *Journal of Information Systems Education*, 14(4), 389-400.

Mok, H. N. (2014). Teaching Tip: The Flipped Classroom. *Journal of Information Systems Education*, 25(1), 7-11.

Murden, T. & Gillespie, C. S. (1997). The Role of Textbooks and Reading in Content Area Classrooms: What are Teachers and Students Saying. In Linek, W. & Sturtevant, E. G. (Eds.), *Exploring Literacy* (pp. 87-96). College Reading Association.

Paxton, J. (2002). Live Programming as a Lecture Technique. *Journal of Computing Sciences in Colleges*, 18(2), 51-56.

Pears, A., Seidman, S., Malmi, L., Mannila, L., Adams, E., Bennedsen, J., & Paterson, J. (2007). A Survey of Literature on the Teaching of Introductory Programming. *ACM SIGCSE Bulletin*, 39(4), 204-223.

Porter, L., Guzdial, M., McDowell, C., & Simon, B. (2013). Success in Introductory Programming: What Works? *Communications of the ACM*, 56(8), 34-36.

Rubin, M. J. (2013). The Effectiveness of Live-Coding to Teach Introductory Programming. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education* (651-656).

Saulnier, B. & White, B. (2011). IS 2010 and ABET Accreditation: An Analysis of ABET-Accredited Information Systems Programs. *Journal of Information Systems Education*, 22(4), 347-354.

Sengupta, A. (2009). Teaching Tip: CFC (Comment-First-Coding) – A Simple yet Effective Method for Teaching Programming to Information Systems Students. *Journal of Information Systems Education,* 20(4), 393-400.

Simkin, M. G. & Kuechler, W. L. (2005). Multiple-Choice Tests and Student Understanding: What is the Connection? *Decision Sciences Journal of Innovative Education*, 3(1), 73-98.

Thomas, M. K., Ge, X., & Greene, B. A. (2011). Fostering 21st Century Skill Development by Engaging Students in Authentic Game Design Projects in a High School Computer Programming Class. *Journal of Educational Computing Research*, 44(4), 391-408.

Vidgen, R. & Wang, X. (2009). Coevolving Systems and the Organization of Agile Software Development. *Information Systems Research*, 20(3), 355-376.

Wing, J. M. (2006). Computational Thinking. *Communications of the ACM*, 49(3), 33-35.

Wilkerson, J. W. (2012). An Alumni Assessment of MIS Related Job Skill Importance and Skill Gaps. *Journal of Information Systems Education*, 23(1), 85-97.

Zagalsky, A., Feliciano, J., Storey, M. A., Zhao, Y., & Wang, W. (2015). The Emergence of GitHub as a Collaborative Platform for Education. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing* (1906-1917).

AUTHOR BIOGRAPHIES

Madhav Sharma is a Ph.D. student studying management science and information systems at Oklahoma State University. His research interests include diffusion of innovation, use and implications of artificial intelligence, machine learning, and Internet of Things.

David Biros is an associate professor of management science and information systems and Fleming Chair of Information Technology Management at Oklahoma State University. A retired Lieutenant Colonel of the United States Air Force, his last assignment was as Chief Information Assurance Officer for the AF-CIO. His research interests include deception detection, insider threat, information system trust, and ethics in information technology. He has published in *MIS Quarterly, Journal of Management Information Systems, Decision Support Systems, Group Decision and Negotiation, MISQ Executive, Journal of Digital Forensics Security and Law*, and other journals and conference proceedings.

Surya Ayyalasomayajula is a Ph.D. student studying management science and information systems at Oklahoma State University. His research interests include healthcare analytics, use and implications of deep learning in healthcare contexts, and optimization of operations. He has extensive experience in software development and training.

Nikunj Dalal is professor emeritus of management science and information systems in the Spears School of Business at Oklahoma State University. His research interests involve practical wisdom, learning, philosophical issues in information systems, modeling, and web perception. His work has been published in journals such as: *Information Systems Journal*, *Communications of the ACM*, *Decision Sciences, European Journal of Information Systems*, and *International Journal of Human-Computer Studies,* among others.

APPENDIX

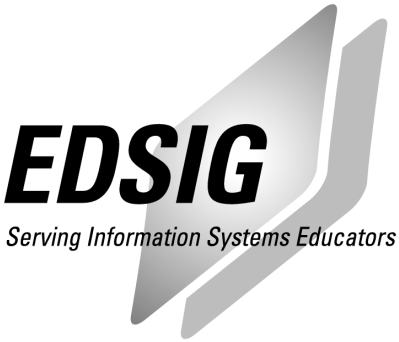
Concept	Exercise	Solution
If Statement	You are applying for a scholarship. You have to make a C# application that calculates your GPA for 3 courses.	*//Getting Course information for Course 1* Console.WriteLine("Course 1"); string course1 = Console.ReadLine(); *//Getting Grade information for Course 1* Console.WriteLine("Grade in Course1"); string grade1 = Console.ReadLine(); *//Declaring and reading integer variable for quality points* int qp1 = 0; *//Declaring and reading if statements course1* if (grade1 == "A") qp1 = 4; if (grade1 == "B") qp1 = 3; if (grade1 == "C") qp1 = 2; if (grade1 == "D") qp1 = 1; if (grade1 == "F") qp1 = 0; *//Getting Course information for Course 2* Console.WriteLine("Course 2"); string course2 = Console.ReadLine(); *//Getting Grade information for Course 2* Console.WriteLine("Grade in Course 2"); string grade2 = Console.ReadLine(); *//Declaring and reading integer variable for quality points* int qp2 = 0; *//Declaring and Reading if statements course 2* if (grade2 == "A") qp2 = 4; if (grade2 == "B") qp2 = 3; if (grade2 == "C") qp2 = 2; if (grade2 == "D") qp2 = 1; if (grade2 == "F") qp2 = 0; *//Getting Course information for Course 3* Console.WriteLine("Course 3"); string course3 = Console.ReadLine(); *//Getting Grade information for Course 3* Console.WriteLine("Grade in Course3"); string grade3 = Console.ReadLine(); *//Declaring and reading integer variable for quality points* int qp3 = 0; *//Declaring and reading if statements course1* if (grade3 == "A") qp3 = 4; if (grade3 == "B") qp3 = 3; if (grade3 == "C") qp3 = 2; if (grade3 == "D") qp3 = 1; if (grade3 == "F") qp3 = 0; *//Performing calculation on integer* int GPA = (qp1 + qp2 + qp3)/3 ; *// Declaring and Reading if statements for qualification.* string qualmessage = ""; if (GPA >= 3) qualmessage = "You qualify for our scholarship"; if (GPA <= 3) qualmessage = "You do not qualify for our scholarship"; Console.WriteLine(qualmessage); Console.ReadKey();

For Loop	You are applying for a scholarship, and Excel has decided to update. You want to write a quick C# program that calculates your GPA for 5 courses and returns a message if you qualify for the scholarship. The program you made before was functional but too lengthy to use again. Write this program using the 'for statement.'	*//Declaring a float variable for quality points* float qp = 0; float totalqp = 0; *//For loop for taking inputs and adding quality points for 5 iterations* for (int i = 0; i < 5; i++) { Console.WriteLine("What is your course grade" + i + "?"); string grade = Console.ReadLine(); if (grade == "A") qp = 4; if (grade == "B") qp = 3; if (grade == "C") qp = 2; if (grade == "D") qp = 1; if (grade == "F") qp = 0; totalqp = totalqp + qp; } { *//calculate GPA* float gpa = totalqp / 5; Console.WriteLine(gpa); string qualmessage = ""; if (gpa >= 3) qualmessage = "You qualify for our scholarship"; if (gpa <= 3) qualmessage = "You do not qualify for our scholarship"; Console.WriteLine(qualmessage); Console.ReadKey();
Arrays	Your application did not go through. Even though your qualification was decided, your program did not store which course had what grade. Use 'for loop' and 'arrays' to make a program that displays what grades you got in what course.	*//Declare Course* string[] CourseName = new string[5]; string[] CourseGrade = new string[5]; *//For loop to get course and grade information* for (int i =0; i<5; i++) { Console.WriteLine("Enter Course name " + i); string CourseName[i] = Console.ReadLine(); Console.WriteLine("Enter Grade for Course name " + i); string CourseGrade[i] = Console.ReadLine(); } *// Display course and grade* for (int i =0; i<5; i++) { Console.WriteLine("Course: " + CourseName[i] + " Grade: " + CourseGrade[i]); } Console.Readkey();

Table 1. Examples of Authentic Tasks Solved in Different Ways

Table 1 shows three exercises dealing with similar information: courses, grades, and GPA. As we moved from week to week to more advanced topics (for example, 'if statement' to 'for loop'), the solutions for these problems started becoming smaller and more parsimonious. A key take-away we wanted our students to get was that the same problem could be solved in different ways such that they strived to construct the most parsimonious solution.

Information Systems & Computing
Academic Professionals



STATEMENT OF PEER REVIEW INTEGRITY

All papers published in the Journal of Information Systems Education have undergone rigorous peer review. This includes an initial editor screening and double-blind refereeing by three or more expert referees.

ISSN 2574-3872