Abstract—This Research to Practice Full Paper presents our experience of positive outcomes with increased motivation and retention in teaching an introductory Computer Science course with Python programming. Without reinventing the wheel, we infused a few well established pedagogies by integrating and evaluating Computational Thinking (CT) skills in a meaningful way. We integrated CT with existing curriculum alongside programming and teaching general problem-solving techniques with a flowchart-based programming environment and without using specific programming concepts or languages at the beginning. Our aim here is not only to teach a programming language per se, but also to teach, at the beginning, the different ways of problem solving, logical reasoning, algorithm design, and programming constructs with minimal or no emphasis on syntax. A positive learning experience is successfully developed for our students by using appropriate pedagogies and strategies. To evaluate the impact of this infusion, a pre- and post-survey as well as a pre- and post-CT test were conducted on student cohort in different sections. The statistical analysis of the survey and test results show evidence of improvement in student’s problem solving and coding skills as well as increase in motivation towards programming.

Keywords—Introductory Programming, Computational Thinking, Visual Learning, Dynamic Classroom, Retention.

I. INTRODUCTION

Introductory programming courses (often referred to as CS1) is among the most challenging ones in CS curriculum, particularly for those who are new in programming. Many students get panicked by the core concepts and extents of this very first programming course, perform poorly, and eventually quit CS program [1] due to failure rates that is observed as high as 30% [2]. Despite different efforts to make introductory programming languages easy to comprehend, freshmen students still struggle with syntax and semantics. However, the aim of an introductory CS course is not to teach a programming language per se, but also to teach the different ways of problem solving, logical reasoning, algorithm design, and programming constructs with minimal or no emphasis on syntax. As Lu & Fletcher wrote [3]: “Programming is to CS what proof construction is to mathematics and what literary analysis is to English”.

Instead they focused about another kind of knowledge as fundamental - computational thinking (CT), which led to the conclusion that a substantial preparation in CT is required before students enroll in programming courses. For example, importance of basic knowledge of the characteristics of CT at an early age is emphasized by Jeanette Wing in [4]. Instead of using CT as an alternative to learning program; it might act as a complement of programming education. Recently, the important of CT education gets more attention in high school level. Still, many freshmen generally do not perform well in the introductory computing courses due to lack of CT skills, leading to poor performance in subsequent courses.

The introductory CS courses should focus on CT principles without teaching the details (syntax and semantics) of a language at the very beginning of the learning process. A visual programming development environment with dynamic flow-charts might be effective in aiding the understanding of CT processes and problem solving skills of novice programmers [5]. After introducing the flow-chart based language, the focus might be eventually shifted to introduce a syntactically simple and interactive language, such as Python. It is a simple and expressive scripting and interactive programming language for beginners to start programming [6,7]. According to a survey analysis by Philip Guo [8], Python is currently the most popular language for teaching introductory computer science courses at top-ranked Universities in the USA. By using Python, a greater emphasis on core CT principles is expected with less of an unwanted focus on syntax. The intention of improving the teaching of computer programming to freshmen students with CT and visual learning is receiving a lot of attention from researchers in recent years [9-14]. At the university level, the development of introductory computing courses with a focus on CT has recently received attention at a wide range of institutions [2,15]. The computer science education literature shows many different approaches to incorporate CT with less focus on computer programming and more on problem solving and critical thinking in introductory courses [14,15].

This paper investigates the impact of our approach of infusing such CT and visual learning to teach an introductory CS course. It extends our previous works [16,17] of
interactive (active) and collaborative learning based pedagogical approaches to teach the same course with the additional development of a framework to integrate and evaluate CT. Within the larger goal of enhancing students’ learning experience and retention, the objective here is to infuse and evaluate CT and flow-chart based visual programming with the existing curriculum alongside programming to fill the gap between CS fundamentals and programming skills. To accomplish this goal, we have implemented the following in our instructional approaches:

1. Developed a CT framework by identifying key concepts and designed lesson activities revolving around those concepts.
2. Taught in a problem-driven way and demonstrated where and how CT principles are needed.
3. Integrated a visual flow-chart based programming environment to focus on CT principles right away.
4. Created an assessment to measure the CT ability of incoming freshmen at the beginning and measure the progress at the end of a semester.

In this paper, we describe our experience of instructing this course (COSC111: Introduction to Computer Science I) in Fall 2019 in six (6) different sections by three different instructors. To evaluate the impact of this infusion on student learning outcomes through robust statistical analysis, a pre- and post-survey were conducted on student cohort. The students were also asked to complete a CT test at the beginning of the semester and again (same items) at the end of the semester to assess skill growth. In addition, data available for the analysis included students’ scores on several homework and lab assignments, quizzes, the midterm exam, the final exam, and the final grades. The evaluation, data analysis, and results complement us to conclude that the proposed approach increases student engagement, facilitates learning, increased student motivation, and contributed to student progress by reducing failure rate and improving retention rate in introductory programming course as well as in other related courses in CS.

II. INFUSING AND EVALUATING CT

CT is identified as a process of solving problems with a focus on algorithms, decomposition, abstractions, and pattern recognition as shown in Fig. 1. This mental process can be described by decomposing a problem into smaller pieces to individually solve each smaller part, which in CS is known as divide and conquer. After the problem is decomposed the work with abstracting the problem to find the smallest solvable problem and to find the parts that can be drawn to a more general solution begins. When the problem has been abstracted and generalized the attempt to develop an algorithm to automate the solution begins. The algorithm must be efficient and easy to replicate and maintain. The solution must also consider organizing, modeling and simulating data in a proper way. Hence, an algorithm is an abstraction of a step-by-step procedure for taking input and producing some desired output. Whereas, algorithmic thinking is the ability to understand, execute, evaluate, and create algorithms. Pattern generalization/recognition is creating models, rules, principles, or theories of observed patterns to test predicted outcomes. The objective here is to integrate and evaluate CT in a meaningful way and to teach in a problem-driven way by presenting only those programming language features that are used and are meaningful to students and demonstrating where and how computational principles are needed.

![Figure 1: CT as a mental process](image)

By following the framework of the above mental process in Fig. 1 and Table 1, several teaching materials (e.g., lecture slides, exercises, quizzes, and small projects) were developed to infuse CT in our course. In addition, an assessment tool is developed to measure the student’s ability in CT. The main purpose of this assessment is to gain an understanding of the existing CT ability of freshmen students at the very beginning of the semester and check the progress in understanding at the end of the semester. Based on the quantification of CT abilities of students with different levels of experience, it allows us to assist individual students in addressing their weakness. The format selected for the assessment is a test of multiple choice questions (25) that was administered in a single session with the students. The CT test is created based on the following considerations:

- Gauge the CT ability of individual students as well as the picture of the entire class.
- Make suitable for freshmen students with or without any previous programming experience.
- Use MCQ type questions with definite answers.
Table 1: A CT framework with key concepts and example exercise (lesson) activities

<table>
<thead>
<tr>
<th>Concepts</th>
<th>Definition</th>
<th>Exercise Activities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Think Algorithmically</td>
<td>An algorithm is a precise, step-by-step set of procedures.</td>
<td>Walk students through the process of creating an algorithm when a chef writes a recipe for a dish.</td>
</tr>
<tr>
<td>Think Abstractly</td>
<td>Generalize patterns into a rule.</td>
<td>Show how understanding images requires understanding a set of abstractions: pixels with different RGB values and position with x, y coordinates.</td>
</tr>
<tr>
<td>Decompose a Problem</td>
<td>Break the problem into smaller, manageable parts.</td>
<td>Show how to reframe a big problem as a series of smaller problems, such as identifying a fake coin when the fake one weighs more than all the others.</td>
</tr>
<tr>
<td>See Patterns</td>
<td>Identify patterns and trends.</td>
<td>Notice or identify similarities or repetition (each smaller part of a fake coin problem) that will help to make predictions or lead us to shortcuts.</td>
</tr>
</tbody>
</table>

III. FLOWCHART-BASED VISUAL PROGRAMMING IN RAPTOR

For teaching CT process with dynamic flow-chart based execution, RAPTOR [19] is used as a free programming environment, which is designed and developed at the United States Air Force Academy. It provides an easy to use interface where only six symbols (e.g., Input, Output, Assignment, Call, Selection, and Loop) are used to solve problems related to decision making and looping concepts of a language as shown in Fig. 4. It also minimizes the amount of syntax that students must learn to write correct program instructions and allows to generate outputs dynamically based on users’ inputs and logical flow of the program.

For our actual instruction, after first couple of weeks of lectures which involves understanding different CT concepts, students are ready to learn problem solving skills with algorithm design, pseudocode, and flowchart in RAPTOR (Figs. 3 and 4) before transitioning to Python programming. Students are taught basic coding skills in RAPTOR without using any syntax. Flowchart doesn’t provide them the final learning objective but using RAPTOR students solve problems in real time. They learn about strings and numbers (basic data types) and how to enter mathematical expression by using different operators. After basic coding concept such as calculating average of three numbers, calculating weekly pay, temperature conversion or calculating simple physical, and mathematical equations, students also learn how to get user inputs and solve those problems. They are also introduced with simple decision making processes (e.g., if-else statement), such as calculating salary based on regular and overtime hours, calculating letter grades (A, B, C, D, F) based on scores (0-100) etc. Finally, students are introduced with simple loop examples, such as printing number in 1-10 or 1-20, counting numbers, calculating running total such as calorie burn in 5 days, number of bugs collected for a week, etc.

The sources for the test questions are identified from several online resources for aptitude test, such as Whimbey Analytical Skills Inventory (WASI), Math Olympiad from high schools, and the University of Maryland High School Mathematics Competition [18]. None of these problems require mathematical skills beyond basic algebra, though most problems require logical or analytical skills and attention to details as shown in Fig. 2 with two sample questions. The test was administered twice online through a Learning Management System (LMS) assessment tools of Canvas: once at the beginning of the CS course and again at the end of the semester with the same set of questions. The results from both test iterations were collected and analyzed to see whether CT scores of students improved after a semester of learning and whether the score can be used as a predictor for success in the introductory programming courses.

Figure 2: Sample CT test questions

The sources for the test questions are identified from several online resources for aptitude test, such as Whimbey Analytical Skills Inventory (WASI), Math Olympiad from high schools, and the University of Maryland High School Mathematics Competition [18]. None of these problems require mathematical skills beyond basic algebra, though most problems require logical or analytical skills and attention to details as shown in Fig. 2 with two sample questions. The test was administered twice online through a Learning Management System (LMS) assessment tools of Canvas: once at the beginning of the CS course and again at the end of the semester with the same set of questions. The results from both test iterations were collected and analyzed to see whether CT scores of students improved after a semester of learning and whether the score can be used as a predictor for success in the introductory programming courses.
Figure 3: Algorithm, Pseudocode, and Flowchart of an example problem for pay calculation

For example, Figs. 3 and 4 show an example problem (lesson) to calculate pay with overtime with algorithm, pseudocode and flow-chart implementation in RAPTOR. Based on the above pedagogy, students can build simple procedural programs without even knowing the syntax of a language (Python) at the very beginning of their learning process. Several lesson plans and assignments were implemented to cover the CT and RAPTOR flow-chart based programming concepts for the first few weeks of the semester before transitioning to Python programming.

IV. RESULTS AND DISCUSSION

For evaluation and result analysis, student cohort from six sections of COSC 111 in Fall 2019 are included in this study which were offered in a consistent manner (similar course materials and instructional approach) by three different CS instructors. A total of 119 students were enrolled across these six sections as shown in Fig. 5.

Data available for the current report includes student scores on homework and lab assignments, quizzes, the midterm exam, the final exam, and the final grades from the Fall 2019 sections of COSC111 (six sections) as well as two voluntary surveys that students were asked to complete, one early in the semester (n = 116; called pre-course) and one near the end (n = 90; called post-course). These two surveys included a few parallel items (to enable matching students to responses on both) and focused primarily on students’ experience with coding prior to the course and on gathering student perceptions of the pedagogical approaches used in the course. Attitudes toward computer science and enrolling in future computer science courses were also included. Finally, students were also asked to complete a CT test at the beginning of the semester (n = 119) and again (same items) at the end of the semester (n = 41), to assess skill growth. Nineteen students took both the pre- and post- CT test.

Table 2 partially provides a demographic summary of the students enrolled. Most of the students grouped themselves as 18-20 years of age (71.6%), male (66.4%) and first-year
students (64.7%) with a self-reported high-school GPA between 3.0 and 3.49 (45.7%). Further, most reported that they intend to take computer science majors (67.2%), but also reported not having had any computer courses in high school (44.0%), though many (31.0%) had one course. Just over 41% (41.4%) agreed or strongly agreed that they had little or no programming experience prior to enrolling in this course. The early-semester survey also included three items testing students’ basic understanding of mathematical operations. The students who correctly answered those items were as follows: #1 – 55.2%; #2 – 39.7%; #3 – 80.2%.

Table 3: Students global perception of course experience

<table>
<thead>
<tr>
<th>Perception</th>
<th>Strongly Disagree</th>
<th>Disagree</th>
<th>Neutral</th>
<th>Agree</th>
<th>Strongly Agree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Having completed this course, I now feel that I can program or solve small problems in Python</td>
<td>2.0%</td>
<td>5.3%</td>
<td>18.4%</td>
<td>41.7%</td>
<td>11.8%</td>
</tr>
<tr>
<td>As a result of my experiences in this class, I plan to take more computer science courses.</td>
<td>9.2%</td>
<td>3.9%</td>
<td>9.2%</td>
<td>25.0%</td>
<td>55.3%</td>
</tr>
<tr>
<td>Overall, the course reduced my fear of programming</td>
<td>5.3%</td>
<td>7.9%</td>
<td>17.1%</td>
<td>10.5%</td>
<td>14.5%</td>
</tr>
<tr>
<td>This class reduced my interest in computer science</td>
<td>25.0%</td>
<td>7.9%</td>
<td>7.9%</td>
<td>5.3%</td>
<td>9.2%</td>
</tr>
</tbody>
</table>

Eleven homework assignment scores were recorded over the duration of the semester that varied in the total number of points possible. These scores were totaled and divided by the total number possible to calculate percentage overall homework score. The same process was implemented for the twenty-one individual lab scores, the seven quiz scores, the midterm exam score, and the final exam score. This approach provided comparable metrics for further analysis that would not be skewed by the number of assignments in each group or the number of points possible for each assignment. Fig. 6 shows the distribution of mean scores of each section as a bar graph. It shows that two of the sections (3 & 5) had notably lower overall averages where majority of the students were from non-CS major. Individual one-way ANOVA tests were run, using the section number as the independent variable and the calculated percentage scores as the dependent variable. Analysis showed that there is significant difference across sections on percent Homework (F(5,113) = 5.293; p = 0.000), Lab% (F(5,113) = 6.960; p = 0.000), and Final Exam % (F(5,113) = 3.169; p = 0.010). Tukey post-hoc comparisons (using an alpha = 0.05) confirm that student scores from Sections 3 and 5 differed from the other sections in more than one instance on these calculated percentage variables.

Table 4: Mean Scores on CT Pre- and Post-Tests

<table>
<thead>
<tr>
<th></th>
<th>PRE Score</th>
<th>PRE %</th>
<th>POST Score</th>
<th>POST %</th>
<th>IMPROVE Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>All Sections (n = 41)</td>
<td>12.12</td>
<td>48.40%</td>
<td>13.56</td>
<td>54.24%</td>
<td>1.44</td>
</tr>
<tr>
<td>Sections 2 and 4 (n = 19)</td>
<td>12.00</td>
<td>48.00%</td>
<td>13.95</td>
<td>55.79%</td>
<td>1.95</td>
</tr>
</tbody>
</table>

Beyond student self-reported improvement in coding skills, the scores of those who took the pre- and post- CT test also show improvement. A total of n = 41 students took both the pre- and post-test on CT. These students were from Sections 2, 3, 4 and 5. (Students from Sections 1 and 6 did not participate). Table 4 shows the students’ scores on both tests in raw score and also a percentage score, as well as an improvement scores, calculated by subtracting the pre-test score from the post-test score (i.e. POST – PRE). To assess student mastery of the course material, the twenty-five (25) questions on the pre-test were repeated on the post-test. The pre-test was administered at the second week of the semester, and the post-test was administered near the end of the semester. Overall, students did improve over the course of the
semester in their computational thinking. The mean pre-test score was 48.49% (all students who participated), while the mean post-test score for those same students was 54.24%. This suggests that the pedagogical approaches used in the course did indeed improve students’ skills. Further, Table 4 also shows the scores of the students in Sections 2 and 4 (that is, the remaining students after Sections 3 and 5 were removed where majority of students were non-CS major). For Sections 2 and 4, the pre-test mean for the students is 48.00% and the post-test mean score is 55.79%.

Table 5: One-way ANOVA comparing two groups of students

<table>
<thead>
<tr>
<th></th>
<th>Between Groups</th>
<th>Within Groups</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUM Squares</td>
<td>19399</td>
<td>124947</td>
<td>144346</td>
</tr>
<tr>
<td>df</td>
<td>1</td>
<td>24</td>
<td>26</td>
</tr>
<tr>
<td>Mean Square</td>
<td>19399</td>
<td>5206</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>3.726</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sig.</td>
<td>0.065</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A one-way ANOVA was run (Table. 5) comparing the students in Sections 3 and 5 with the remaining students (in this case, Sections 2 and 4). As mentioned, no students from Sections 1 or 6 were participated in the CT tests. The section indicator (2 and 4 vs 3 and 5) was used as the independent variable with the mean improvement score as the dependent variable. The results show a marginally significant difference between the two groups (F(1,24) = 3.73; p = .065). This is an encouraging sign, though subsequent studies will need to include more students to be more confident of this effect.

V. CONCLUSIONS

This paper proposes novel pedagogies by incorporating CT and dynamic flow-chart based visual learning in the classroom alongside teaching Python in an introductory CS course. The statistical analysis of the survey and test results show positive outcomes with an increase of student’s learning experience and motivation toward programming. Our future research would include some follow-up measures with those students who either dropped the course or who earned a D or an F as a final grade. This would likely better inform as to what the students who struggled might have found helpful; what might have helped them succeed. Questions already included in the surveys about the amount of coding experience students bring to the course is helpful, but future studies might convert those items to closed-ended ones to allow more targeted comparisons between those with experience, and experience with particular coding languages, with those who do not. Finally, an incentive might be included to encourage students to complete the end-of-semester survey and both the pre- and post-CT tests. Comparisons of the value that the course added to the coding skills would be strengthened by greater numbers of participants.

ACKNOWLEDGMENT

This research was supported by an NSF grant (Award ID 1623335), entitled “Targeted Infusion Project: Infusing Computational Thinking and Visual Learning into an Introductory Computer Science Course to Promote Students' Success and Retention”.

REFERENCES


