Evaluating a Heavily-Instrumented Python CS1 Course

Anonymous1, Anonymous2, Anonymous3
Anonymous Institution
anonymous@anonymous.edu, anonymous@anonymous.edu, anonymous@anonymous.edu

ABSTRACT

The CS1 course is a critical experience for most novice programmers, requiring significant time and effort to overcome the inherent challenges. Ever-increasing enrollments mean that instructors have less insight into their students and can provide less individualized instruction. Automated programming environments and grading systems are one mechanism to scale CS1 instruction, but these new technologies can sometimes make it difficult for the instructor to gain insight into their learners. However, learning analytics collected by these systems can be used to make up some of the difference. This paper describes the process of mining a heavily-instrumented CS1 course to leverage fine-grained evidence of student learning. The existing Python-based curriculum was already heavily integrated with a web-based programming environment that captured keystroke-level student coding snapshots, along with various other forms of automated analyses. A Design-Based Research approach was taken to collect, analyze, and evaluate the data, with the intent to derive meaningful conclusions about the student experience and develop evidence-based improvements for the course. In addition to modeling our process, we report on a number of results regarding the persistence of student mistakes, measurements of student learning and errors, the association between student learning and student effort and procrastination, and places where we might be able to accelerate our curriculum's pacing. We hope that these results, as well as our generalized approach, can guide larger community efforts around systematic course analysis and revision.

Categories and Subject Descriptors

Social and professional topics [**Professional topics**]: Computing education; Information systems [**Information systems applications**]: Data mining

Keywords

cs1, modeling, dbr, python, procrastination

1. INTRODUCTION

The first Computer Science course (CS1) can be a challenging experience for novices given the constraints of a semester [19], but success in CS1 is critical for computer science students, as it sets a foundation for subsequent classes. Large amounts of practice and feedback are critical to this experience, so that learners can overcome programming misconceptions [14, 17] and develop effective schema. Instructors have a key role in developing materials to support learners' productive struggle. Recently, however, scaling enrollments [23] and the move to remote/hybrid learning environments has shifted much of this work away from interacting with individual students towards interacting with systems (which in turn interact with the students directly). For example, programming autograders [16] remove the instructor from the grading process, automatically assessing and sometimes even providing feedback directly to the learner.

Although these systems scale the learning process, they can inhibit the evaluation and revising of course materials. Instructors do not have as many first-hand interactions with students or the artifacts that they produce. When homeworks and exams are no longer hand-graded, teachers may not be as directly motivated to review each submission. Similarly, when automated feedback systems are effectively supporting students, teachers will have fewer opportunities to get direct insight into what issues students are encountering. This knowledge of the students' experience is critical to gauge the effectiveness of the course materials. Instructors need a new model to guide their revision decisions.

We propose instructors follow a Design-Based Research (DBR) approach [6, 3] to iteratively improve their course. In particular, course development should be seen as an iterative and statistical Instructional Design process; each semester, a curriculum is built and presented to learners as an intervention, data is generated and collected as learners interact, that data is analyzed to discover shortcomings and successes of the intervention, and then modifications to the "protocol" are identified for the next iteration of the study. Instructional Design models provide a systematic framework for this development process, but the DBR approach augments this to emphasize the statistical and theory-driven nature of the evaluation process. Fortunately, the same autograding tools that scale practice and feedback opportunities for students can also be used to collect many kinds of learning analytics, permitting the use of educational data mining to garner insights into learning [2].

In this paper, we present our experience of evaluating a CS1 course that has been heavily instrumented to provide rich data on student actions. Our goal is not to prove that our curriculum was a "success" or "failure" as a whole, but to empirically judge specific pieces and identify components that should be modified or maintained. We draw upon programming snapshot data, non-programming autograded question logs, surveys, exam data, and human assessments to produce a diverse dataset. In addition to sharing our conclusions about the state of our course, we believe that we present a formative model for other instructors who wish to evaluate their courses systematically. In fact, our specific analyses are recorded in a Jupyter Notebook[1]. Our hope is that others will use our own analyses as a baseline to develop their own questions, and to motivate others to approach their courses with a more systematic, empirical method.

2. THEORIES AND RELATED WORK

The central premise of our approach is inspired by Design-Based Research, which has been well established in the education literature for decades. Those interested in an introduction to DBR can refer to [6]. Briefly, there are several key tenets: 1) Development is an iterative process of design, intervention, collection, and analysis. 2) Educational interventions cannot be decontextualized from their setting. 3) Processes from all phases of development must be captured and provided sufficient context to ensure reproducibility and replication. 4) Developing learning experiences cannot be separated from developing theories about learning. 5) Results from an intervention must inform the next iteration and communicated out to broader stakeholders.

Messy authenticity is inherent in this process, and naturally limits the theoretical extent of findings in a DBR process. Therefore, any conclusions derived should not be seen as broadly applicable, but only meaningful for the context in which they were developed. Although theories of learning are generated from DBR, this is less true for early iterations. True success for a course is a moving target. As the curriculum improves and students overcome misconceptions faster, more material can be added. Over time, the curriculum necessarily needs to be updated and assignments refreshed. Further, courses often need to be adapted for new audiences with different demographics and prior experiences. Given the DBR model strongly incorporates context, these realities can be accounted for at some level.

DBR has been somewhat underused in Computing Education Research (CER). Recently, Neslon and Ko (2018) made a strong argument that CE research should almost *exclusively* follow Design-Based Research methodologies [24], for three reasons: 1) avoid splitting attention between advancing theory vs. design, 2) the field has not generated enough domain-specific theories, and 3) theory has sometimes been used to impede effective design-based research in the peer review process. Many of the recommendations made in the paper echo the tenets of DBR listed above and are consistent with our vision for communicating our course designs. In fact, their paper was a major guiding inspiration.

Another major inspiration for our approach is Guzdial's 2013

paper evaluating their decade-long Computational Thinking course ("MediaComp") [13]. Although a longer time scale, this paper takes a scientific, cohesive look at their course using a DBR lens. They critically evaluate what worked and contextualize all their findings by their design. They begin with a set of hypotheses about what aspects of the course will be effective, and then systematically review data collected from the offerings to accept or reject those hypotheses. Their conclusions, while not transcendent, are impactful for anyone modeling themselves after their context.

In computing education, programming log data has been used to make various kinds of predictions and evaluations of student learning. Applications include predicting student performance in subsequent courses [8], identifying learners who need additional support [29], modelling student strategies as they work on programming problems [20], evaluating students over the course of a semester [4, 21]. These approaches tend to rely on vast datasets or seek to derive conclusions that are predictive, highly transferable, or are about individual students. Although such research work is valuable, the goal is distinctive. We recognize that each course offering has an important local context that cannot be factored out, and that collecting sufficient evidence over time inhibits the process of iterative course design. Rather than developing generalizable theories or predicting performance, we seek actionable data from a single semester an instructor can use to evaluate and redesign their course.

Effenberger et al [9] are perhaps an example more closely aligned with our own research goals. Rather than evaluating students, their work sought to evaluate four programming problems in a course. Their results suggest that despite commonalities in the tasks, the problems' characteristics were considerably different, underscoring the danger of treating questions as interchangeable in course evaluation.

The process of systematic course revision is similar to the ID+KC model by Gusukuma (2018), which combines formal Instructional Design methodology with a cognitive student model based on Knowledge Components [12]. Instead of focusing on a student model, however, we focus on components of the instruction such as the learning objectives. Still, the systematic process of data collection and analysis to inform revision is common between our methods.

3. CURRICULUM AND TECHNOLOGY

In this section, we describe the course's curriculum and technology. DBR necessitates a clear enough description of the curriculum to understand the evaluation conducted, so we cannot avoid low-level details —the context matters. We have attempted to separate, however, the specific experiential details of our intervention (i.e., the course offering), which are described in Section 3.

As a starting point, we based our course on the [REDACTED] curriculum [2]. This curriculum has students move through a large sequence of almost 50 lessons over the course of a semester, with each lesson focused on a particular introductory programming topic. Each lesson is composed of a set of learning objectives, the lesson presentation, a mastery-

---

[1]REDACTED URL

[2]URL REDACTED

based quiz, and a set of programming problems. We have made a number of modifications to the materials reported in [27], such as the introduction of static typing and increasing the emphasis on functional design to better suit CS1 for Computer Science majors. A full listing of all the learning objectives covered is available [3].

Learning Management System: The course was delivered through Canvas, which was our university's Learning Management System. All material, including quizzes, programming assignments, and exams, were directly available in Canvas (either natively or through LTI).

Lesson Presentation: The lessons were PowerPoint slides with a recorded voice-over, embedded as a YouTube video directly into a Canvas Page. The content of these slides are transcribed directly below the video, including any code with proper syntax highlighting. Finally, PDF versions of all the slides with their transcriptions are also available.

Mastery Quizzes: After the presentations, students are presented with a Canvas Quiz containing a series of True/False, Matching, Multiple Choice, and Fill-in-the-blank questions. This assignment is presented in a mastery style, where learners can make repeated attempts until they earn a satisfactory grade. Each of the 200+ questions are annotated with a specific identifier. These quizzes are 10% of students' grade.

Although Canvas provides an interface to visualize statistics about individual quiz questions, this is obfuscated by the students multiple attempts–only the final grade is shown, so instructors cannot see how difficult a question was for a student. To provide greater detail in an instructor-friendly report, the Canvas API was used to pull all submission attempts for each student. The scripts used in analysis [4] and an example of the instructor report [5] are publicly available.

Programming Problems: Additionally, most lessons contain two-eight programming problems through a web-based Python coding environment [26]. These problems were also presented in a mastery style, allowing learners to spend as much time as they want until the deadline. These problems are 15% of students' final course grade. The environment has a dual block/text interface, although students were discouraged from using the block interface past the first two weeks of programming activities. The environment naturally records all student interactions in ProgSnap2 format [25], making it readily accessible for our evaluation.

Students were also required to install (and eventually use) a desktop Python programming environment, Thonny [1]. Students largely used Thonny for their programming projects, particularly the final project, although a small number chose to use the environment to write code for other assignments. The Thonny environment was not instrumented to collect log data, but students were required to submit their projects through the autograder in Canvas–therefore, submission data should not be affected by the relatively small number of students who used Thonny.

When students submitted a solution to a programming problem, the system evaluated their work using an instructor-authored script written using the Pedal autograding framework [11]. This system generates feedback to learners and calculates a correctness grade (usually 0 or 1, although partial credit was possible on exams). The existing curriculum had a large quantity of autograded programming problems, some of which needed to be updated based on our changes.

Exams: There were two midterm exams and a final exam. These exams were all divided into two parts: 1) multiple-choice/true-false/matching/etc. questions, and 2) autograded programming questions. For the latter, students were given five-six programming problems that they could move freely between. These problems were automatically graded and given partial credit (20% for correctly specifying the header, and the remaining points allocated based on the percentage of passing instructor unit tests). Both parts were presented in Canvas through the systems students were already familiar with, but students were not allowed to use the internet or to Google. Students took the exam at a proctored testing center and had two hours. They were only allowed to bring a single sheet of hand-written notes. Multiple versions of each exam question were created and drawn from a pool at random, so that no two students had the exact same exam.

Projects: There were six projects throughout the semester, although the first two were very small and heavily scaffolded. The final project was relatively open-ended and meant to be summative, but the middle three projects allowed more mixed forms of support. Although students were largely expected to produce their own code, they were encouraged to seek help as needed from the instructional staff. For the final project, students used the Python Arcade library [6] to create a game. Because students were not previously taught Arcade, two weeks were allocated for students to work collaboratively on extending sample games with new functionality. Then, they individually built one of 12 games.

4. INTERVENTION

In this section, we describe the specific intervention context in more detail. The curriculum and technology was used in the Fall 2019 semester at an R1 university in the eastern United States for a CS1 course that was required for Computer Science majors in their first semester. An IRB-approved research protocol was followed. At the beginning of the semester, students were asked to provide consent via a survey, with 103 students agreeing out of 136 (for a 75.7% consent rate). A separate survey was also administered at the beginning of the semester to collect various demographic data (summarized in Table 1, only for consenting students) relating to gender, race, and prior coding experience.

	Percentage	Number
Identifies as Woman	19%	20
Black Student	6%	6
No Prior Coding Experience	37%	38
Total number of students	100%	103

Table 1: Demographic Data for Intervention

---

[3]https://pastebin.com/raw/qKhZgtS9 (anonymized)
[4]REDACTED URL
[5]https://tinyurl.com/ydgsyopl

[6]https://arcade.academy/

Instructional Staff: The course was taught by a single instructor. He managed a team of 12 undergraduate teaching assistants. These TAs varied from CS sophomores to seniors, and not all of them had taken the curriculum before. However, they were all selected by the instructor for both for their knowledge and amiability. All members of the instructional staff hosted office hours. The TAs were also responsible for grading certain aspects of the projects (e.g., test quality, documentation quality, code quality), although this amounted to relatively little of the students' final course grade. The instructor met with these TAs every other week for an hour to discuss the state of the course and provide some minimal training on pedagogy, inclusivity, etc.

Structure: The lecture met Monday-Wednesday-Friday for 50 minutes across three separate sections. The sections were led by the same instructor, but were taught at different times of day (mid-morning, noon, and afternoon). The instructor did not attempt to provide the exact same experience to all three sections–if a mistake was made in the morning section, they attempted to avoid that mistake in the following lectures. Typically, the first lecture session of a module started with 20-30 minutes of review of the material guided by clickers, and then students spent the rest of the module's class time working on open course-work problems. There were several specifically in-class assignments such as worksheets, coding challenges, and readings placed strategically throughout the semester. The lab met on Thursdays for 1.5 hours. Students worked on open assignments with the support of two TAs, who would actively walk around and answer questions.

5. RESULTS AND ANALYSIS

Our ultimate goal is to evaluate the course and identify aspects that were successful and unsuccessful. First, we consider basic course final course outcomes. Then, we use the programming log data to analyze students' behavioral outcomes from the semester. We dive deeper into this data to characterize the feedback that was delivered to students over the semester. We look at fine-grained data from both parts of the final exam to develop a list of problematic subskills, and then review more of the programming log data in light of these results. We particularly focus our efforts on subskills related to defining functions, to tighten our analysis.

The instructor's naive perception of the course was that things were largely successful, except for the final project. Insufficient time was given to the students to learn the game development API, and instructor expectations were a bit high (which was adjusted for in the grading, but may have caused students undue stress). However, the material prior to the final project went smoothly. Office hours were rarely overfilled, with the exception of week 4 (the module introducing Functions), which had one lesson too many–this was resolved by making the last programming assignment optional (Programming 25: Functional Decomposition).

5.1 Basic Course Outcomes

As a starting point, we consider basic course-level outcomes, the kind that could be determined even without the extra instrumentation. This will include the overall course grades, the major grade categories, and the university-administered course evaluations. As a starting point, the total number

of failing grades and course withdrawals (DFW rate) was 14.5%, considered acceptable by the instructor.
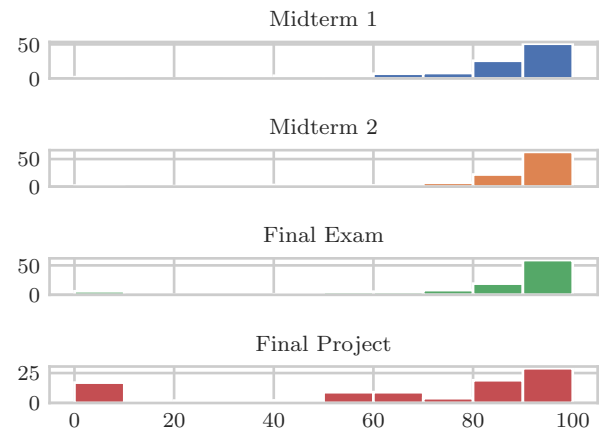

Figure 1: Exam and Final Project Grade Distributions

Figure 1 gives histograms for Midterm 1 and 2, Final Exam, and Final Project scores. The midterms were largely the same as the final exam. There was considerably more variance in the final project scores, possibly due to the issues outlined before. The fact that many students were incapable of producing a final project may be evidence that the assignment was beyond what students may have been capable of.

A Kruskal-Wallis non-parametric test was used to analyze final exam scores by demographics. There were no significant differences for gender, but a large difference for black students (H(1)=6.39, p=.01) and a smaller difference for prior programming experience (H(1)=5.51, p=0.02). The students without prior experience scored about 12% lower on average, while the black students scored about 41% lower. Given the concerning spread here, we review this data with more context in the next section before drawing any conclusions.

The anonymous course evaluations from students run by the university yielded positive but simplistic results. Both the course and the instructor were separately rated on a 5-point likert scale (Poor/Fair/Good/Very Good/Excellent). Both the course (Mdn=5, M=4.62, SD=0.77) and the instructor (Mdn=5, M=4.70, SD=0.67) achieved very high results, but ultimately this tells us little about the students' experience. Course evaluation data is known to contain bias and provide limited data [5, 22]; these results must be taken in context with other sources of data. Note that because the course evaluations are anonymous, they cannot be cross-referenced with other data sources.

A brief review of the students' free response answers reveals that many were unsatisfied with the Final Project. In fact, the word "Arcade" appears in 41 of the 86 text responses, often as their singular point of criticism. Although this helps us understand a major point of failure in our curriculum, it highlights the necessity of the alternative evaluation mechanisms. Relying solely on student final perceptions leaves us vulnerable to student biases in judging success of course elements.

5.2 Time Spent Programming

The keystroke-level log data allows us to determine a number of interesting metrics beyond what is available from our grading spreadsheet. As a simple starting point, using the timestamps of the programming logs we can get a measure of how early students started working on assignments and total time spent. `Earliness` was measured by taking each submission event across the entire course, finding the difference between this and the relevant assignment's deadline, and averaging those durations together within each student. `Hours Spent` was measured by grouping all the events in the logs by student, finding the difference with the next adjacent event (clipping to a maximum of 30 seconds, to consider breaks), and summing these durations.

Figures 2a, 2b, and 2c show a marginal plot between earliness, hours spent, and final exam grade. Spearman's Rho was used to calculate the correlation between each of these outcomes. Consistent with recent studies by Kazerouni [15], earliness (a measure of procrastination) had a significant medium correlation with exam scores ($r_s = .49, p < .001$), while time spent was only modestly correlated ($r_s = -.32, p = .001$). Interestingly, there was no significant correlation between student's time spent and their procrastination ($r_s = -0.09, p = .36$).

Analyzing behavioral outcomes by demographics using a Krusakl-Wallis test indicated no differences, with the exception of total hours spent between women vs. men (H(1)=9.77, p=0.002) and between students with vs. without prior experience (H(1)=7.28, p=0.007). This comparison is visualized in Figures 3a and 3b. Women and students with no prior experience spent, on average, about 8 and 5 hours more than their counterparts. Importantly, this means that there was no significant difference in how early students started between subgroups.

Given the difference in final exam scores, black students appear poorly served by the current curriculum. On average, these students spent as much time as their peers on assignments, but their final exam scores were lower than students outside of this category. Given the evidence for the continued education debt owed to non-White students (Ladson-Billings, 2006) [18], more work is needed to identify both potentially problematic structural elements of the course and how the course can better draw on student strengths to produce more equitable outcomes.

Figure 4 visualizes the total time spent by students per week on the programming problems. The data collected raises an interesting question–how many hours should we ideally expect students to spend on our courses? How much should that amount vary over time and between individual students? At our institution, the guidance from the administration[7] is that in a three-credit course like this one, students should spend 45 hours in class and 90 hours outside of class over the course of the 15-week semester.

The median time spent in our course by a given student on all the programming assignments was 19 hours, while the highest time spent by any individual student was just

---

[7]REDACTED URL

Category	Subcategory	Percentage
Instructor		37.8%
	Problem Specific	32.1%
	Not Enough Student Tests	1.0%
	Not Printing Answer	.8%
Analyzer		22.1%
	Initialization Problem	6.9%
	Unused Variable	5.9%
	Multiple Return Types	2.9%
	Incompatible Types	1.4%
	Parameter Type Mismatch	1.0%
	Overwritten Variable	.6%
	Read out of scope	.5%
Correct		17.8%
Syntax		11.4%
	No Source Code	.3%
Runtime		7.8%
	TypeError	3.7%
	NameError	1.0%
	AttributeError	.8%
	ValueError	.4%
	KeyError	.4%
	IndexError	.4%
Student	Student Tests Failing	2.9%
Instructions		2.4%
System Error		.8%

Table 2: Frequency of Error Messages by Category

over 42 hours. This does not take into account time spent outside of the coding environment (e.g., working on projects in Thonny), working on quizzes, and reading/watching the lesson presentations. However, some students did complete their projects in the online environment, and we expect most of those activities to take considerably less time than the programming activities. This may suggest that we are not asking our students to dedicate as much time as we might.

Looking at specific time periods within the data, we see that students spent considerably less time programming in the first few weeks of the course, around the midpoint, and near the end of the programming problems (the last few weeks, representing the final project, necessarily took place outside of the online coding environment). Especially for the earlier material in the course, it is likely that the pace can be accelerated. Of course, there may be advantages to the gentle initial slope of the material–making students more comfortable with the material. Still, given the time pressure we observed in Week 4, it would seem prudent to space that work out a bit more.

5.3 Error Classification

The feedback from the autograder is available in the log data, and can be plotted over time. Table 2 gives the percentage of different feedback messages that students received on programming problems as a percentage of all the feedback events received. Our numbers vary from those reported by Smith and Rixner (2019) [28], possibly because of our very different approach to feedback and the affordances of our programming environment.

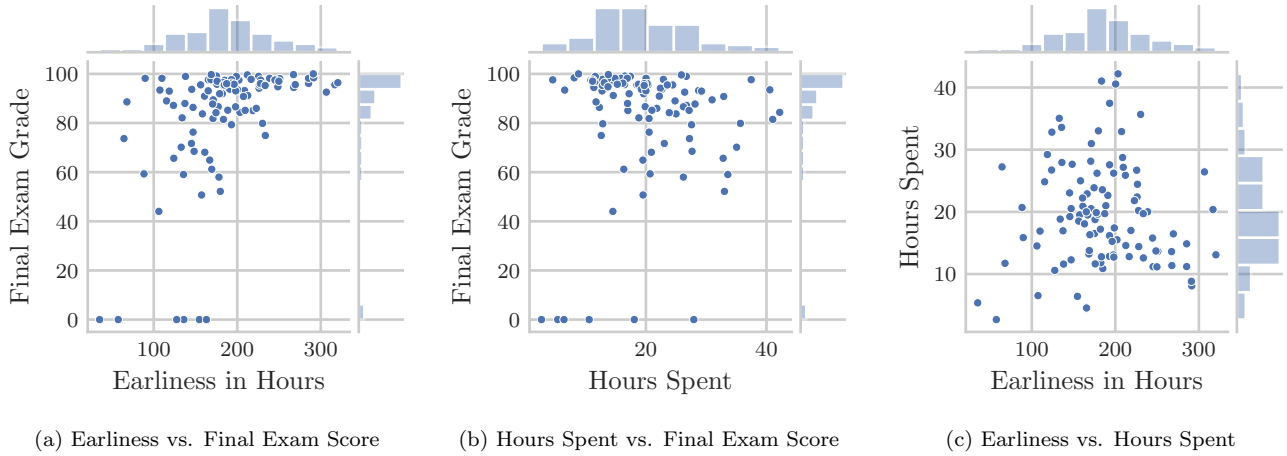One of the most notable departures is the separation of An-

(a) Earliness vs. Final Exam Score

(b) Hours Spent vs. Final Exam Score

(c) Earliness vs. Hours Spent

Figure 2: Comparison of Earliness, Time Spent, and Final Exam Score

(a) Hours Spent by Gender

(b) Hours Spent by Prior Experience
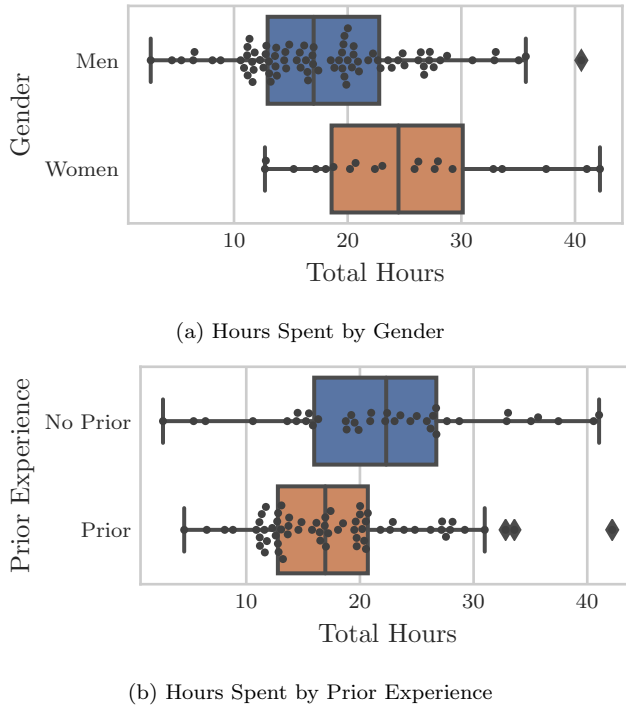
Figure 3: Hours Spent by Demographics

Figure 4: Time Spent per Week of Semester

The Analyzer contributes more than a fifth of all feedback delivered to students, suggesting it has a significant role. Although this data shows us that students are receiving our enhanced feedback, this does not mean we understand its impact on students' learning–we observe that further work is needed to evaluate the quality, accuracy, and reliability of this specialized feedback.

We observe that the highest category (Problem-specific Instructor feedback) is opaque and coarsely grouped. Given that almost a third of the feedback is classified with this label, it is unhelpful that the category cannot be easily broken down further. Sampling the specific text provided to students, we see examples like students failing instructor unit tests, a reminder about assignment instructions to call a function just once, and a suggestion to avoid using a specific index in a subscript. Although the autograder provides a powerful mechanism for delivering contextualized help to students, the lack of organization severely limits the amount of automated analysis possible. As part of our process in the future, we intend to more carefully annotate each piece of
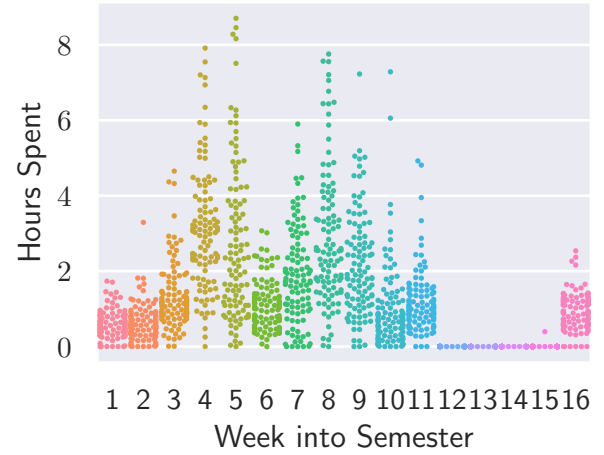
alyzer and Problem-specific Instructor feedback from the Runtime and Incorrect Result categories. The autograding system is capable of overriding default error messages with ones that are determined to be more appropriate. In particular, one of its key features is a type inferencer and flow analyzer that automatically provides more readable and targeted error messages. The subcategories under the Analyzer category give examples of the kinds of errors encountered: `initialization problems` (using a variable that was not previously defined) frequently supersedes the classic NameError, for example. Meanwhile, some issues have no corresponding runtime error, such as `unused variables` (never reading a variable that was previously written to).

feedback in our autograding scripts with unique labels to better support this kind of automated analysis.

Figure 5a gives the ratio of correct submission events in the log data over each week of the semester. Early on, students are able to complete problems with fewer attempts than later problems. This would seem to explain the steady growth in ratios of different kinds of feedback over time, as evidenced by Figures 5b and 5c. It is interesting to observe that the runtime error frequency grows almost linearly over the course of the semester, with the exception of week 7 (which is a peak week for the Analyzer feedback, per Figure 5d). We hypothesize this particular outlier is the result of some more carefully-refined feedback available during that week.

5.4   Final Exam Conceptual Questions

The first part of the final exam was composed of conceptual questions for topics across the curriculum, drawn largely from the quiz questions students had already seen. In this section, we reviewed the automatically generated report to determine the specific set of questions that students struggled with, in order to evaluate their conceptual understanding. Most students performed relatively well across the questions, so we emphasize errors where more than 80% of the students had incorrect answers.

Students largely had no issues with questions involving evaluating expressions. A small exception to this is students' struggle with Equality vs. Order of different types. In Python, as in many languages, it is not a type error to check if two things of different types are equal (although that comparison will always produce false); however, it will cause an error if you attempt to compare their order (less than/greater than operators). In fact, 58% of students got this specific question wrong.

There were three questions related to tracing more complex control flow: tracing loops, `if` statements, and functions. Tracing seemed to pose difficulties, with between 25-40% of the students getting these questions wrong. We believe that more emphasis should be placed on tracing in the curriculum; currently there are several quiz questions and a worksheet dedicated to the topic, but there are plenty more opportunities to expand this material. Tracing has been a recent area of focus within computing education research, with promising teaching approaches by Xie [30] and Cunningham [7].

Dictionaries also posed significant trouble for students. We posit a few reasons: dictionaries come up later in the course, so students had less time to learn about them; dictionaries, although powerful, also incorporate a considerable amount of complexity in how they mix heterogeneous types and data; dictionaries conflate syntactic operations with lists and have confusing surface similarities with the concept. In fact, this last point is evidenced by the data from the exam. In a question comparing the relative speed of traversing lists and dictionaries, 50% of the students got one variant of a True/False question incorrect (in other words, they might as well have been guessing); notably, the same question was phrased positive and negatively in the pool, and the negative phrasing only had 33% of the students get the answer wrong.

Again, the point of our analysis is not to necessarily develop a validated examination instrument or to distill an exhaustive and authoritative set of misconceptions. Instead, we seek to demonstrate the insight we have garnered from reviewing our exam. With these simple percentages, we have found targets to aim our revisions towards.

5.5   Deeper Dive on Functions

In looking over the second part of the final exam questions, we are faced with a tremendous number of concepts being integrated into each problem. In fact, with over 261 learning objectives in the course, analyzing the entire set is still an overwhelming prospect. To scope our analysis for this paper, we decided to focus on a subset of skills just related to Functions. In fact, that subcategory of objectives alone has 56 items, so we chose to only focus on seven particular subskills that we felt we could clearly identify with computational analysis. Although there are many other subskills involved in the topic of functions, these were a reasonably-sized set of critical ones that the instructor felt, a-priori, they had seen students struggle with over the course of the semester. Appendix X gives a more precise description of the criteria we applied to each submission's code, but Table 3 gives a brief summary and the percentages (and quantity) of students who successfully demonstrated the subskill on each of the three exams.

Header Definition: Function use begins with function definition, so even though we had not observed many students struggling with syntax during the semester, we felt it critical to analyze the incidence of submitted code that had malformed headers. Although the numbers were a little higher than expected, we are not terribly concerned with these values - reviewing the submissions, many seemed like simple typos (e.g., a missing colon) that were relatively easily fixed. However, those students who fail to define their header do seem worthy of particular attention, particularly earlier in the course.

Provided Types: Students were not required to provide types in their function headers, nor even explicitly encouraged during the exam. In fact, since the advanced feedback features were turned off, their runtime feedback would not actually reference any parameter or return types they specified (as long as they were syntactically correct Python code). Note that we are not assessing the correctness of their provided types - merely their existence (which suggests that students felt that documenting their function's type was advantageous for themselves).

In the final exam, the number of students who annotated their parameter types falls off sharply after the first three questions (55, 51, 51, 21, 22, 22). We offer two explanations: first, the fourth question is one of the most difficult in the entire course, suggesting that students may have been distracted by the problem's difficulty. Second, the 4-6 questions all involve more complicated nested data types (e.g., lists of dictionaries) that may have proved too troublesome for the students to describe.
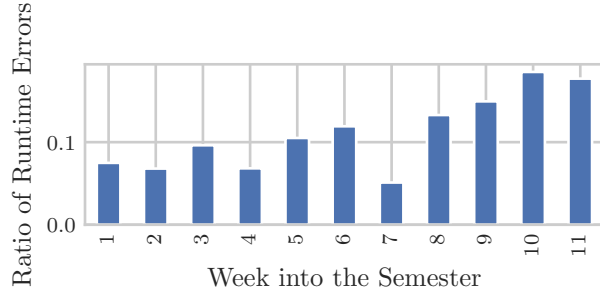
Parameter Overwriting: This particular misconception is one that the instructors were very concerned with, having observed it repeatedly among certain students early in
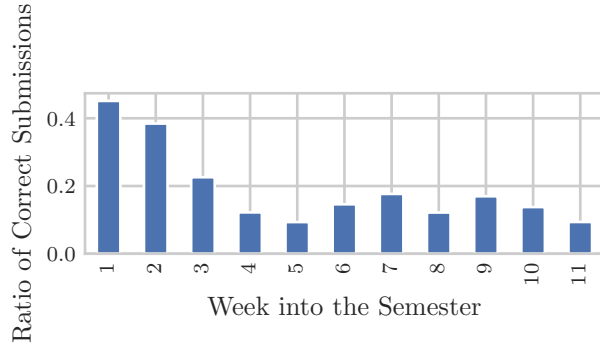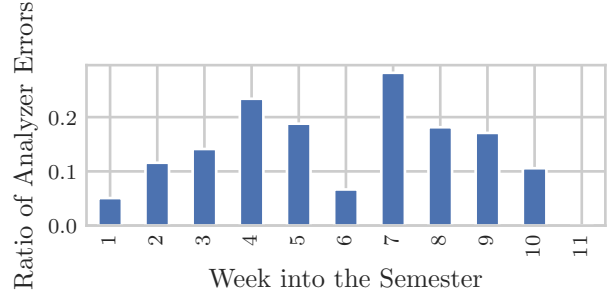
(a) Ratio of Correct Submissions by Week of Semester
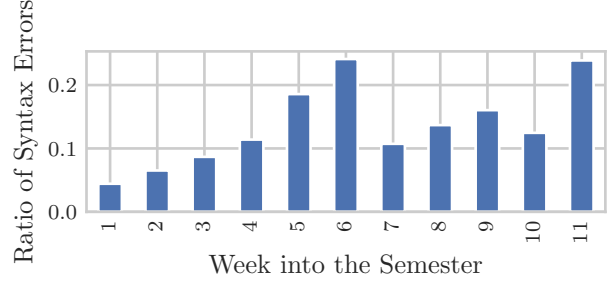
(b) Ratio of Syntax Errors by Week of Semester

(c) Ratio of Runtime Errors by Week of Semester

(d) Ratio of Analyzer Errors by Week of Semester

Figure 5: Ratio of Feedback Types by Week of Semester

Subskill	Description	1st Exam	2nd Exam	Final Exam
Header Definition	Defined the function header with correct syntax	83.5% (86)	84.5% (87)	91.3% (94)
Provided Types	Provided types for all parameters and the return	40.8% (42)	45.6% (47)	37.9% (39)
Parameter Overwrite	Did not assign literal values to parameters in the body	88.3% (91)	98.1% (101)	99.0% (102)
Return/Print	Did not print without returning	80.6% (83)	89.3% (92)	91.3% (94)
Parameters/Input	Did not use the input function instead of parameters	96.1% (99)	100.0% (103)	99.0% (102)
Unit Testing	Wrote unit tests	88.3% (91)	79.6% (82)	67.0% (69)
Decomposition	Separated work into a helper function	1.0% (1)	17.5% (18)	19.4% (20)

Table 3: Percentage of Students Demonstrating Subskill across Exams

the semester (and concerned with its persistence). Applying the parameter overwriting pattern to the rest of the submissions over the entire semester, we see that parameter writing behavior trails off over the course of the semester. By the final exam, almost no students were making this particular mistake. Although the instructor believes that more can be done up front to avoid this critical misconception, it is comforting that the existing curriculum seems to largely stamp out the idea by the end.

Return/Print: Although printing itself is a common strategy for debugging, we have observed that some introductory students struggle to differentiate between the concepts of return statements and print calls. In particular, we only analyzed the final exam questions that required students to return a value (only one exam question required printing). However, largely students were successful with this subskill, despite a quarter of students getting a related (more abstract rendering) version of this subskill wrong on part 1 of the final exam. It seems that although persistent for a small

clutch of students, most are able to understand the concept by the end of the semester.

Parameters/Input: Similar to students' issues with returning vs. printing, some students were observed in individual sessions mixing up parameters and the input function (which was presented as a very distinctive way that data could enter a function). However, it appears that this was truly isolated to just a few students; the vast majority understood the distinction between these cases.

Functional Decomposition: Largely inspired by the success of Fisler et al [10] in teaching Decomposition to overcome the difficulties of the Rainfall problem, Functional Decomposition was taught as a method for handling complexity, and was encouraged on problems that would be particularly difficult. Students had previously been taught 8 different looping patterns (summing, counting, accumulating, mapping, filtering, finding, taking, and min/maxing). A number of assignments required students to decompose

	Decomposed		Monolithic	
	Pass	Fail	Pass	Fail
Problem #35.4	37	8	29	27
Midterm 2 Q5	13	5	40	43
Final Exam Q4	7	8	42	44
Total	57	21	111	114
	18.8%	6.9%	36.6%	37.6%

Table 4: Student Use's of Decomposition vs. Outcome across Assessments

problems. Therefore, it is somewhat disappointing that so few students chose to attempt decomposition (particularly since the harder final exam problems were naturally susceptible to a decomposition approach).

In addition to the midterm 2 and final exam questions, we also took a closer look at an earlier open-ended programming problem that was particularly complex and well-suited to decomposition. In these problems, there seemed to be a pattern of students being more successful when they leverage decomposition. Although far from conclusive, this supports the hypothesis that decomposition may be an effective strategy and is worth investing more course time into teaching.

Unit Testing: Given that students were not required to unit test their code on the final exam, we were pleased to find that many students wrote unit tests regardless. Interestingly, though, the percentage of students who used this strategy decreased over the course of the semester, even as the programming problems became more difficult. We hypothesize that since the later problems involve fairly complex nested inputs, and students either did not feel comfortable generating test data, or they felt that it would not be an efficient use of their time. If this is the case, we believe that we need to do more in the course to sell the concept to students - rather than letting them think that writing test cases would be a detriment to their success, they should see it as one of the most direct paths to completing the problem.

6. DISCUSSION

Reviewing our findings, we made several decisions about places to modify our curriculum. The log data suggests that some of the earlier material can be accelerated, so that more time can ultimately be allocated to week 4 (critical material covering functions). This would hopefully improve students' foundational skills such that they will perform better on the latter Dictionary material in weeks 6-8, where they seem to currently struggle. We also believe we need to spend more time throughout the semester convincing students that subskills like decomposition and unit testing can help them solve challenging questions, although follow-up analyses will be needed to confirm this theory. Finally, we must come up with new ways to support some of our demographic subgroups, given that outcomes in that area are not yet equal.

During the process of reviewing the data from our semester, we were most struck by the overwhelming quantity. The problem became as much one of filtering and deciding what data to focus on, so as to not get lost. We intentionally excluded a large amount of survey data that we had collected, logs from students attempting the mastery quizzes, and a considerable number of other data sources.

Better structure to our existing data sources might help in future analyses. For example, although each quiz question was labeled with a unique identifier, we realized during analysis that we really needed every quiz *answer* (and in some cases, sets of answers) to have a unique identifier as well. In particular, some questions had multiple parts, or different answers yielded information about different misconceptions. In a similar vein, annotating instructor feedback for the programming problems would have substantially increased the differentiation of our feedback messages.

More metadata about each identifier would also help efforts to cross-reference and cluster related problems (especially over time). This is a non-trivial effort, given the quantity of course materials present in the curriculum. As a starting point, we believe this effort should probably be focused on certain major learning objectives and topics (e.g., functions) that are particularly worthy of attention based on the formative evaluation conducted here.

We expect that before our next iteration of our analyses, we need to develop more hypotheses up front for guidance. A considerable amount of time was spent performing exploratory analyses, trying different approaches and seeing what emerged from the data. Although helpful as we oriented ourselves, the data dredging that can emerge may yield false conclusions that are not actually worth investing in.

In retrospect, we felt we should have separated out students by section to rerun some of the categorical analyses, to confirm that there were no differences between sections. We had intentions to run factor analyses and hierarchical clustering algorithms to develop learner profiles and build up trajectories of student learning. Although there are some residuals of this analysis in our Jupyter Notebook, none of the findings crystallized into something suitable for this paper. Finally, while we attempted to follow a replicable process in our data collection and analysis, we believe more should be done to streamline and package our data pipeline to encourage replication and reproduction.

7. CONCLUSION

In this paper, we have described our evaluation of data from a heavily-instrumented CS1 course. Our goal was less about judging the course overall, and more about finding specific areas of improvement and success. We feel that course evaluation is less about the end-goal and more about small iterative augmentations that collect over time. To structure our approach, we followed a loose Design-Based Research model supported by educational data mining. In our experience, the high volume and variety of data sources can be very helpful in understanding the successes and failures of the course, although it does pose difficulties for analysis. As always, a better pipeline could help make sense of these data and results more quickly, possibly even during the semester. However, in the immediate term, our data analysis contributes to the community's knowledge of students and ideally provides a model for others to follow along. In general, we hope to encourage increased rigor in course evaluation as we start integrating data-rich tools into our

courses.

8. REFERENCES

[1] A. Annamaa. Introducing thonny, a python ide for learning programming. In *Proceedings of the 15th Koli Calling Conference on Computing Education Research*, pages 117–121, 2015.

[2] R. S. Baker and P. S. Inventado. Educational data mining and learning analytics. In *Learning analytics*, pages 61–75. Springer, 2014.

[3] S. Barab and K. Squire. Design-based research: Putting a stake in the ground. *The journal of the learning sciences*, 13(1):1–14, 2004.

[4] K. Buffardi. Assessing individual contributions to software engineering projects with git logs and user stories. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, pages 650–656, 2020.

[5] S. E. Carrell and J. E. West. Does professor quality matter? evidence from random assignment of students to professors. *Journal of Political Economy*, 118(3):409–432, 2010.

[6] D.-B. R. Collective. Design-based research: An emerging paradigm for educational inquiry. *Educational Researcher*, 32(1):5–8, 2003.

[7] K. Cunningham, S. Blanchard, B. Ericson, and M. Guzdial. Using tracing and sketching to solve programming problems: replicating and extending an analysis of what students draw. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*, pages 164–172, 2017.

[8] N. Diana, M. Eagle, J. Stamper, S. Grover, M. Bienkowski, and S. Basu. Measuring transfer of data-driven code features across tasks in alice. 2018.

[9] T. Effenberger, J. Cechák, and R. Pelánek. Difficulty and complexity of introductory programming problems. 2019.

[10] K. Fisler. The recurring rainfall problem. In *Proceedings of the tenth annual conference on International computing education research*, pages 35–42, 2014.

[11] L. Gusukuma, A. C. Bart, and D. Kafura. Pedal: An infrastructure for automated feedback systems. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, pages 1061–1067, 2020.

[12] L. Gusukuma, A. C. Bart, D. Kafura, J. Ernst, and K. Cennamo. Instructional design+ knowledge components: A systematic method for refining instruction. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, pages 338–343, 2018.

[13] M. Guzdial. Exploring hypotheses about media computation. In *Proceedings of the ninth annual international ACM conference on International computing education research*, pages 19–26, 2013.

[14] L. C. Kaczmarczyk, E. R. Petrick, J. P. East, and G. L. Herman. Identifying student misconceptions of programming. In *Proceedings of the 41st ACM technical symposium on Computer science education*, pages 107–111, 2010.

[15] A. M. Kazerouni, S. H. Edwards, and C. A. Shaffer. Quantifying incremental development practices and their relationship to procrastination. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*, pages 191–199, 2017.

[16] H. Keuning, J. Jeuring, and B. Heeren. Towards a systematic review of automated feedback generation for programming exercises. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, pages 41–46, 2016.

[17] E. Kurvinen, N. Hellgren, E. Kaila, M.-J. Laakso, and T. Salakoski. Programming misconceptions in an introductory level programming course exam. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, pages 308–313, 2016.

[18] G. Ladson-Billings. From the achievement gap to the education debt: Understanding achievement in us schools. *Educational researcher*, 35(7):3–12, 2006.

[19] A. Luxton-Reilly. Learning to program is easy. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, pages 284–289, 2016.

[20] P. Mandal and I.-H. Hsiao. Using differential mining to explore bite-size problem solving practices. In *Educational Data Mining in Computer Science Education (CSEDM) Workshop*, 2018.

[21] C. Matthies, R. Teusner, and G. Hesse. Beyond surveys: analyzing software development artifacts to assess teaching efforts. In *2018 IEEE Frontiers in Education Conference (FIE)*, pages 1–9. IEEE, 2018.

[22] K. M. Mitchell and J. Martin. Gender bias in student evaluations. *PS: Political Science & Politics*, 51(3):648–652, 2018.

[23] E. National Academies of Sciences, Medicine, et al. *Assessing and responding to the growth of computer science undergraduate enrollments*. National Academies Press, 2018.

[24] G. L. Nelson and A. J. Ko. On use of theory in computing education research. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*, pages 31–39, 2018.

[25] T. W. Price, D. Hovemeyer, K. Rivers, B. A. Becker, et al. Progsnap2: A flexible format for programming process data. In *The 9th International Learning Analytics & Knowledge Conference, Tempe, Arizona, 4-8 March 2019*, 2019.

[26] REDACTED. Redacted. 2017.

[27] REDACTED. Redacted. In *REDACTED*, 2019.

[28] R. Smith and S. Rixner. The error landscape: Characterizing the mistakes of novice programmers. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, pages 538–544, 2019.

[29] Y. Vance Paredes, D. Azcona, I.-H. Hsiao, and A. F. Smeaton. Predictive modelling of student reviewing behaviors in an introductory programming course. 2018.

[30] B. Xie, G. L. Nelson, and A. J. Ko. An explicit strategy to scaffold novice program tracing. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, pages 344–349, 2018.

APPENDIX

A. FIRST ONE