PythonSneks: An Open-Source, Instructionally-Designed Introductory Curriculum with Action-Design Research

Austin Cory Bart
University of Delaware
Newark, DE
acbart@udel.edu

Allie Sarver
Virginia Tech
Blacksburg, VA
afsarver@vt.edu

Michael Friend
Virginia Tech
Blacksburg, VA
mrf7@vt.edu

Larry Cox II
Virginia Tech
Blacksburg, VA
lacox@vt.edu

ABSTRACT

Rising enrollments and limited instructor resources underscores the growing need for reusable, scalable curriculum. In this paper, we describe an open-source introductory Python course for non-Computer Science majors in STEM, designed following best practices of Instructional Design (a process similar to Software Engineering). The created resources include 234 learning objectives, 51 lesson videos, 45 lecture slides, 170 programming problems, 281 quiz questions, 6 unit tested projects, and 4 ethical prompts. A teaching field guide has also been produced as a result of this effort, documenting how to deploy this curriculum on a daily level. We describe our experiences deploying over two semesters. The course serviced over 500 students, with 100s in some sections. Along the way, two interventions were conducted in an Action Design Research style: one using Worked Examples, and another using Structured Small Groups. We report on the mixed results of these experiments, plus evaluations of the assignments from student surveys and statistical measures of item effectiveness. Finally, we describe lessons learned when following Instructional Design processes.

CCS CONCEPTS

• **Social and professional topics** → **Computing education**; **Model curricula**; **CS1**;

KEYWORDS

Python; Instructional Design; Open Curriculum

ACM Reference Format:
Austin Cory Bart, Allie Sarver, Michael Friend, and Larry Cox II. 2019. PythonSneks: An Open-Source, Instructionally-Designed Introductory Curriculum with Action-Design Research. In *Proceedings of Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3287324.3287428

SIGCSE '19, February 27-March 2, 2019, Minneapolis, MN, USA

ACM ISBN 978-1-4503-5890-3/19/02…$15.00
https://doi.org/10.1145/3287324.3287428

1 INTRODUCTION

Ever rising enrollments are bringing more students to computing courses, requiring more instructors and scalable curriculum. Frequently, these busy instructors are forced to reinvent much of their curriculum from scratch even as they gather materials from disparate external sources. The limited time and resources available leads to a lack of systematic processes in curriculum development, which can cause a wide range of issues: instructors focusing on lesson plans without reference to established assessments, course staff becoming out-of-sync on learning objectives, and untested assignments with errors and typos. The end result is overworked instructors struggling to make do with inadequate curriculum.

Curriculum Sharing is one mechanism that can reduce the burden on instructors. Open-sourced curriculum developed by other instructors can be adopted and improved. However, the authors have found few modern introductory computing curricula that follow best practices of *Instructional Design*: systematic construction of a curriculum, with parallels to Software Engineering. Most resources available are only parts of a full curriculum, not easily modified, and not tuned to a conventional undergraduate setting.

Working with an Instructional Designer, we created an introductory Python curriculum (PythonSneks) to fill this gap. Our vision is of an introductory curriculum that is easily adopted by instructors but inherently open and modifiable. The material is targeted at undergraduate non-computing majors with a range of prior experiences in a typical undergraduate setting, but suitable for other contexts. The course incorporates many modern pedagogical methods to reach large (100+) audiences: flipped-classroom lessons, peer instruction, active learning, and activities with immediate feedback. When deploying our course in the past year, the developers collected a large amount of data for course evaluation and answering research questions using an Action Research approach. We summarize the results of the above in the hopes that other adopters will use the materials to refine their own courses. Ultimately, this paper makes the following contributions:

- Description of the development, deployment, and evaluation of an open-source Python curriculum (available publicly at https://acbart.github.io/python-sneks/).
- Results from two experimental interventions on the courses' second offering, on Worked Examples and Structured Groups.
- Lessons for other curricula that seek to be open-source.

2 EXISTING CURRICULUM

As the most popular introductory language for university undergraduate courses [9], there are a large number of existing introductory Python curricula. Prior to developing our curriculum, we surveyed a number of courses from the literature and explicitly sought out more via a post on the SIGCSE-Members mailing list [2]. In this section, we review a subset of our findings, and discuss why we think our approach is a useful addition to the field. A complete listing of alternative curricula that we found is available at https://acbart.github.io/python-sneks/alternatives.

A recurring problem we found in reviewing courses is the limited accessibility of materials. For instance, [17] describes a curriculum, but provides no way to access materials. Although most instructors are willing to share when prompted, the social barrier of reaching out is often associated with unpackaged materials. Of course, there are projects that have published their materials. Chor & Hod describe a largely lecture-based curriculum (CS1001.py) focused on a wide range of Computer Science oriented topics (hashing, fibonacci, sorting, etc.) [6], representative of many similar approaches from the last few decades. Although suitable for students already interested in Computer Science, it is unclear how well such topics would appeal to broader, non-major audiences. Further, as we will describe, few such curricula follow principles of Instructional Design.

In response to the more abstract Computer Science courses, a number of courses have been created with a contextualized approach. Guzdial popularized a Python curriculum based on Media Computation in the early 2000s that saw wide-spread adoption [11]. This curriculum attempted to appeal to students' sense of interest in media and animation manipulation. Unfortunately, much of the curriculum's dependent technology seems to have stagnated [12]. Kafura et al describes an introductory Computational Thinking curriculum for non-majors based around Data Science [15], an arguably more authentic and useful context. However, because it was designed to be accessible for a wide audience, its learning objectives are targeted at a lowest common denominator of students and covers a limited set of topics.

Joyner's open MOOC-based curriculum is perhaps closest in our vision to an open introductory curriculum [14], and should be seen as a comparable project. Joyner's work is still being refined and disseminated, but preliminary reviews suggest there are many fine-grained differences in the approaches (e.g., different styles of material presentation, different content delivery platforms). For now, the course seems to be more biased towards its MOOC roots, without as much emphasis on the in-classroom perspective. Our hope is to design a curriculum that can be easily adaptable to instructors in traditional university settings. Another similarly high-quality open curriculum is the Exploring Computer Science program [8]. Although their materials are extremely well designed and articulated (including both assessments and learning objectives alongside their teaching field guide), the expected audience is primary and secondary schools. A number of companies and private entities have published free online curriculum for learning Python (e.g., CodeCademy, Coursera), but the materials are rarely open-source and are not tuned for university settings either. Evaluations of these curriculum are internal and not publicized.

There have been a number of papers that describe parts of curricula, projects, and assignments. There are even repositories (e.g., EngageCSEdu [19] and Ensemble [5]) of such materials. While many of these can become invaluable components of highly effective courses, it is critical to understand that, from an Instructional Design perspective, these are only *pieces* of a curriculum, whether they be interactive textbooks (e.g., [18]), lecture slides, or collections of auto-graded programming problems (e.g., CodingBat [22]). We argue that a curriculum is the sum of a large number of course artifacts that are designed to provide a series of learning experiences to learners, mediated into concrete course offerings by instructors. It is this sum of pieces that we seek to develop, disseminate, and evaluate in this publication, not individual components or ideas.

3 CONSTRUCTING A NEW CURRICULUM

After reviewing the above curricula, we chose to move forward with developing our own course. We do not claim that any particular element of our courses' design is ground-breaking. This is intentional – we seek to incorporate best practices and existing material. Instead, the major distinction of our curriculum from the prior work is the intersection of 3 principles:

(1) Materials should be documented and created in such a way that they can be shared externally in an editable way.
(2) Modern pedagogical techniques should be used to teach modern topics supported by modern technology.
(3) The course must scale to hundreds of students while still fitting within the conventional university course model.

3.1 Instructional Design

To satisfy our first principle, the lead course designer worked with an instructional designer and followed a formal Instructional Design model based on Dick & Carey [7]. Instructional Design is a systematic process akin to Software Engineering [3] that aligns learning outcomes, assessments, and instruction while simultaneously encouraging documentation and evaluation along the way.

We began with topics drawn from various sources, including syntactical features (e.g., how to write `for` loops), software engineering strategies (e.g., writing unit tests), computational theory (e.g., runtime complexity), and high-level critical thinking (e.g., solving open-ended problems). We oriented our topics around language features as an organizational aid: Figure 1 visualizes their order, divided into two strands of Data and Algorithms (branching off of Program Fundamentals). From the topics, 234 subordinate learning objectives were detailed and grouped, all descended from the following terminal learning objective: *"Create simple Python programs that solve problems."*

Parallel to the development of learning objectives, we conducted an analysis of the learners so we could establish appropriate criteria: student demographics and instructor interviews led to a determination of expected student prior skills and motivations. Informed by these analyses, we then transformed each subordinate learning objective into one or more assessment questions. Finally, instructional materials were prepared based directly on the objectives and assessments. These materials incorporated a mixture of presentation (asynchronous video lessons and synchronous lecture slides) and practice opportunities (programming problems, quiz and clicker

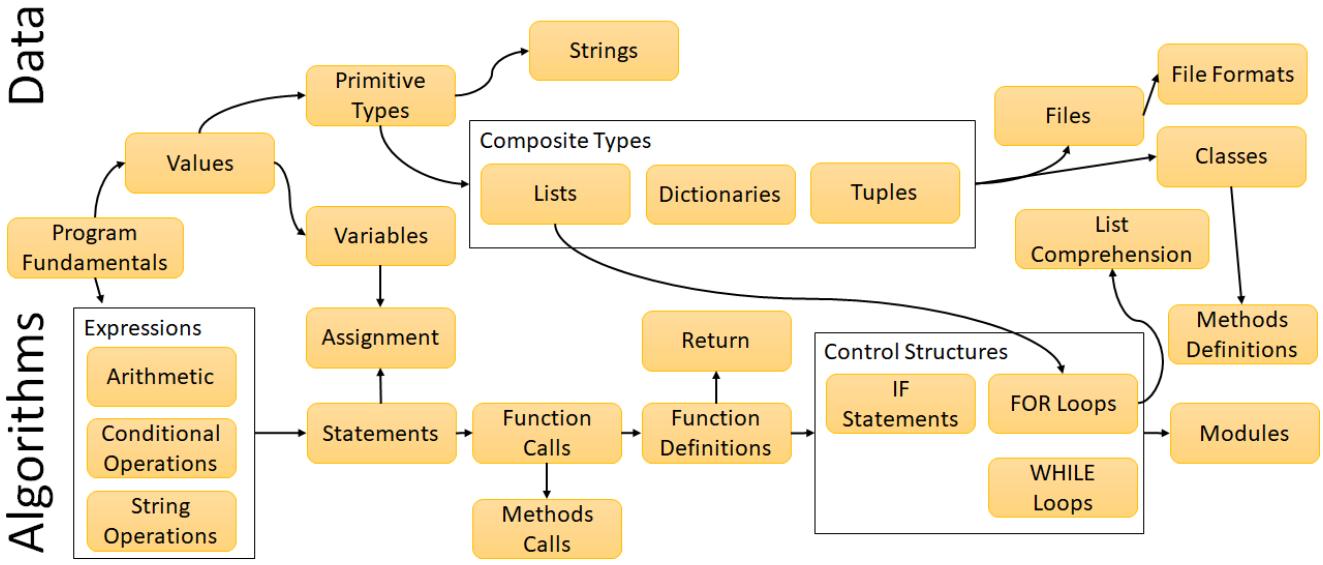

Figure 1: An Overview of the Sneks Curriculum, from a topics perspective

234	Learning objectives
281	Quiz Questions
170	Programming problems
51	Video lessons
45	Annotated Lecture Slides
6	Projects
4	Essay-based Ethics Assignments
2	Final Assessments
1	Teacher's Guidebook

Figure 2: Quantity of Curricular Resources

questions, essays, and projects). Along the way, we documented our instructional strategies into a Teacher's Guidebook (publicly available at https://acbart.github.io/python-sneks/). Figure 2 breaks down the complete set of created resources.

3.2 Pedagogy and Materials

In this section, we detail the pedagogy selected to deliver the course content. The Sneks curriculum is not meant to be revolutionary or experimental, it is meant to be well-engineered and clearly articulated. To that end, and in satisfaction of our second principle, most of the courses' pedagogy was selected based on best practices from the literature. Per our third principle, course materials were designed to fit into the framework of a typical undergraduate university course. For our purposes, this means a 14-week semester where students meet in-person for lecture and have access to a modern online learning management system (i.e., Canvas). However, we also wish to handle large classes of hundreds of students, so many decisions were motivated by scalability concerns.

To handle the large class sizes, a team of (largely undergraduate) teaching assistants supports the instructors by attending class, with a 30:1 student-to-staff ratio. The instructional staff met weekly to preview upcoming lessons, review previous assignments, coordinate grading, and receive pedagogical training. The training material was based off the curriculum by Tychonievich [27]; our strategies for deploying it are included in the Teacher's Guidebook.

The majority of the course content is delivered in a Flipped Classroom style [4, 13] via 51 asynchronous video lessons. Based on the guidelines of [10], each video was designed to be short (around 2-4 minutes) and concisely teach a small set of learning objectives. Each video has PDFs, narration, captions, and transcripts available. Lectures slides were also created, but are brief: the in-class lecture component lasts less than 20 minutes. Little didactic instruction is used during this time, instead focusing on Peer Instruction activities [23], live coding demos, and other active learning lecture strategies [24]. The majority of class time is instead dedicated to students completing assignments with help from the course staff.

Each video lesson is associated with a short quiz (averaging about 5 questions) of multiple choice, fill-in-the-blank, and other automatically graded conceptual questions. These quizzes are meant to be delivered in a mastery learning style, with students having repeated opportunities to complete them. Most lessons also have 3-4 programming problems associated with them. Like the quizzes, the programming assignments are automatically graded and paired with immediate feedback to scaffold the learners. These problems are small exercises, typically requiring less than 10 lines of code.

Lessons are organized into modules, and each module pair is capped with a project. Lessons align with the subordinate learning objectives, in the same way that the modules align with the major topics. The nature of the projects vary in topic, scope, and context. However, each attempts to incorporate opportunities for student

agency and creativity, amusing stories, or other motivational elements.

(1) Turtle Art (Function calls): Use functions from the Turtle module to create a piece of art of their own design.
(2) Magical Banking (Function definitions and data passing): Integrate a black-box function and code a sequence of simple functions to make a banking application for wizards.
(3) Survey Statistics (For Loops and Lists): Develop a survey question for an alien overlord, gather responses, and analyze the data using simple iteration patterns.
(4) Text Adventure (While Loops and Dictionaries): Plan and create an interactive text-based adventure game with mutable, structured state based on a simple game framework.
(5) Data Science (Data Processing and Visualization): Analyze a self-selected dataset to generate visualizations and perform open-ended statistical explorations.
(6) Canvas Analyzer (Summative of all course topics): Use the Canvas LMS API to retrieve and analyze mock student data and the learner's actual course data.

Interweaved between the modules are 4 essays oriented around ethical dilemmas in computing. These essays are ordered to situate the students first as consumers of technology and then as producers of technology. Student responses are meant to be brief and are graded with an established rubric. For each prompt, students must 1) summarize a provided situation incorporating an ethical dilemma, 2) take a clear stance, 3) write a coherent argument supporting their stance, 4) reference a relevant ethical theory or guideline, and 5) write with understandable spelling, grammar, and punctuation.

The course is capped with a final exam based directly on the quizzes and programming questions. This exam is administered using the same platform as the other questions, and graded automatically. TA proctors are physically present to prevent cheating.

3.3 External Adoption

External adoption is a primary goal of the PythonSneks curriculum. The courses' components (and their source) are available through GitHub, while the course itself is available on the Canvas learning management system. Two audiences are in mind for these resources: power users who want a la carte access, and casual adopters who want to use the curriculum without modification. We encourage both cases, but request that adopters report their experiences (either informally or formally) using the GitHub Issue tracker and by filling out the Experience Report forms available in the repository.

Very little space in this paper is dedicated to the technology used in the course. Although our Teacher's Guide covers a number of options and our own reasoning for using various tools and platforms, our goal is to decouple the curriculum from the tools. The justification for this is that tools are transient: for example, the desktop programming environment used by students to complete projects (Spyder) is in the process of being retired – future semesters plan to use the educational Thonny environment, but there are a number of other viable options (PyCharm, Wing, etc.). To this end, many of the assignments and resources are being modified to be compliant with emerging standards developed for the CSSPlice project [1] to encourage adoption without respect to a particular tool.

[1] https://cssplice.github.io/

Project	Easy	Interest	Useful
Turtle Art	3	3	2
Magical Banking	2	3	3
Survey Statistics	2	3	3
Text Adventure	1	4	3
Data Science	3	3	4
Canvas Analyzer	1	3	3

Figure 3: Student Responses about Projects' Attributes

4 DEPLOYMENT AND EVALUATION

A critical element of the Sneks curriculum is pre-planned evaluation of the materials after its deployments. Evaluation is an important phase of any Instructional Design process, and critical to improvement. We present quantitative metrics of the quiz questions and programming problems, along with subjective student survey responses about the overall course components. In addition to the evaluation of the curriculum, we also conducted two quasi-experimental interventions. Issues with the course were identified in the fall as baseline data was collected. Several issues were particularly targeted in the Spring with interventions. In both cases, we attempted to apply emerging theories from the Computing Education literature that we hoped would overcome the issues. Each intervention was led by an undergraduate research assistant.

4.1 Evaluations

So far, the PythonSneks curriculum has been offered in Fall 2017 (281 students) and in Spring 2018 (240 students). Roughly 34% of students were female, and roughly 84% had some kind of prior programming experience (high school, another course at community college, etc.). The DFW (Ds, Fs, and Withdraws) rate was 7.1% in the fall and 7.3% in the spring. In this section, we offer some of the preliminary evaluations conducted of these offerings. The evaluations given here are not meant to be summative or definitive, but demonstrate the range of ways that the course can be measured.

Data was collected through both the LMS' grade systems, the coding platforms logging systems, and from surveys. These surveys were administered every other module to both students and teaching assistants. Student surveys asked questions about which learning resources were particularly helpful to their learning, their habits with regards to the video lessons, overall course motivation, and their suggestions for improvements to the modules they just completed. TA surveys asked questions about particular student misconceptions, useful teaching strategies, and suggestions for revising the most recent modules.

Figure 3 gives a summary of student responses to survey questions about the six course projects. For each project, students chose from 5-point likert style questions (e.g., "Very hard", "Somewhat hard", "Neither hard nor easy", "Somewhat easy", "Very easy" map to 0-4). The number shown is the median numeric response to each question, with the final column as a cumulative score for the project. Most projects were at least somewhat interesting to students, but they ranged in difficulty and usefulness. From a revision viewpoint, the final project's difficulty may be appropriate, but it would be

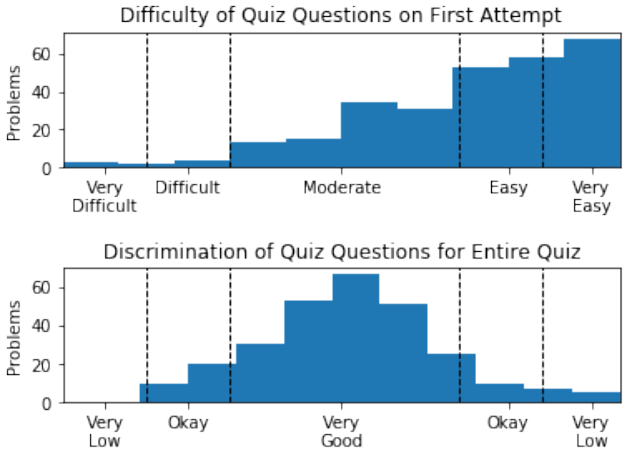

Figure 4: Summary of Quiz Questions

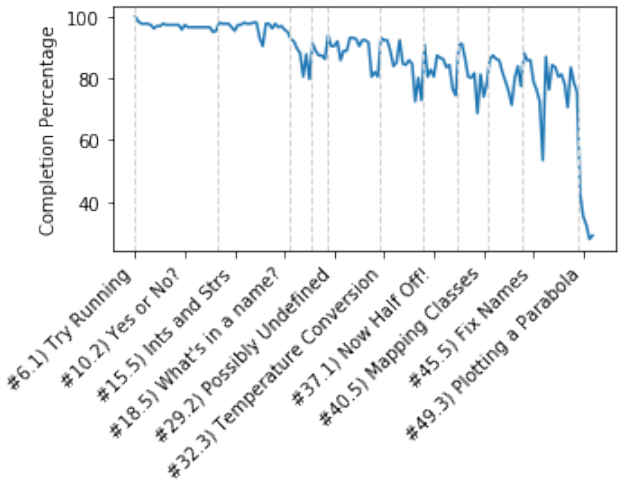

Figure 5: Completion of Programming Problems over Time

ideal if the project's interest or usefulness were higher to balance this out (as the Text Adventure accomplishes).

Figure 4 are histograms of the quiz questions' difficulty and discrimination. In this case, difficulty was measured as the proportion of students who gave a correct answer on their first attempt at a question; the X-axis has been labeled with an interpretation of these numbers. Discrimination is given as the Pearson correlation of each students' score with their overall quiz score, indicating how well this particular question predicts the quiz. Using these metrics, individual questions can be assessed for their fit. The distribution of question difficulty does suggest that most questions skew towards being easier, but that they tend to discriminate fairly well.

Figure 5 gives a time-oriented view of programming problem completion rate. The vertical lines indicate the start of a new module, while the blue line is the percentage of students who have completed a given problem. Although completion rates trend well over most of the course, they nosedive when the assignments overlap with the final project, possibly indicating a need to bifurcate the start of the project from the remainder of the assignments.

Upon finishing the second deployment of the Sneks curriculum, a total of 41 major redesign considerations were identified. These considerations have been divided into low (16), medium (21), and high (4) priority issues, and into categories such as organizational issues (*"TAs sometimes were unprepared: allocate time for TAs to try assignments as part of their job responsibilities"*), topical issues (*"String traversal is explained better in lecture than in the video lessons, so relevant content should be moved."*), and lingering student misconceptions (*" Students believe that variables passed into a function call must have the same name as the formal parameters"*).

4.2 Action Design Research

The next two sections describe interventions staged during the second offering of the course. Educational research is frequently confounded by the rapidly changing target space and variability between semesters, frustrating efforts to control populations for experimental validity. We share the beliefs suggested by Nelson and Ko [21], that the quest for unarguably valid experimental results is a detriment to progress in perfecting designs within computing education. The Sneks curriculum is ultimately a design artifact, and so frequently decisions were made that could affect the sanctity of our experiments. We allowed modifications in a systematic way that would contribute to the research space, while acknowledging the dynamic nature of educational settings. Therefore, we approach our modifications from a Action Design Research (ADR) perspective [26]. When ADR is used in an educational setting, the result is similar to Instructional Design: an iterative cycle of identifying problems in existing curricula, implementing changes, and observing the impact. Therefore, the interventions presented in this section should be taken with the caveat that the studies are static-group comparison designs, and understood for their limitations.

4.3 Worked Examples Intervention

Problem: Students struggle with difficult programming problems.

Theory: Worked Examples are examples of similar problems, consisting of a "problem formulation, solution steps, and the final answer itself" [25]. Prior work suggests providing Worked Examples with clear subgoal labels could help students deconstruct problems and improve performance [16, 20].

Hypothesis: Providing high quality Worked Examples could help students complete more problems faster and with less stress.

Research Questions:

(1) Do students take advantage of Worked Examples?
(2) Do Worked Examples improve performance?
(3) Do students find Worked Examples helpful?

Intervention: In the second course offering, 8 problems were selected as the hardest (based on the metric that it took students longer than average). Worked Examples were created for each problem and made available to students in the problem prompt.

Data Collection: Quantitative Data was collected via exercise completion rates and student interaction in the coding platform. Qualitative Data was collected via a survey on student opinion and usage of Worked Examples.

Results: Usage of the Worked Examples varied between problems, but roughly 42% of students took advantage of them. Surprisingly, students who took advantage of Worked Examples complete problems at roughly the same rate as those who did not. Students who used WEs actually took significantly more time to complete problems; focusing on students with no prior experience, we see those that used the WE actually took over twice the time as those who did not. However, almost 64% of students found the WEs to be helpful to their learning, with only 18% explicitly disagreeing.

Conclusion: The Worked Examples seemed to have minimal effect on the completion rate of problems, and possibly even negatively affected students by slowing them down, even though many students used them and found them helpful.

Limitations: First, all problems had high completion rates (80-90%) even before introducing Worked Examples. Second, the non-random assignment of students in the treatment could hide a more positive effect size. Third, it is possible that our WEs were more like tutorials than WEs.

4.4 Structured Groups Intervention

Problem: Students report that they do not feel prepared to begin a large-scale project

Theory: Micro-classes group students within a large lecture hall, previously shown to foster community [1].

Hypothesis: If students are given structure to form groups naturally, the enhanced community will contribute to their preparedness

Research Questions:

(1) Do groups increase students' sense of community?
(2) Is students' sense of community associated with their preparedness to begin a complex project?

Intervention: On the first class period after Project 2 began, the class was divided into 8 sections, each managed by a TA. Within each section, students were encouraged to organically form small groups. Students then worked together to answer questions about the upcoming project. Afterwards, students were prompted to begin working on the project together, with an emphasis on using student community to solve any problems they may come upon.

Data Collection: After two classes of project work, students took a Likert-style survey to assess their readiness to begin the project, their sense of classroom community, and if their sense of community was helpful in solving problems. This same survey was administered after the final project in the previous semester

Results: As shown in Figure 6, students reported a significantly (as evaluated by a Mann-Whitney U test) increased sense of community relative to the prior semester, but did not feel significantly more prepared to begin the project. There was a significant but moderate Spearman Rank Correlation of .41 between these metrics.

Conclusions: Although working with others did make a majority of students feel more prepared to start the project, the increased sense of community did not seem to help students feel more prepared to start their project.

Limitations: Data was never collected directly on whether students struggled to get started with each project, and the surveys were administered between different semesters for different projects. Finally, A possible limitation of student-selected peer groups is that novices working with novices may not feel that they have any new information, and will still need additional assistance.

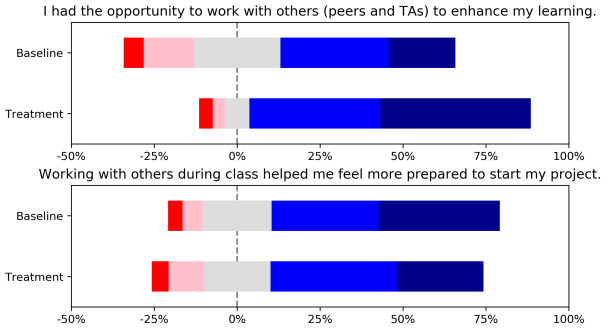

Figure 6: Students' Perceptions of Structured Groups

5 RECOMMENDATIONS

In this section, we describe lessons learned when developing.

1) Follow Instructional Design processes: Not only does ID provide a clear, systematic process to guide novice course developers, but it encourages high quality material that is well-aligned.

2) Design for privacy: As course resources are developed, consider how you will maintain question answer security and anonymize data for potential research publications and future course analysis. Once released, data can be difficult to remove from records.

3) Consider evaluations: How will you assess your questions for their quality? How will you know if students enjoyed your projects? Which resources were most helpful to students? Do not rely on anecdotal evidence, but collect concrete data.

4) Plan for a lengthy development process: As is frequently recognized, course development is a lengthy process. Course development began the summer before the course's first offering, and continued throughout despite full-time commitment from the primary developer. In the latter portions of the course, projects were frequently completed shortly before they were deployed.

6 CONCLUSIONS

In this paper, we describe an Instructional-Designed, open-source curriculum consisting of a large number of formal learning objectives, aligned assessments, and learning material. Although the Python Sneks curriculum has not been externally validated or summatively evaluated, it is not meant to represent a final version. In some sense, the curriculum is a straw-man, ready to be knocked down by more sophisticated curriculum with superior learning objectives and activities. We hope that our reporting on it will motivate not only external adoption, but also the creation and dissemination of other university-level open-source curricula that follow formal Instructional Design processes.

ACKNOWLEDGMENTS

The authors would like to thank the Virginia Tech Technology-enhanced Learning and Online Strategies (TLOS) group for their support in developing this course.

REFERENCES

[1] Christine Alvarado, Mia Minnes, and Leo Porter. 2017. Micro-Classes: A Structure for Improving Student Experience in Large Classes. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. 21–26.

[2] Austin Cory Bart. 2017. What's Your Favorite "Introduction to Computing" with Python Curriculum? (2017). SIGCSE-Members Electronic Mailing List; posted on 27 Apr 2017.

[3] Austin Cory Bart and Clifford A Shaffer. 2016. Instructional Design is to Teaching as Software Engineering is to Programming. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. 240–241.

[4] Jennifer Campbell, Diane Horton, Michelle Craig, and Paul Gries. 2014. Evaluating an Inverted CS1. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '14)*.

[5] Lillian Boots Cassel, Ed Fox, Frank Shipman, Peter Brusilovsky, Weiguo Fax, Dan Garcia, Greg Hislop, Richard Furuta, Lois Delcambre, and Sridhara Potluri. 2010. Ensemble: enriching communities and collections to support education in computing: poster session. *Journal of Computing Sciences in Colleges* 25, 6 (2010), 224–226.

[6] Benny Chor and Rani Hod. 2012. CS1001.Py: A Topic-based Introduction to Computer Science. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '12)*. 215–220.

[7] Walter Dick, Lou Carey, and James O Carey. 2005. The systematic design of instruction. (2005).

[8] Joanna Goode, Gail Chapman, and Jane Margolis. 2012. Beyond curriculum: the exploring computer science program. *ACM Inroads* 3, 2 (2012).

[9] Philip Guo. 2014. Python is now the most popular introductory teaching language at top us universities. *BLOG@ CACM, July* (2014), 47.

[10] Philip J Guo, Juho Kim, and Rob Rubin. 2014. How video production affects student engagement: an empirical study of MOOC videos. In *Proceedings of the first ACM conference on Learning@ scale conference*.

[11] Mark Guzdial. 2003. A media computation course for non-majors. In *ACM SIGCSE Bulletin*, Vol. 35. 104–108.

[12] Mark Guzdial. 2016. Where are the Python 3 Libraries for Media Computation. (2016). https://computinged.wordpress.com/2016/08/19/where-are-the-python-3-libraries-for-media-computation/

[13] Diane Horton and Michelle Craig. 2015. Drop, Fail, Pass, Continue: Persistence in CS1 and Beyond in Traditional and Inverted Delivery. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15)*.

[14] David A. Joyner. 2017. Congruency, Adaptivity, Modularity, and Personalization: Four Experiments in Teaching Introduction to Computing. In *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale (L@S '17)*. 307–310.

[15] Dennis Kafura, Austin Cory Bart, and Bushra Chowdhury. 2015. Design and Preliminary Results From a Computational Thinking Course. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*. 63–68.

[16] Lauren E Margulieux and Richard Catrambone. 2014. Improving problem solving performance in computer-based learning environments through subgoal labels. In *Proceedings of the first ACM conference on Learning@ scale conference*.

[17] Cindy Marling and David Juedes. 2016. CS0 for Computer Science Majors at Ohio University *(SIGCSE '16)*. 138–143.

[18] B Miller, D Ranum, J Elkner, P Wentworth, AB Downey, C Meyers, and D Mitchell. 2012. How to think like a computer scientist. *Runestone Interactive Project (http://interactivepython. org/courselib/static/thinkcspy/index. html)* (2012).

[19] Alvaro E Monge, Cameron L Fadjo, Beth A Quinn, and Lecia J Barker. 2015. EngageCSEdu: engaging and retaining CS1 and CS2 students. *ACM Inroads* 6, 1 (2015), 6–11.

[20] Briana B Morrison, Lauren E Margulieux, and Mark Guzdial. 2015. Subgoals, context, and worked examples in learning computing problem solving. In *Proceedings of the eleventh annual international conference on international computing education research*.

[21] Greg L. Nelson and Andrew J. Ko. 2018. On Use of Theory in Computing Education Research. In *Proceedings of the 2018 ACM Conference on International Computing Education Research (ICER '18)*.

[22] Nick Parlante. 2015. CodingBat. *Com (Retrieved 1/08/2011 from http://codingbat. com, 2011)* (2015).

[23] Leo Porter, Dennis Bouvier, Quintin Cutts, Scott Grissom, Cynthia Lee, Robert McCartney, Daniel Zingaro, and Beth Simon. 2016. A Multi-institutional Study of Peer Instruction in Introductory Computing. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16)*.

[24] Kate Sanders, Jonas Boustedt, Anna Eckerdal, Robert McCartney, and Carol Zander. 2017. Folk Pedagogy: Nobody Doesn'T Like Active Learning. In *Proceedings of the 2017 ACM Conference on International Computing Education Research (ICER '17)*.

[25] Ben Skudder and Andrew Luxton-Reilly. 2014. Worked examples in computer science. In *Proceedings of the Sixteenth Australasian Computing Education Conference-Volume 148*.

[26] Ernest T Stringer. 2008. *Action research in education*. Pearson Prentice Hall Upper Saddle River, NJ.

[27] Luther A. Tychonievich. 2017. Training Course for Teaching Assistants in Computing. (May 2017). https://www.cs.virginia.edu/luther/ta-training