Towards Measuring Basic Coding Proficiency

Summary

This report explains Code.org's investigations into whether or not students using its courses designed for K-5 were achieving any level of "basic coding proficiency". Since the course materials weren't originally engineered to measure coding proficiency, nor was the curriculum designed with specific coding proficiency targets in mind, our challenge was to see what we could learn from data we had on hand. We have collected a relatively massive amount of user activity data over the year, and despite the fact that no universal definition of "coding proficiency" exists for this age group, we wanted see if we could at least gain some insight into how well students were doing with the coding concepts presented across the courses we offer on our platform. This paper gives background about the puzzles in our courses and the kinds of data we collect, explains our methods for tagging puzzles for concepts and difficulty, and defining what "proficiency" means for our platform. Finally we present a few of our results and describe learnings and future work.

This report describes a starting point for us as our organization begins to turn its eye toward more rigorous experimental designs, statistical analysis and modeling to measure student learning. We have some ideas about how to improve our work but we also invite feedback and others' ideas about what we could be doing. If you'd like to get involved please contact data@code.org

Background

Code.org's Computer Science Fundamentals program consists of five courses geared toward K-5 students. Courses 1-4 are designed to be approximately 20 class-hours long, including "unplugged" activities which are lessons that happen off the computer. Many of our materials are designed to be taught by a teacher in a classroom. They are typically taught once or twice a week over the course of a semester.

The primary feature of these courses are programming puzzles that guide students through using a certain programming concept (like a loop) to solve a small problem. Our puzzles are similar to what are known as "Parson's Puzzles"[1] in which a small challenge or puzzle is laid forth, and you are presented with a set of coding blocks as tools to solve that problem. This approach to learning programming makes it impossible to make certain kinds of typical "beginner mistakes" related to syntax and, in theory, allows the learner's cognitive load to focus more on how to use code as a problem-solving medium.

[1] Dale Parsons and Patricia Haden. 2006. Parson's programming puzzles: a fun and effective learning tool for first programming courses. https://dl.acm.org/citation.cfm?id=1151890

Computer Science Fundamentals courses are broken into “stages” (roughly speaking, each stage is one lesson or class-hour) and each stage is comprised of a sequence of “levels” - usually 10-15 per stage.

Most levels are coding levels (which we’ll call “puzzles” for this paper) but not all. There are whole stages devoted to unplugged activities, and there are a handful of “quiz”-style questions like matching or multiple choice. Below is table that summarizes the courses, showing how many stages and levels each has along with programming concepts that students encounter in the course.

CS Fundamentals Course	Who’s it for?	Concepts
Course 1 18 stages 119/146 coding levels	**Pre-readers (K-1)** **NOTE:**this course was not used in measuring student proficiency	Instruction Sequencing, Loops
Course 2 19 stages 128/159 coding levels	**Elementary school students** Recommended starting point for students in 2nd-5th grade who have never tried CS before	Sequencing, Loops (+ small amount of Conditionals and Events)
Course 3 21 stages 148/177 coding levels	**Elementary school students** Designed for students 3rd-5th grade after completing course 2	Sequencing, Loops, Conditionals, Events, Functions
Course 4 22 stages 161/167 coding levels	**Elementary school students** Designed for 5-6th grade students after completing course 3	Sequencing, Lots of Loops, Conditionals, Events, Functions with parameters, Variables
Accelerated Course 20 stages 109/109 coding levels	**Middle and high school students** Designed for older students new to computer science. The course covers most of the material in courses 2 and 3 (and a few concepts from course 4) in an accelerated format.	Sequencing, Loops, Conditionals, Events, Functions

A Brief History of our Data Investigations

Over time, this collection of courses became popular beyond anything we could have imagined. The puzzles contained in the courses have been attempted millions of times by millions of students and we have logged much of that student activity, but haven’t done much with it, yet.

Historically we reported superficial measures like the number of active student accounts or lines of code written. Going another iteration deeper we began to see if we could track student engagement in the courses. We tracked how many puzzles a student had solved and roughly mapped CS concepts to stages (groups of puzzles that comprise a “lesson”) of each course.

Looking at engagement data coupled with concept-tagging gave some further insight about whether students were progressing through the course, which puzzles students got stuck on or did quickly. We could see where activity tailed off, presumably due to lack of interest or due to puzzles being too difficult. The insights gained from these investigations led us to develop more targeted hints and error messages for students in an effort to get them “over the hump.”

However, these measures were far too coarse-grained to tell us much beyond basic student activity, and whether they were they actively using a puzzle or not. It did not differentiate the difficulty of the puzzle or student performance.

Looking for student learning

We wanted to take a step further to see if students in our CS Fundamentals courses were actually learning and applying CS concepts. Most of the students using these courses are in elementary school and coding for the first time. So while the courses ask students to solve puzzles using very basic coding concepts, anecdotally, we hear from teachers and professional development facilitators that students *are learning* the fundamentals of sequencing instructions, loops, conditionals, events, and more.

As we looked into measuring this, we could not find an externally validated assessment for measuring computer science for this age group. Furthermore, the research on learning trajectories for students in this age range is at best in the nascent stages.[2] Measuring actual student learning in anything is a difficult and rigorous process and one for which our courses and puzzles were not originally designed. The CS Fundamentals puzzles are designed to help students learn to use basic coding concepts like sequence, iteration, and conditional logic, not for assessments.

Our challenge then became: given all of this student activity data can we detect whether or not students are demonstrating any “proficiency” at coding? Could we take this student activity data, combined with our knowledge about the puzzles themselves, to determine if the user was doing more than passively watching the screen, randomly moving coding blocks around, and hitting run?

While we do not have externally validated assessments “baked” into the course, we do gain some information every time a student solves a puzzle on Code Studio. Among other things we

---

[2] *K–12 Computer Science Framework*. (2016). Chapter 10 - The Role of Research in the Development and Future of the Framework. Retrieved from http://www.k12cs.org

capture: timestamps for starting, attempting and finishing a puzzle; the student's code from each attempt; which hints they viewed, and so on.

We know that we can't make any strong claims about (capital L) learning or what caused it. There are too many environmental variables - students working together, teacher interactions, homework, parents, direct instruction, real class v. playtime - that we don't have any insight into and will never be in our control. The best we can do is look at our puzzles as though they are individual test or quiz questions and see if a student got them right. Since our puzzles have a wide range of difficulty in terms of both the core concepts at play and the amount of guidance provided, we thought we could develop a cutoff point or bar over which we could theoretically say a student has demonstrated some amount of "coding proficiency".

Methods

Defining "Coding Proficiency"

What does "coding proficiency" mean to us? In a nutshell it means: the student can apply coding concepts (sequencing, iteration, functions, etc.) to solve some number of coding puzzles on our platform. Because of the lack of validated assessments for grades K-8 computer science we want to be cautious about the term "proficiency". We use it as a shorthand, but it has a narrow and specific definition for our purposes here. You should continue to read to understand how we measured things and where we set the proficiency bar.

Producing the Concepts-Difficulty Matrix

About a year ago, we published this blog post outlining a new approach of attempting to measure student "proficiency". We looked at the CS Fundamentals curriculum and identified 6 broad programming concepts addressed: sequencing, loops, conditionals, events, functions, and variables.

Since then we also broke some of the concepts up into more descriptive parts. For example, "loops" (shorthand for: iteration) is a pretty big concept that is presented in our environment in several different ways: repeat loops, repeat until, for-loops and so on. Similarly, we broke up functions to differentiate using functions with parameters. In the end our set of concepts was:

Sequencing (Algorithms)	Conditionals	Variables	Loops	Functions	Events
			• Repeat • Repeat Until / Repeat While • For Loops	• Defining and Calling • Defining and Calling w/Params	

The courses also present concepts such as debugging, collaboration, internet safety, and more, but those concepts and lessons are often “unplugged” or happen in the classroom outside the the Code Studio environment. The metrics presented here only measure student performance on puzzles in Code Studio related to computer programming.

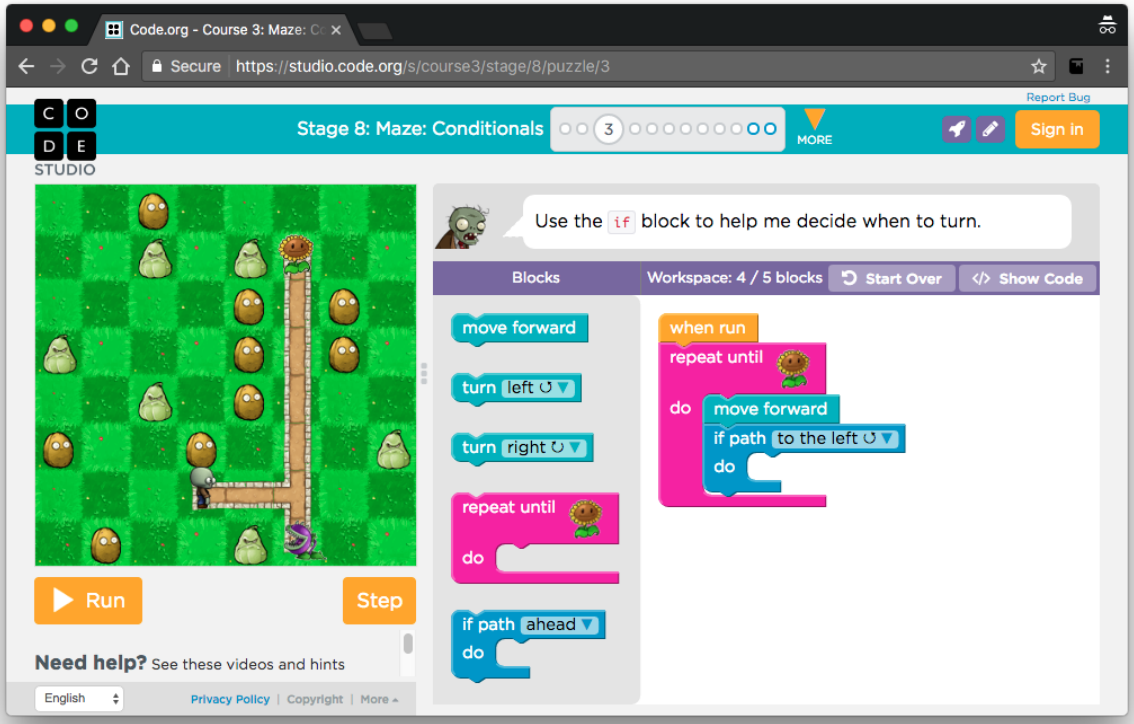
We decided to rate puzzles for “difficulty” on a 5-point scale, with 1 being “easy” and 5 being “hard”. Generally speaking, a rating of 1 meant the puzzle required virtually no problem solving - it might simply ask the student to click a button and observe the results, a stepping stone in a longer progression. A rating of 5 meant the puzzle was very challenging, or the solution required sophisticated application of concepts to get it right or to produce the optimal solution.

We produced a matrix of the concepts v. difficulty ratings. Theoretically, each cell contained a list of items, typical for our puzzles, that we thought matched the given concept and difficulty. Here’s a snippet that shows the difference between difficulty 1 and 5 for repeat loops:

Difficulty	Concept: Repeat Loops
1	a) Move existing code into a loop with prompting b) Construct a simple (1-instruction) loop with prompting c) Identify areas of a program that repeat exactly d) Modify or delete a single loop instruction with prompting e) Change loop run parameter
•••	•••
5	a) Use a single nested loop layer without prompting b) Use multiple nested loops in sequence with or without prompting c) Use nested loops more than one layer deep with prompting

We shared the resulting concept-difficulty matrix for review in May of 2016.
You can read more the work that led up to that in this page we created to define: What is Basic Coding Proficiency?

Once we had the matrix, we set about tagging each of the ~500 puzzles across all of the CS Fundamentals courses to associate each puzzle with one or more concepts, and for each concept, a difficulty rating. For example, the puzzle shown below has different levels of difficulty for different concepts. It is tagged: *Sequencing-4, Repeat Until Loops-2* and *Conditionals-2.*



https://studio.code.org/s/course3/stage/8/puzzle/3

Once tagged, we turned the crank on the database to produce all of the student activity data side by side with the concept-difficulty rating of each puzzle they completed. Our hope was that doing even superficial analysis of this new data would yield more insights and point us in the direction of determining some level of “coding proficiency”.

First attempts at Measuring student performance in Code Studio

The first step of our analysis was simply trying to get a handle on what a student has demonstrated when they complete a particular puzzle. We expect that this will be an area where we continually iterate and improve our understanding, but as a first attempt we looked at three factors:

1. **Did they complete the puzzle correctly?** On many puzzles, the student cannot complete the puzzle without using the concept correctly. Students who don’t get it may try over and over again or skip to the next puzzle leaving it temporarily incomplete. We only counted puzzles that were solved correctly.

2. **Did they complete the puzzle with optimal block counts?** Most of the puzzles on Code Studio show an “optimal” block count - the expected number of lines of code in the

desired solution. We chose to use optimal block count as an indicator of "proficiency" since in many cases the optimal block count is not achievable without applying the desired concept - this is especially true for puzzles with loops. For example, you can pass the puzzle without a loop by manually repeating lines of code over and over, but only students who use loops can pass with the optimal block count.

3. **Did they complete the puzzle without using hints?** Many puzzles offer one or more hints which a student may elect to view by clicking on a "hint" button. We only count a puzzle as evidence of student proficiency if the student solves it without using any hints. This was a hard choice to make. These hints vary tremendously in the amount of help they give, from a gentle nudge in the right direction to something that basically gives away the answer. And hints also have their own progressions. It's worth noting that our system does give some custom error messages when an incorrect solution is attempted. Custom error messages are a form of hinting. So for our analysis, to keep it simple, we only counted puzzle completion toward proficiency if a student did not ask for any hints.

There are many additional factors that may be indicators of proficiency such as length of time to solve the puzzle, number of attempts, or particular solutions. However, as a first attempt at measurement, we only used indicators that unambiguously indicated success. A high number of attempts at a puzzle, for example, may indicate a student slowly building up his/her code and testing as they go along (a common best practice) or it could be a student struggling to find the correct answer.

Drawing the line on proficiency: Concepts + Difficulty

Defining "proficiency" using our tagging system is both somewhat arbitrary and nuanced. But as a high-level metric we wanted to be able to say *X number of students using our platform demonstrated "basic coding proficiency"*. We have to draw a line somewhere. This required us to make some choices about:

1. How many different puzzles did a student have to complete correctly?
2. Across how many different concepts?
3. At what level of difficulty?

How many puzzles? Three.

For each puzzle a user completes correctly, with optimal block count, and without hints, we give that student "credit" for the associated concept and difficulty. For example: if a certain puzzle had tags for: *Sequencing - 2* and *Repeat-Loop - 4* then that student would get 1 "point" toward sequencing at difficulty level 2, and one point for Repeat-Loops at difficulty 4.

For our first pass we set the number of puzzles to solve at three (3) to say the student "demonstrated proficiency" for a given concept-difficulty pairing. It's important to note that

solving a higher-difficulty puzzle *also* counts toward the student's proficiency at a lower levels of difficulty. So, for example, the student who completes a puzzle tagged *Repeat-Loop - 4* gets "points" toward proficiency for difficulties 1-3 as well. Thus:

> *A student has demonstrated proficiency at concept X, difficulty Y when they have solved three puzzles tagged with concept X and difficulty Y or higher.*

It's also worth noting that some pairs, like Sequencing at difficulty 3 are present in hundreds of puzzles, and also very easy -- you might be demonstrate proficiency within only a few puzzles attempted, and have many opportunities to do so. Other pairs, like for-loops at difficulty 5 do not apply to many puzzles, and these puzzles tend to be fairly challenging.

The heat map below shows the total number of concept-to-difficulty mappings across all puzzles that we counted toward "proficiency". Notice that "sequencing" is prevalent - every puzzle that we tagged had some level of sequencing involved.

Total # of Concept-Difficulty Mappings across Courses 2, 3, 4 and Accelerated

		Difficulty				
		1	2	3	4	5
CONCEPT	Sequencing	137	56	199	50	91
	Conditionals	9	19	13	9	8
	Repeat Loop	123	70	58	51	35
	Repeat Until/While Loop	27	15	8	25	1
	For Loop	25	1	7	25	5
	Variables	44	8	39	3	10
	Functions	38	5	24	1	3
	Functions w/Params	14	7	10	8	7
	Events	17	13	12	7	2

* Note that there are 3 sub-categories of loops. Proficiency in any one sub-category was required to achieve "proficiency in loops". Similar is true for functions.

How many concepts? Three.

Basic computer programming requires students to learn multiple concepts that interact with and build on each other, but there is no established or "correct" order in which to learn them. For example, one might learn conditionals before loops and vice versa. Thus, we wanted to look at students who had demonstrated proficiency for *some number* of different concepts.

As an initial estimate we chose to look at students who demonstrated proficiency in three (3) concept areas.

How difficult? Difficulty Level 3.

Tagging puzzles for difficulty was problematic and, when vetted internally, showed some inconsistencies. (See: Weaknesses and Challenges section at the end of this doc). In the end we found that our 1-5 rating whittled down to three useful buckets:

Difficulty 1 - 2	Difficulty 3	Difficulty 4-5
Difficulty 1 and 2 tend to measure measure introduction or basic exposure to a concept or idea. The student does not have to understand the concepts to solve puzzles at these levels. While they may use loops, conditionals, etc., the puzzles are simple or the student is prompted to do it in the right way.	Difficulty 3 tends to be puzzles where students have to apply loops, conditionals or other concepts, toward solving a problem. The instruction prompts for these puzzles also tend to be less explicit or prescriptive, and more like an open-ended challenge: "can you solve it?"	Difficulty 4-5 tend to be puzzles that are one or combination of open ended instructions, longer solutions, more complex application of ideas, or even require novel approaches to new situations.

Since Difficulty 3 was the first where students tended to be solving problems rather than simply following instructions, we chose difficulty level 3 as the lower cut off point for demonstrating "basic proficiency". We hope to have many students go beyond this basic level to reach difficulty levels 4 and 5. And, our middle school and high school courses go much further in teaching computer science.

Finally, our definition of "proficiency"

In the end our definition for showing "basic coding proficiency" is:

> *A student has demonstrated basic coding proficiency when they have solved three or more puzzles, for three or more concept-areas, at difficulty level 3 or higher, across one or more CS Fundamentals courses in Code Studio.*

The fact that there is some symmetry to this definition (3 puzzles, 3 concepts, difficulty 3) is purely coincidental. We were not trying to be cute or to develop some kind of mnemonic. Given the distribution of concept-difficulties (see the heat map above) this seemed to be a fair bar for our initial definition of "basic proficiency" and is helpful for visualizing progress, but it also has flaws, described later in this document, which we hope to improve over time.

Results

We have only begun to dig more deeply into the data to look for patterns and we still have more questions than answers. But, we wanted to share some of what we've found so far to begin the conversation.

Dates and Scope of Data

What follows below are results and data collected between **January 1 - December 31, 2016** from student users who were **logged in** to the Code Studio environment. Code Studio does not require users to be logged in order to use the puzzles. A large number of users, roughly a third, use these materials without logging in. However, most of the non-logged in user sessions are show activity on puzzles early in courses, so our assumption is that most of these students either drop out or create accounts before they reach basic coding proficiency.

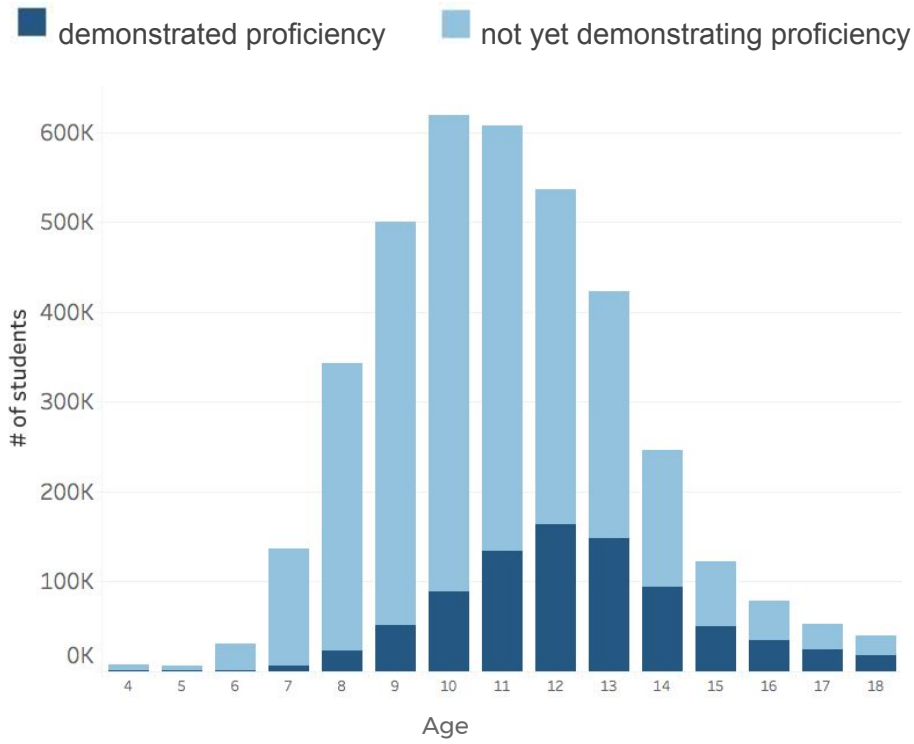
How many students?

	Number of students	% of all student users
Students demonstrating basic coding proficiency	887,829	23%

887,829 students of all ages demonstrated proficiency according to our definition (3 concepts, 3 correct puzzles, difficulty 3 or higher). This number represents roughly 23% of all students who successfully complete any puzzles in our system. Here is a view of how the raw numbers break down with students who are in the K-12 age range[3].

[3] Student ages are derived from birthdate which comes from a variety of sources. It can be self-reported, reported by teacher, or from an external linked account like Google or Facebook. Over 90% of student accounts have an age associated.

Fig. 0 - Number of students demonstrating basic proficiency in 2016 by age



An important note on percentages

Why do “only” 23% demonstrate basic coding proficiency? It’s important to note that the students using Code Studio aren’t all studying as part of a long-term course. Again, roughly a third of the students using Code Studio never even register for an account and their usage cannot be measured in this fashion, even if they complete every single puzzle in every single course. Looking only at students with registered accounts, they fall into three roughly equal-sized buckets in terms of their general activity on our system:

1. Roughly a third use the materials for *less than a day*. Many teachers and schools use CS Fundamentals courses for Hour of Code[4] activities; students work on the first few puzzles for a one-time event and never come back.
2. Roughly a third try puzzles for less than a month, which probably isn’t enough time to complete enough qualifying puzzles.
3. And roughly a third continue for at least a month, which is the group most relevant for the data in this report since it is the most likely that this group represents students of teachers who implemented the course(s) as part their classroom curriculum.

[4] The Hour of Code is an annual event to celebrate and increase participation in computer science by encouraging as many people as possible around the world to spend one hour engaged in some kind of coding activity during Computer Science Education Week (early December). See: https://hourofcode.com/us

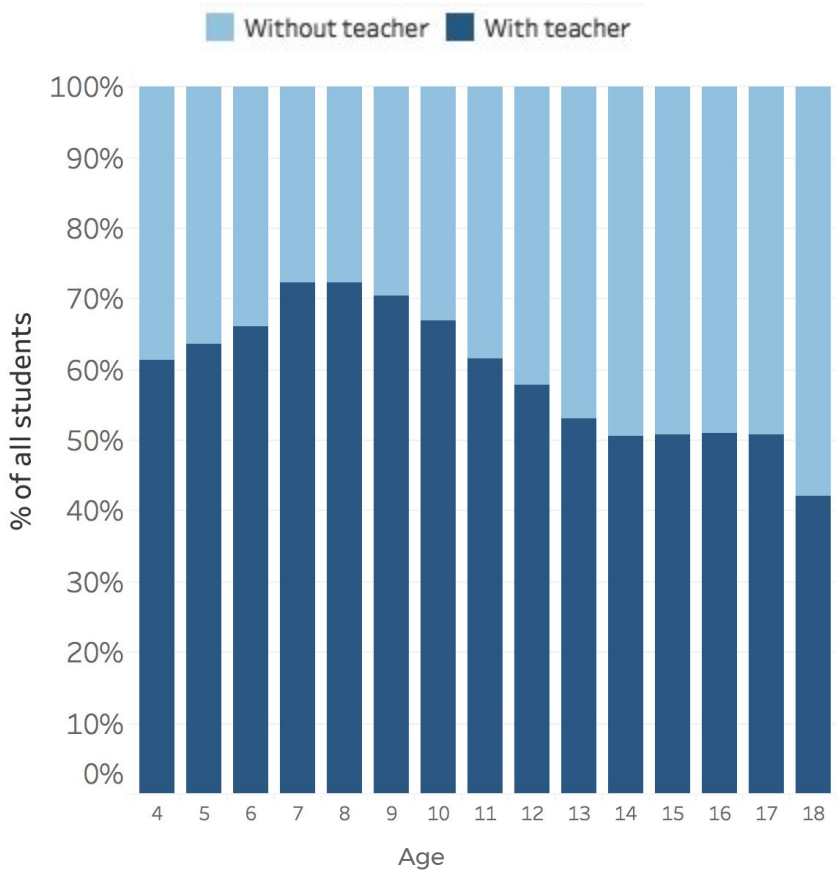
Given the nature of how different students, schools, and teachers use Code Studio, we don't set goals around the *percentage* of these students who demonstrate proficiency, because the fidelity or mode of implementation varies wildly from Hour of Code activities (one hour) all the way to complete course implementations (20+ hours). What's more important to us is the absolute number of students who demonstrate proficiency, and the absolute number of teachers who implement a high-fidelity course in their classrooms.

In the charts below, unless noted otherwise, when we report a percentage, *the denominator is the set of registered student accounts on Code Studio where the students completed at least one puzzle (with optimal block counts and without viewing hints) in any one of the courses where student progress counts towards our measure of coding proficiency during the period of Jan. 1 - Dec. 31, 2016.*

Breakdown between classrooms vs. independent study

From our data we see that most students in the K-12 age range are using the materials "with a teacher" which means that the student account has been assigned to a section that belongs to a teacher account.

Fig. 1 - Students with and without a teacher as a % of all students within their age group



Breakdown by concept and difficulty

The table below shows, for each concept, the raw number of students who solved 3 or more puzzles (with optimal block count and no hints) at or above each level of difficulty. Recall that difficulty 3 is where we set the bar to say a student "demonstrated proficiency" for any given concept, but this table effectively shows what it would look like if we had set the bar at other levels of difficulty.

Fig. 2 - Number of students demonstrating proficiency across concepts and levels of difficulty

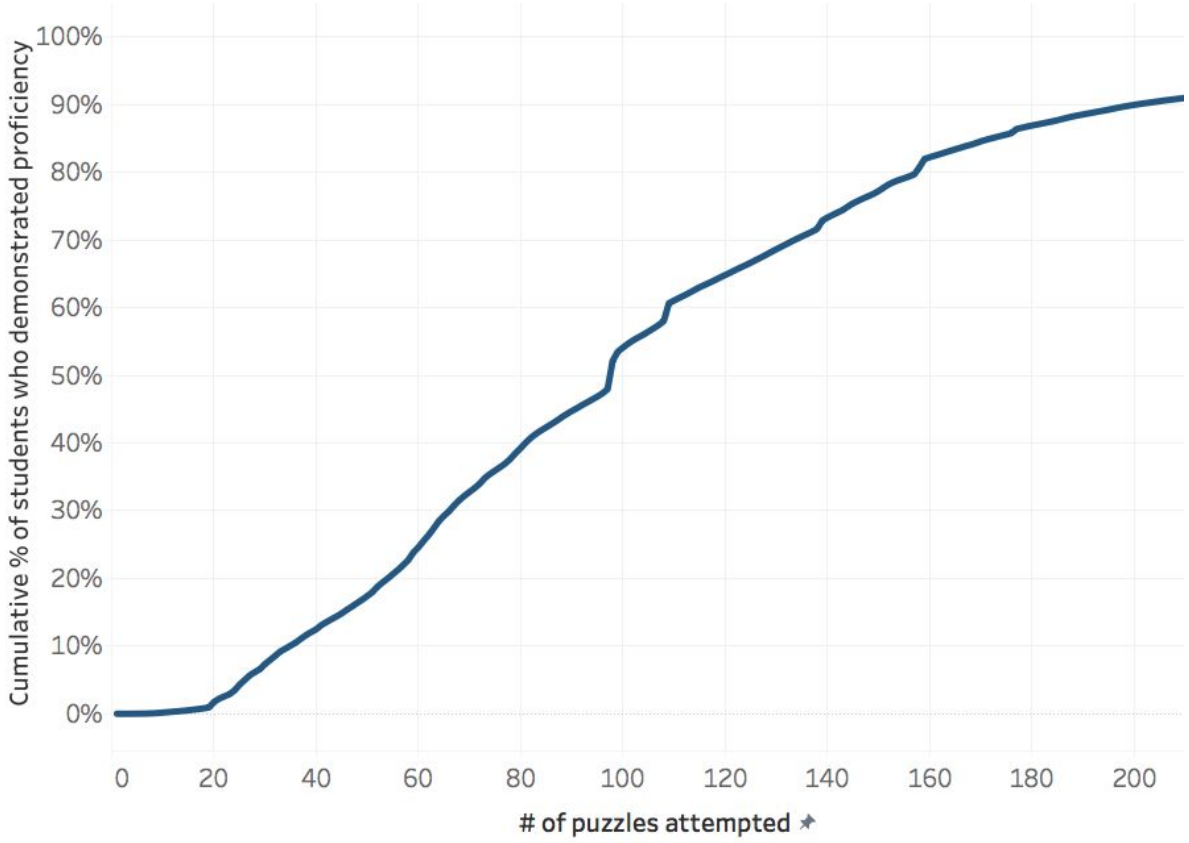
	Difficulty 1	Difficulty 2	Difficulty 3	Difficulty 4	Difficulty 5
Sequencing	3,619,302	3,555,217	3,286,079	1,112,935	981,516
Conditionals	1,018,804	1,013,773	771,871	431,386	93,874
Loops	2,343,097	1,922,722	1,734,299	1,387,980	353,852
Variables	341,402	249,063	146,267	13,906	7,438
Functions	547,077	450,360	313,938	22,290	7,525
Events	582,198	546,671	473,030	106,810	N/A

How long does it take to demonstrate proficiency?

Roughly speaking, a majority of students who reach the proficiency bar do so after working on about 100 puzzles and most have done so by 200 puzzles. Overall, most student activity is confined to one course as the number of coding puzzles in a course ranges from 109 to 161. Many students reaching proficiency in 100 puzzles are older students using the accelerated course which has 109 coding puzzles.

The chart below shows for all students who eventually reach the proficiency bar, how many total puzzles they attempted. For example, of all students who eventually reached proficiency roughly 50% have done so in 100 puzzles or fewer. Note that it does not indicate the moment when they actually reached proficiency. Rather, it shows for a student who reached proficiency, the all-time number of puzzles that student attempted.

Fig. 3 - Cumulative % of students demonstrating proficiency by number of puzzles attempted

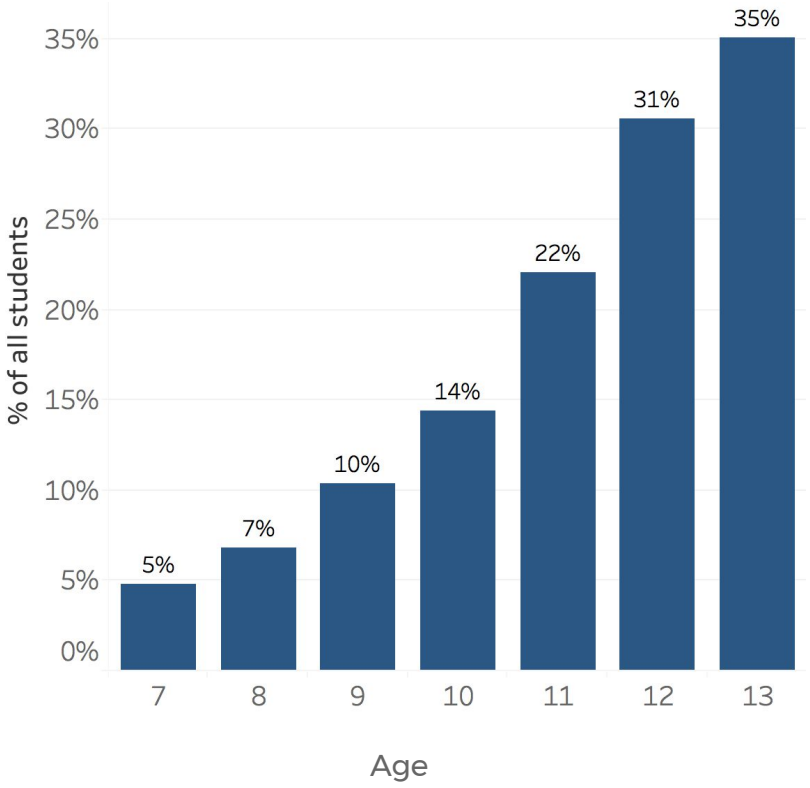


Note: in the graph above we cut the x-axis off at 200 puzzles because the remaining portion of the curve flattens out considerably over another 680 puzzles. The remaining ~10% likely represents students who did every puzzle in many, or all, of the different courses.

Does the age of the student make a difference?

Keeping in mind that 23% of all students demonstrate proficiency, the chart below shows how it breaks down by age. You can see that larger *percentages* of older students reach proficiency, though it's worth noting (see above) that fewer older students overall use the courses. For the chart below we've confined the data to the age range 7-13 which is the age the puzzles were designed for.

Fig. 4 - Students demonstrating basic proficiency as % of all students within their age group



Because of where we set the bar, it's pretty difficult for young students to demonstrate proficiency. We did not count *any* of the puzzles in course 1 (course for pre-readers) toward the proficiency measure. We suspect that older students' reading level, physical dexterity with computer controls and the fact that they are more likely to be using these courses in a classroom setting, contribute to these effects.

Does having a teacher make a difference?

We found students associated with teachers[5] were more likely to demonstrate proficiency. Having a teacher appears to make a slight difference for younger students, but for older students the difference is quite pronounced. There are a variety of factors that may lead to this that we don't have insight into at the moment. For example, a younger student who appears to be working "on their own" might be at home side by side with a parent, whereas an older student might actually be on their own. This is another area of potential future investigation.

[5] Our database keeps track of accounts associated with a "class" that a teacher (self-identified) sets up in Code Studio.

Fig. 5 - Students demonstrating proficiency with or without a teacher as % of all students

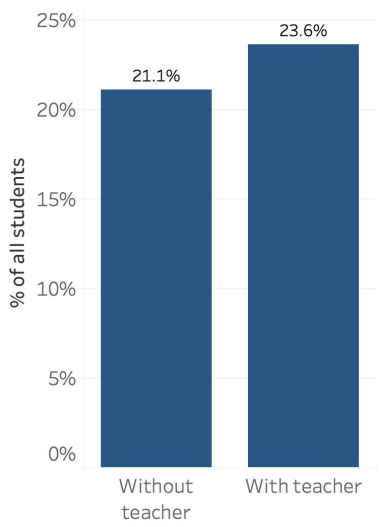
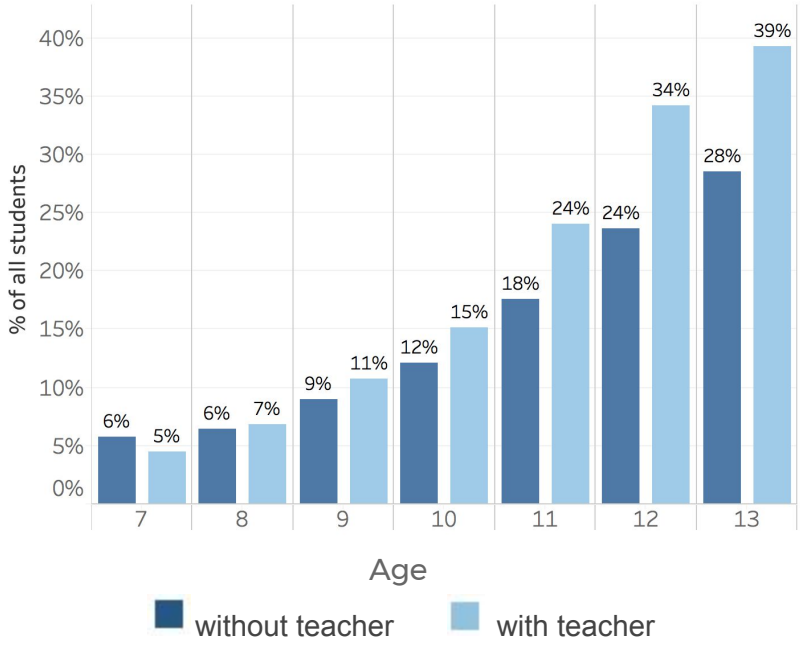


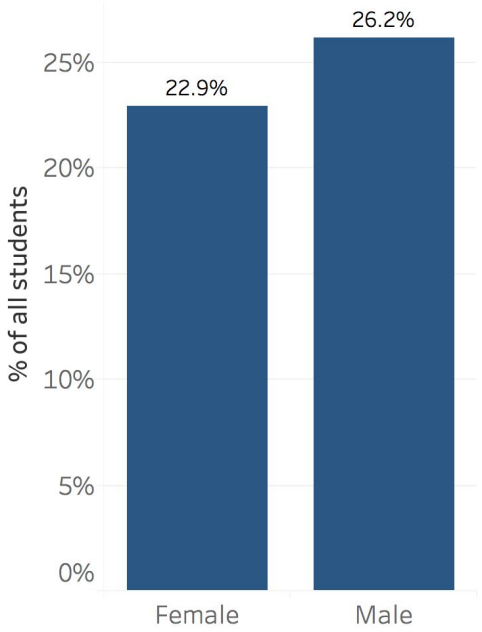
Fig. 6. - Students demonstrating basic proficiency with or without a teacher as % of all students in their age group



Does gender make a difference?

Males are more likely to demonstrate proficiency than female students. Trying to slice the data other ways - by age, by course, with or without a teacher, etc., - the gender difference remained not only present, but roughly the same proportionally.

Fig. 7 - Percentage of students (all ages) demonstrating proficiency by gender



We're curious why we see this gender disparity - is it something inherent in our course materials, or other factors? We cannot make any broad generalizations about course implementation details that may affect this. For example, anecdotally, we know that some teachers use our materials as instruction for all students in the classroom, while others leave it as an optional or "special" activity. There are plenty of other external factors that influence classrooms or might drive students to our site to learn on their own that we do not account for in these measures. What can we do to improve this? We'll continue to look into it.

Weaknesses and challenges with this metric

While this metric improves our previous attempts to measure student coding activity, it is far from perfect. The top challenges we've identified include external validation, age appropriateness and the subjectivity inherent in the tagging process, especially as it related to "difficulty".

Furthermore, since the puzzles were not designed to give equal or even robust coverage to various concepts across levels of difficulty, there is very little opportunity for the student to demonstrate proficiency for certain concepts.

External Validation

Ideally this measure would be validated against external tests of students proficiency with basic computer science concepts. As validated assessments are created for elementary school computer science, we will use them to test and refine the measure. As more work on developing learning trajectories for students of this age group is done by other researchers and institutions, we'll look to align our work with theirs.

Age appropriateness

In this work we've defined concepts and difficulties independent of age. Ideally, we would move to a framework where our definition of proficiency varies by age. Again this work will align with work on learning trajectories. Unlike subjects such as math, teaching computer science is challenging because most of our students are entering as beginners at every age. This means that 4th graders and 2nd graders are both starting with the basic concepts of loops and sequencing. We would like to refine the metrics so that we can measure what students should learn at each grade level and match the age appropriate learning progressions based on the K12 CS framework.

Subjectivity in tagging and refining the concept difficulty matrix

While we tried to define strict guidelines for a given concept and difficulty, applying these in a uniform way across 500+ puzzles proved challenging. For example, we determined a puzzle to be more or less "difficult" based on the kind of prompt presented to the student. In theory what a student was asked to do could run the gamut from ambiguous - "solve this problem" - to explicit instruction - "drag out a repeat block and set the number to 5". Since the blocks made available are also a form of prompting or hinting about how to solve the puzzle, it was difficult to judge how this should affect the difficulty rating of a puzzle.

For the work outlined in this report, we had two different people tag the puzzles to cross reference and double-check tags, and then that tagging was audited by more people. But, even with this, there is a certain amount of inherent subjectivity. We'd like to refine the metric to make it more objective.

In addition to reducing inconsistencies in tagging there are a few other changes we would like to make. As we tagged every puzzle, we struggled to account for the context in which the puzzle was presented, namely, taking into account the puzzles that came before it. We realized there were puzzles that strictly adhered to the concept-difficulty specifications we set, but in practice turned out to be easier or harder than expected. For example, a puzzle might have a limited toolbox that makes it easier for the student to find the right block to use to solve it.

We also struggled to classify puzzles that touched on multiple concepts. Or, a single concept that could be done at many different levels of difficulty. For example, a simple nested loop puzzle where the instructions prompt you how to begin might be easier or harder than a single loop where you have to do it all on your own.

To clarify these kinds of issues, we are drafting a new version of the matrix that will attempt to make the “difficulty” of the puzzle emerge from the data rather than be a subjective assignation. We have some initial ideas about how to do this but would love to hear other ideas too!

Predictivity
An ideal metric of student proficiency would be predictive of future student performance. At this point, we can’t yet say how predictive this measure will be.

In our current toolset in Code Studio, every student gets the same puzzles in the same order. Thus, we can not easily test how students would perform on harder puzzles if they saw them earlier in the sequence. There is no indication of puzzles that are too easy. We also can’t easily measure how completing level-three difficulty puzzles might vary based on the problem context - is reasoning about a loop in a “zombie” puzzle different from a similar loop in “angry birds”? Nonetheless, we are beginning to look at what we can understand around predictivity in our current environment.

One area in particular we’d like to explore is how many puzzles a student should solve to demonstrate proficiency. We are currently using 3 to make sure we don’t give a student credit for anomalous successes (perhaps with help from a neighbor or teacher). At the same time, 3 puzzles may be too few. We’ll look at whether changing the metric to require 4, 8, or 10+ puzzles makes it more predictive, but need to investigate further to find the right balance based on the number of puzzles that actually exist for certain concept-difficulty pairings. However, we don’t want to make the number so large that it effectively makes it impossible to achieve “proficiency” in cases where there are only a handful of puzzles available for a particular concept.

We’ll share any findings as we continue to study predictivity.

Next steps and future work

We plan to continue to measure and refine our metrics to assess student proficiency, but we also recognize we need to do more rigorous experimental designs, statistical analysis and modeling to measure student learning.

Some of the things we are planning to do in the near future related to this K-5 “coding proficiency” work:

- Remake the concept-difficulty matrix (and associated puzzle tags) to make it more objective, reliable and auditable
- Do classroom observations and studies to validate and calibrate proficiency measures from data
- Apply more rigorous statistical models and techniques (such as ELO modeling) for analysis

- Do a public data release

We have some ideas about how to improve our work but we also invite feedback and others' ideas about what we could be doing. If you have suggestions or thoughts on how we can continue or improve this research or if you'd like to get involved, we'd welcome your input and feedback. Please contact us at data@code.org we'd welcome your feedback.