
An Evaluation Tool For Computer Science Education Resources

Senior Capstone Project

Robert Cobb
University of Maryland – Individual Studies Program
Entrepreneurship and Innovation in STEM Education
Faculty Mentor Dr. Robert Grimm
December 2014

Table of Contents

Introduction	1
Trends in Computer Science Education.....	2
Who is learning to code?	2
Where are they learning?	2
Backgrounds and Demographics.....	4
Goals of Computer Science Education	6
Educators	6
A Computer Science Mindset: Computational Thinking.....	6
Computational Thinking Example: What If?.....	7
Computer Science Concepts	8
Programming and Learning Goals.....	9
Practical Computer Skills	9
Visualizing The Balancing Act.....	10
Student Goals	11
Computer Science Pedagogy.....	13
Constructivist CS Learning Tools	13
Relevant and Authentic Problems.....	14
Creative Community.....	14
Relating Pedagogy to Goals.....	15
Immediacy and Engagement	16
Build Delightful Experiences	16
Categorizing and Evaluating Learn to Code Resources.....	17
Current Attempts to Gather, Categorize, and Evaluate Resources.....	17
Categorizing Online Code Learning Resources	18
Evaluating Online Learn To Code Resources	19
Resource Analysis	20
Gaps and Opportunities.....	21
Common Failings of Resources	21
Under Resourced Subjects and Concepts.....	22

Addressing Gaps and Failures	22
Conclusions	24
Works Cited	25
Appendix 1A: Resource Evaluation Tool	1
Appendix 1B: Resource Evaluation Results	3
Appendix 2A: Sample Use of Tool - Analysis of Resources	4
StackExchange/StackOverflow	4
Codecademy	4
Mike Hartl's Rails Tutorial	6
Coursera - Learn to Program: The Fundamentals	6
Project Euler	7
Appendix 2B: Side-by-Side Resource Comparison	9

Introduction

STEM Education is a hot button issue. Science, math, and technology education are vital to our future, and critical for students' individual development as informed, capable citizens. Computer programming education in particular has received a great deal of attention. The 2014 Hour Of Code initiative saw the first line of code written by a sitting President, along with more than 80 million students around the world.

Computer Science is young compared to other STEM fields, and its newness is reflected in the sparse research into effective curricula and pedagogy. Online resources supplement the deficit of computer science teachers and pedagogical support.

The number and diversity of available learning resources has skyrocketed, creating a new problem: finding the best resources to use. The creators of the resources also need ways to evaluate the soundness and effectiveness of their approach.

This Capstone Project explores the trends, goals, and pedagogy of computer science education in order to construct a tool for evaluating and improving computer science learning resources. The tool itself is presented, along with examples of its use, in Appendices 1 and 2. Appendix 3 is a collection of online resources. The collection, categorization, and analysis of resources illuminate gaps and opportunities for improvement and the creation of new resources. Those opportunities are explored, and Appendix 4 contains two tutorials created to address some of the identified gaps.

Trends in Computer Science Education

Who is learning to code?

Code powers much of the world, but a small fraction of the population writes it all. International Data Corporation found that there were 11 million people worldwide who write code professionally, and another 7.5 million hobbyists, including students, who write code but do not do so for a living [1]. Sources vary widely about the breakdown of what kinds of development different programmers do, but the estimate of 18 million seems consistent across sources [1,2,3].

The US Bureau of Labor Statistics estimates that there are more than a million professional software developers, and another 340,000 computer programmers in the US. A number of related fields and jobs listed by BLS also involve coding, such as web developers, system and network administrators, and computer network architects [4]. In fact, many other technical professions may require some coding proficiency; engineering, economics, actuarial science, and statistical research are driven by heavy use of math, statistics, and modeling software, which often involve programming. Even in fields like journalism, policy, and marketing, coding skills are increasingly relevant. The total number of professional and hobbyist programmers in the US is estimated to be 3.4 million [1].

All of these professionals and all students who hope to be employed in similar positions in the future are likely to use online learning resources to further their understanding of programming and computer science. It is a normal part of a developer's daily workflow to look up answers and solutions to problems they face. Success in programming in the workplace depends on the ability to forage for information, formulating good search queries and navigating online references like the language documentation and the forum Stack Overflow [5].

This year's Hour of Code [6] initiative aims to introduce 100 million people to at least an hour of coding education. Coding is no longer the domain of professional hackers; it is becoming a normal part of everyone's education. Not everyone plans on joining a startup or becoming a programmer full-time, but more and more individuals have small goals like building their own website, designing a game, or manipulating data more effectively [8].

Where are they learning?

Offline

Opportunities to learn computer science are growing at every level of the education system. In K-12, the number of students taking the AP Computer Science exam has leveled off at just over 20,000 [3]. Code learning in schools is not limited to the AP; curricula like the recently developed Exploring CS [10] are bringing age appropriate computing education to more and more students. The number of students learning computer science in K-12 is constrained by the number of qualified computer science teachers [9]. While efforts like the CS10k initiative [11] aim to train computer science teachers to meet the demand, it will take time to change the reality: only 1 in 10 schools offers programming classes.

Fortunately, after-school clubs, camps, and programs exist to supplement classroom CS education. Annually, more than 400,000 students participate in FIRST Robotics programs, where they use code and engineering to solve problems and compete in challenges [12]. Smaller programs and camps support hundreds of students — TIC Summer camp, for instance, instructs about 2000 students per year [13]. The total number of students engaged in any kind of extra-curricular computer science is difficult to estimate — many programs overlap, and while there are studies of overall participation in extra-curricular activities, computer science and programming related clubs do not rate high enough to have their own count or category [14,15]. My best guess is that of the 55 million students in K-12 education in the US [16], more than 500,000 but less than 5 million are engaged in regular, structured computer science activities in school, clubs, or camps. More may be learning some coding online themselves.

In higher education, surveys of academic majors make it easier to count the number of students studying computer science. The totals change slightly depending on which majors count as computing degrees. With the most liberal categories, there are only 40,000 computing bachelors degrees awarded each year. Another 40,000 associates degrees, 20,000 masters and 2000 PhDs are awarded, for a total of about 100,000 computing degrees each year. BLS predicts 140,000 annual openings for programmers and computer scientists— an astonishing 40% more than the number of degrees produced. Filling those positions is a huge opportunity [4,17,18].

New institutions are taking advantage of that gap. Programming bootcamps, nonexistent 10 years ago, have sprung up to teach thousands of students every year how to program apps and websites. Usually offering 12 to 16 week intensive courses, the 50 or so bootcamps teach programming and web development to new coders with the promise of finding employment after graduation [19,20]. At the top bootcamps, that promise is fulfilled nearly every time [20] — companies really are in need of talented programmers, and the bootcamps are a reliable source of new hires.

Hackathons and Meetups deserve mention as the other organized, in-person code learning settings. Hackathons are dominated by college computer science majors, but are not limited to them. With hundreds of events worldwide and the participation of hundreds of students at each event, hackathons encourage students to learn to code via 24- and 36-hour projects. Students participate for free, with food provided, and often have their travel expenses reimbursed. Sponsors interested in hiring talented coders typically pay for the event. Meetups, on the other hand, are internet-facilitated events bringing together people with similar interests in the same city. New programmers are encouraged to meet those with more expertise. [21, 22]

Online

Programmers new and old turn to the Internet above any other source for information about coding. Online content comes in a variety of forms, drawing different audiences and teaching coding with different levels of success.

Massively Open Online Courses, or MOOCs, move the classroom experience online, for free. Millions worldwide subscribe to thousands of courses, many focusing on programming. Courses offer video lectures, exercises, and online exams to measure

proficiency with the concepts. While widely criticized for their high dropout rates [23,24], MOOCs are a major part of the new learning landscape, and thousands of people have access to high quality educational materials they otherwise would not.

In a similar style of MOOCs, video tutorials on sites like Khan Academy, Code School, and Railscasts present concepts to students for free via online videos. Some videos are in sync with exercises and written lessons, while others exist as standalone pieces.

Like video, written tutorials are split between single-shot tutorials or blog posts and multi-lesson tracks. Many developers write about how they implement features, even going so far as to share their code so that others can take advantage of the work they have done.

Programming books, online and off, combine theory with practical application and exercises. Similarly, the documentation of different languages, frameworks, and interfaces often contain example code to ease the introduction to the tool.

Codecademy is the most prominent example of in-browser interactive coding lesson platforms, but they are far from the only one. Scratch empowers millions of kids to create games and animations, and sites like CodeAvengers use games to motivate students to improve their knowledge of algorithms and syntax.

Question and answer forums like Quora and StackExchange also provide meaningful avenues for learners. Not only are many beginner questions answered already, but the stories of the community inform students about the attitudes, jargon, and experiences that make up the identity of a coder. [25]

It is difficult to count the total number of online self-studiers. Students enrolled in a Massively Open Online Course likely also use Khan Academy or StackOverflow. The 18 million experienced developers constantly use online resources as a reference and to continue learning. The 2014 Hour of Code aimed to reach 100 million new programmers for at least one hour of code practice. Suffice it to say that online materials dominate the learn-to-code landscape for both beginners and experts.

There are few or no students who do not turn to online resources for help when learning to code. The best resources are available online, but the complexity of the subject and the overwhelming number of resources available result in students struggling to navigate the thousands of resources that might not suit their needs.

Backgrounds and Demographics

As the number of people learning code expands, so do the variety and diversity of code learners. Still, white men are vastly overrepresented in the tech world [25]. Though there have been some efforts to encourage women and minorities to learn code, computer science students are less diverse than their counterparts in almost any other science or technology field [17]. While computer science learners come from all over the world, English dominates the available learning resources. Students' age, language, background, and goals all ought to shape the materials we use to teach code, especially if we aim to close these tremendous gaps.

Students' goals, motivations, and experience differ widely. Learning resources need to take that into account. Materials designed for younger children can seem condescending and

pedantic to older learners; abstract, text-heavy materials might be too dry for younger students. Many learning resources and games appear to be targeted specifically at males. While specific efforts to translate tutorials into different languages and bring coding to girls and minorities have been successful in opening resources up to different populations, we still need more targeted resources and solutions [26,27].

Goals of Computer Science Education

We want to teach computer science brilliantly. To do so, we need to know who the stakeholders are and what they want out of computer science education. Teachers and students have different notions of what good teaching looks like; employers, policy makers, resource creators, and the computer science community all have a stake in what and how students learn.

Educators and policy makers are feeling pressured by a computer science education imperative. Political pressure to improve the economy and create innovative technology spurs efforts to introduce STEM into K-12 schools and improve it in colleges and universities. At the same time, the lack of diversity in the tech community indicates the need for educational efforts to target underserved communities.

Employers want strong coders. They can be grounded in theory, but more and more often, they need a portfolio of past projects. Ability to learn technologies and problem-solving skills is highly valued, but professional developers rely on their technical proficiency with enterprise tools and languages, not the theoretical underpinnings and mathematical proofs.

In the next sections, I explore the different goals of computer science education in more depth, particularly the balance between teaching computer science theory and teaching practical skills for creating programs.

Educators

Seymour Papert's 1980 book *Mindstorms* laid the foundation for an educational philosophy of engagement with ideas through the medium of programming [30]. Since then, computer science in schools has been through several philosophical waves. Papert, working with educational psychologist Jean Piaget, believes that engaging with authentic challenges is the way to approach both theory and creation. In practice, many educators struggle to bring both components together — all too often, one is magnified and the other is forgotten. Creating excellent computer science learning experiences means figuring out how to resolve the apparently perpendicular goals.

A Computer Science Mindset: Computational Thinking

Educators recognize this dilemma. Jeannette Wing's 2006 article *Computational Thinking* [31] signifies the change in scope of K-12 and Higher Ed computer science education. Wing says that the skills and concepts that come from learning programming will be fundamental for future students. Advocates of Computational Thinking see the goal of CS learning as

- Conceptualizing, not programming
- Fundamental to understanding the world and operating in it
- Creative and human, not robotic
- A blend of mathematics and engineering, the abstract and the concrete
- About ideas, not artifacts

Computational Thinking places the emphasis not on production, but on computer science habits of mind. Not everyone needs to write code every day, but a basic familiarity with the concepts should be as widespread as our basic familiarity with math, science, and language.

The problem-solving mindsets and skills used in computer science apply across fields. According to Wing's 2006 article, Computational Thinking is not just about the ubiquitous devices or the software that runs on them, but about the new ways we view the world and the problems we can now tackle [31]. Of course, Wing is neither the first nor the last to formulate a set of computer science education principles. The most recent comprehensive list of principles comes from Grover and Pea's review of the state of Computational Thinking [32]:

1. Computing is a creative human activity
2. Abstraction reduces information and detail to focus on concepts relevant to understanding and solving problems
3. Data and information facilitate the creation of knowledge
4. Algorithms are tools for developing and expressing solutions to computational problems
5. Programming is a creative process that produces computational artifacts
6. Digital devices, systems, and the networks that interconnect them enable and foster computational approaches to solving problems
7. Computing enables innovation in other fields, including science, social science, humanities, arts, medicine, engineering, and business.

Computational Thinking Example: What If?

Computational Thinking involves a range of tools. As practiced by computer scientists, using these mental tools becomes a habit. In particular, the questions that computer scientists ask and answer include:

- How is the problem defined? (Precise constraints)
- How difficult is it to solve a problem? (Complexity theory)
- What is the best way to solve a problem? (Optimization)
- What counts as a good solution? (Estimation)
- How can we transform the problem into something we know? (Decomposition)
- How can we split the problem into repeatable subproblems? (Separation of tasks)
- What are the edge cases and worst-case scenarios? (How do we prevent them, protect ourselves with redundancy, and recover from them?) [28]

While these skills and habits of mind are particularly useful in computer science, they can apply across disciplines. Computers are the means, not the ends, of computer science education.

An excellent illustration of these habits in practice is Randall Monroe's humor blog *What if?* [33]. A computer scientist by training and a web comic artist by practice, Monroe responds to absurd hypotheticals with serious research and analysis. Some of his answers are computer-related, like his treatment of the questions "When, if ever, will the bandwidth of the Internet surpass that of FedEx?" and "How many unique English tweets are possible? How long would it take for the population of the world to read them all out loud?" Most, however, take a Computational Thinking approach to addressing non-computer-oriented questions.

The post *Hair Dryer* is a perfect example. Monroe responds to the question "What would happen if a hair dryer with continuous power was turned on and put in an airtight 1x1x1 meter box?" He breaks the problem down into its core components - in this case, the key pieces of data are the surface area of the box and the wattage of the hair dryer. He estimates a good answer based on the theory of heat transfer, and spells out the difference different factors would make to the result - the box's composition and thickness, the environment around the box, the temperature at which the hair dryer burns out.

What's telling, though, is that he does not stop there. Now that he has defined the problem, transformed it into one he can deal with, and found a good-enough solution to the subproblem, he can generalize. By adjusting the inputs, he can see what happens in edge cases; in particular, he turns the output of the hair dryer way up. By considering pieces of the problem separately, he is able not only to answer questions that might not otherwise be answerable, but also extend that answer to cover a wide range of interesting related cases.

While describing impossible situations like an indestructible hair dryer in an indestructible box putting out 11 Petawatts of power does not necessarily solve useful problems, it is fascinating, as evidenced by the number one bestselling book based on the blog. That kind of thinking can also help everyone better define and solve the real problems they face, and think creatively about the cases, which will push the limits of their solution.

Computer Science Concepts

Computational Thinking involves a particular approach to problems and a particular understanding of the relationship between people and computers. While that mindset is a fantastic goal, it leaves open the specific concepts of computer science education. These concepts support Computational Thinking with concrete metaphors and language for approaching problems and communicating ideas about solutions. For instance, having written code that takes advantage of abstraction, students will have a much better intuitive grasp of how it works and why it is important.

Grover and Pea's State of the Field [32] also listed generally accepted Computer Science concepts:

- Abstractions and pattern generalizations (including models and simulations)
- Systematic processing of information
- Symbol systems and representations
- Algorithmic notions of flow of control

- Structured problem decomposition (modularizing)
- Iterative, recursive, and parallel thinking
- Conditional logic
- Efficiency and performance constraints
- Debugging and systematic error detection

While it may be possible to teach some of these concepts in the abstract, computer science almost always involves programming. Whether dragging logical blocks in Scratch or App Inventor or writing obscure structures in C, seeing and writing programs is fundamental to the computer science learning experience.

Programming and Learning Goals

Despite all the talk about Computational Thinking, students want to be able to make things that work, and employers want to hire employees with practical skills. While deep knowledge of theory is helpful for writing great software, theory is not the only valuable part of computer science education — skills matter. Educators realize that students are motivated to work on projects, and that part of their role is to equip students with coding skills.

Still, it is not obvious what kinds of projects students ought to work on. Projects for class aim to provide practical application of the theoretical concepts instructed. Educators tend to develop challenging puzzles for students to test their skills against and assign recreations of common systems. While these projects force students to use the concepts covered in class, recreating common features is rarely, if ever, best practice for professional developers.

Particularly online, learning goals center on explicit creative abilities. Course modules are often named things like ‘make an interactive website’ or ‘game design: helicopter.’ While they mention theory and principles, the actions they require of learners centers on syntax and problem solving. The projects are intended to attract and motivate students, but the exercises often do not leave students with the ability to repeat what they learned on their own, independent of the resource. In trying to toe the line between theory and practice, tutorials often fall short along both dimensions.

Organic computer science learning often follows the “use-modify-create” path. First, students use the tools and see how they work. Then, they modify program behavior. Finally, they create their own programs from scratch. [32] This behavior is seen in tools like MIT’s Scratch and App Inventor. While the unstructured model can be highly engaging to students, it can also be intimidating to work without scaffolding or a roadmap. Reflection and theory are nowhere in “use-modify-create.” Guidance from a mentor or peers can reduce the intimidation and help tie in theory to context [37].

Practical Computer Skills

The practical skills that a programmer needs are not tied to any particular technology or language, but expertise in some language’s nuances and syntax is expected. More important however, are skills like planning, design, testing, and debugging code. Students need to be

able to break down a problem into component parts, choose appropriate representations and algorithms for a solution, and implement the solution [34].

The skills and knowledge needed to create useful, aesthetic projects are different from the Computational Thinking skills used to approach problems. Often, the difficulty in creating a successful project is not in solving technically difficult challenges, but in designing many small details of the user experience. Students do not necessarily know the distinction, and instructors whose expertise is in solving the technical challenges might not be compelled to spend time on design. It can be discouraging for students to build technically sound but unaesthetic or ill-designed projects. Student-made websites are particularly prone to this problem. Despite a sound grasp of the syntax of CSS and HTML, students may lack the design knowledge to create beautiful websites — their work often looks gaudy or tacky.

Working in industry, programmers need familiarity with the development workflow and tools facilitating collaboration. Provisioning tools, documentation, and software development frameworks like Agile and Scrum are not necessary for class projects, but the missing knowledge means a steep learning curve for programmers when they are hired. [35]

Visualizing The Balancing Act

Learning taxonomies like the revised Bloom's taxonomy (remember, understand, apply, analyze, evaluate, create) help researchers and educators differentiate learning tasks and build curricula [28]. Giving names to levels of knowledge provides a common vocabulary for discussing student level of understanding and lesson outcomes. However, computer scientists have a different implicit understanding of how knowledge of computer science develops. This raises the fundamental question - does learning computer science mean learning concepts to remember, apply, and communicate them? For many, the answer is no; application is king, and creating programs is the chief goal [29].

Fuller et al. developed a two-dimensional matrix specific for CS, combining the traditional Bloom's track with the intuitive sense that after learning Computer Science, a student ought to be able to design and create high quality artifacts.

PRODUCING	Create				
	Apply				
	none				
		Remember	Understand	Analyse	Evaluate
		INTERPRETING			

A two-dimensional taxonomy of learning for computer science. From left to right are the stages of theoretical knowledge. From bottom to top are the stages of practical skills. [29]

The goals can be understood as two concurrent tracks: theory and mindsets going one way, and practical programming competency perpendicular to it. Paths of learning flow from the bottom left towards the top and right. As different activities fall in different places on the matrix, different students might follow different paths. A student might create many programs without a deep understanding, or might demonstrate analytical knowledge of the theory without a corresponding ability to produce.

Good curricula emphasize both production and interpretation, pushing students to think deeply about the underlying theory while empowering them to produce apps, games, and websites. Excellent learning environments are designed so that the two complement each other — practical examples showcasing the need for rigorous theory, and theory supporting the engagement with authentic challenges. [36]

Student Goals

Students have different goals and motivations. Some want to pursue a job in a promising industry. Others want the ability to create cool websites, apps, and games. Some want to be able to communicate with programmers professionally. Others are curious about the technology fundamental to so much change in the world.

Significantly, most learners do not start out knowing what their goals are. Without a good picture of what there is to learn, students decide what resources to use and what to learn. They start without knowing the jargon, without understanding the landscape, and without a big picture understanding of how the different parts come together to make a computer work. Without knowing what a computer language is, they have to decide what language to learn.

Based on their goals, students want different outcomes from their learning, and expect different forms of progress towards those outcomes. Students who want to make games do not want to learn theory before they can get started; they may be more interested when they recognize the relevance of theory to their project, like making their game run faster. Those curious about electronics might not care about the minutia of HTML syntax, but would be fascinated by a video on how hardware turns electronic signals into meaningful data.

Clearly, the best learning occurs when student goals are aligned with the resources they use. Then, it is critical to know the implicit goals learners have when they indicate that they want to ‘learn to code.’ Possible student motivations:

- Creation of apps, site, game, or other program
- Employment as a programmer
- Curiosity; desire to understand
- Analysis of data
- Social - desire to be ‘cool’ and participate in tech culture
- Desire for fluency to communicate with programmers in professional context

This list is limited; students may have other goals, overlapping goals, or lack clarity about their motivation. Especially as code becomes part of the K-12 classroom, students may have been told to learn by a parent or teacher. A different question might ask what activities students feel represent satisfactory progress. Possible answers include:

- Successful creation of programs
- Readings, videos, or tutorials
- Exercises and puzzles
- Badges, levels, grades, or achievements
- Public recognition
- Participation in discussion

Resources aimed at students are usually not ‘smart’ enough to tailor themselves to student needs. Instead, they rely on students to decide whether or not the resource meets their needs. Usually, this results in students finding resources by trial and error. Without adequate understanding of their goals, students may be frustrated that their time searching and studying does not turn into satisfying results. Hours spent on forums and introductory materials feel wasted when there is no visible progress towards a specific goal.

Exceptional code learning resources will communicate their purpose clearly and ensure that students have the context to decide whether to use the resource. They will balance practical and theoretical content, and scaffold students towards sound theoretical understanding and independence in applying their skills to creation. While they may support computational thinking in the big picture, on the individual level, they fit with student motivations, helping to guide and reflect as students explore.

Computer Science Pedagogy

Computer science teaching lacks the rigorous pedagogical theory and support of traditional K-12 science disciplines. Professional development, curriculum, tools, and research into the most effective instructional methods are not as developed or widespread as those for teaching biology, chemistry, or physics - mostly because the subject is newer and less widely taught [38]. Our own University of Maryland's College of Education has tracks for middle and high school science teachers, but no program for potential computer science teachers (a track has been proposed, but it is not available yet [39]).

Still, as with the modification of Bloom's taxonomy for CS, there are attempts to apply general theories of learning to computer science pedagogy. Researchers have studied the effectiveness of current CS classroom pedagogy and proposed strategies for making CS teaching more effective. Good educational resources represent and support effective, theoretically sound instructional strategies.

Constructivist CS Learning Tools

Excellent classroom instruction ought to fit with established theories of learning. The constructivist view of learning advocated by Piaget and Papert holds that learners construct knowledge through engagement in authentic tasks embedded in a social context. Stephen Gance's 2003 paper challenges whether computers naturally facilitate this kind of learning [40]. He outlined the need for:

- An engaged learner
- Hands-on interaction with the materials of the task
- An authentic problem-solving context
- Human interactions during the process

As he saw it, much of computer-aided education facilitated learner engagement and hands-on interaction with the materials of the task, but limited the authentic problem-solving context and human interaction.

Gance, writing before Facebook and the dominance of socially connected technology, stressed the idea that technology-facilitated learning is not inherently constructivist. Better technology does not necessarily translate into constructivist instruction. Still, he recognized that electronic resources have the potential to facilitate engagement, interaction, authenticity of problems, and even human interaction, if those were specifically accounted for in the design of the learning environment.

While teachers and parents design the learning environments, some technologies foster a constructivist, inquiry-based approach, while others constrain it. Tools are used in different contexts - in classrooms, in out-of-school supervised activities, and in self-directed exploration. Constructivist learning looks very different in those different contexts. In particular, the social interaction in the classroom is focused on peers, where online it is with other users, often asynchronously.

Learners need to ask questions and interact with each other, and good environments will foster this social construction of knowledge. The enormously popular StackExchange forum is built almost exclusively around this principle. Other popular sites like Github and MIT's Scratch allow users to share projects and see how peers' programs work. The widespread use and recognition of these resources as learning tools fits with the theory that learners thrive in communities where they share what they learn.

Relevant and Authentic Problems

Android App Inventor is an MIT drag-and-drop app building environment. The creators share a vision for computational thinking integrated into K-12 education. Motivated by need for tool to teach young students about the logic of programming without getting bogged down in the syntax, and driven by the popularity of mobile applications, App Inventor has enjoyed enormous success in introducing younger students to coding [41].

Morelli et al. (2010) found core qualities of App Inventor contributing to its success [42]:

- Low floor, high ceiling (The tool is accessible and easy for beginners to use, but powerful enough for more advanced students to create complex applications)
- Problem-focused (Instead of focusing on concepts and syntax, the platform is designed for students to build the logic to solve problems they care about.)
- Genuine concepts (e.g. while reducing the amount of syntax that students need to get started, App Inventor does not dilute Object Oriented Programming logic.)
- 'Real' and motivating (Apps built on the platform can be downloaded to real phones. Students are motivated because they can actually use their own programs.)
- Support community & infrastructure (The platform promotes sharing of code and ideas, and facilitates shared learning.)

Creative Community

In *Making University Education more like Middle School Computer Club: Facilitating the Flow of Inspiration*, Repenning et. al. point out that programming courses in higher education stifle collaboration and creativity [37]. The finding that students in introductory CS courses learn very little of what is expected of them [34] suggests that the lack of collaboration and creativity may lead to poor academic understanding of concepts.

Repenning et. al. argue that peer learning is a powerful means of developing understanding. In college CS classes, looking at each other's code is frowned upon or outright disallowed. Instead, the "Sage on the Stage" is the source of knowledge, and students are meant to function only as information receivers. Piaget and Papert indicate that learners need to be active in their environment and construct their understanding [30]. Group projects usually do not serve this purpose — group members end up splitting the development into so many smaller individual projects like the ones they are familiar with [37]. Organic conversation and sharing requires a different approach.

The informal, voluntary, exciting collaboration seen in middle school computer science clubs offers an alternative model for building a learning environment. Repenning's team's years of studying these clubs resulted in principles for creating an inspirational community:

1. Display projects in a public forum
2. View and run fellow students' projects
3. Provide feedback on fellow students' projects
4. Download and view code for any project
5. Provide motivation for students to view, download, and give feedback on fellow classmates' projects

The “Sage on the Stage” analogy extends to many online resources. The video lecture style of most online courses and the textbook style tutorials cover the content but do not provide a forum for students to share knowledge and teach each other. The most successful platforms embody these principles, allowing students to explore and learn from each other with enthusiasm and a shared level of experience and understanding.

Relating Pedagogy to Goals

In *Atlas: Practical Computational Literacy For Designers* [43], John Gruen finds several choke points for new code learners:

- Lack of orientation - no topography for what they are learning
- *A priori* fear of failure
- Lack of immediacy and feedback

Gruen describes the sense that many new computer science students have of drowning in a sea of information. Students are overwhelmed by the abundance of information, but have no sense of how to navigate it. They are intimidated by the abstract concepts they are presented with, and many do not believe that they are capable of success in the field.

For the designers he interviewed, the goal of learning to code was not to produce programs; instead, they wanted computational literacy so that they could communicate their needs more clearly to developers. In general, he found that learners

- Lack clarity and technical vocabulary to express needs
- Have a vague sense they need to ‘learn to code’
- Have ill-defined assumptions about what success looks like

Gruen illustrates the need to provide the context for conceptual learning and to clearly and explicitly link theory and application to student goals.

Instructors and interactive resources should first give students an orientation to the computing landscape, and then ask students **why** they want to learn to code. What in their lives motivates them? What problems do they want to solve? With answers to those

questions, instruction and curriculum can be matched to learner goals, and students can be empowered to find the appropriate resources.

Immediacy and Engagement

When new CS students first try out code, particularly when starting with programs longer than one line, the feedback cycle is long. Students do not know whether they are on track until they have tried to compile and run the code — steps they might not even get to. Long feedback loops and periods of frustration lower students' confidence and disengage them from the learning process. *Atlas* describes how certain online resources shorten the feedback loop, giving students an immediate sense of whether or not they understand [43].

Resources often introduce new programming concepts in the “deliver— write— execute” loop. First, the concept is introduced. Then, the student writes code implementing that concept. Next, the student runs the code to see if it works as planned. Lecture-based courses and many text tutorials deliver many concepts at once, and students write and execute concepts in large chunks - sample programs, homework, and labs. Weeks might go by between the introduction of a concept and feedback on student understanding. This results in student frustration and loss of confidence [43].

In-browser interactive code tutorials like those at Codecademy deliver concepts next to the space for writing and executing code. The exercises provide immediate feedback, so learners know right away whether they understand a concept. Platforms that present and evaluate material side-by-side like this dramatically shorten the feedback loop, which in turn engages students, builds their confidence, and keeps them coming back.

Build Delightful Experiences

In *Designing for Emotion*, Aaron Walters' hierarchy of design expresses four levels of need in designing successful products: functionality, reliability, usability, and delight [44]. Most effort in designing pedagogical resources is focused on creating resources that are functional and reliable. Good resources are also designed for usability — they feel intuitive. Rarely do curriculum designers create delightful resources [45].

The most popular computer science resources have a high standard for content — accurate, appropriately leveled, clear, easy to follow. However, tools rarely include surprise or humor. Engaging professors figure out how to tie the appropriate amount of humor and surprise into their classrooms. That technique is not evident even in the best online materials. Among them, Code School is notable for crafting fun, surprising, and delightful learning experiences. I laughed when I followed their pirate-themed R tutorial [46] — and I was more engaged because of it.

Delight is an aspirational goal for materials, and the theory and evidence supporting its effectiveness for improving educational outcomes are relatively shallow. Still, delightful aesthetics are recognized as a major driver of the popularity of many of the most successful web technologies [47]. Delightful experiences, like immediate feedback, engage the learner. Cognitive challenge and meaningful concept delivery do not foreclose humor, fun, or surprise. While the other parts of the design hierarchy must be in place, delight is a final criterion for judging resources [45].

Categorizing and Evaluating Learn to Code Resources

Online resources form a large part of computer science teaching and learning landscape. Not only are they heavily relied on in the classroom, but they are also the primary resource for professionals and self-taught programmers [48]. While some attempts have been to compare and evaluate MOOCs and learn to code resources, most simply list the available resources; some categorize by what is available, and some evaluate resources for learning particular concepts. There are few or no large-scale attempts to both categorize and evaluate online code learning resources.

Current Attempts to Gather, Categorize, and Evaluate Resources

A huge number of tools are available for learning to code. Just as the technology they teach changes, the tools themselves fall in and out of popularity rapidly. Last year's best tool may not be around next year; popular sites LearnStreet, Balloon, and Code Racer, all available earlier this year, are now or will soon be defunct. The flood of new tools and the rapid change in the environment make it difficult for students to determine the best resources to use — particularly when they are unfamiliar with the landscape and cannot articulate their goals.

Online

The most widely used guides to the coding resources are the lists and collections published on popular Internet sites. The discussion forum Reddit, for example, has entire subsections of the site dedicated to learning to code [56]. Question and answer site Quora also has several user-generated lists of resources for learning to program [57]. Popular business, technology, and lifestyle news outlets also publish lists of code learning resources [60,61,62]. Organizations like Flatiron that promote code learning also publish collections of resources for new coders [58].

There are hundreds of such collections, all attempting to address the difficulty that new code learners face: navigating the resources. Bento.io [59] is one of the best resource curation sites. Bento focuses on sending users to off-site tutorials and, once they finish, guiding them to the next appropriate resource. The site's popularity is in large part due to the frame that it provides to other content. By putting the material in a larger context, Codecademy gives learners confidence that they are making progress and helps them understand the relationships between concepts. This framing is rare, even among the most popular collections. While lists often group links in categories, little is done to communicate how concepts fit into the big picture of computing. No online collection seems to evaluate resources systematically; even Bento solely depends on the opinion of its curators for the selection of content.

In Academia

Research into online learning resources fails to provide a useful guide for those interested in evaluating learn to code resources. Online and Distance Education have been subject to intense scrutiny [7], and there are general frameworks for evaluating online learning programs and tools [49]; however, most online programming resources differ in kind from the tools used to teach other courses. Two papers attempted to systematically evaluate code-

learning tools [50,51]; however, as those papers were published in 1998 and 2003, most of the currently popular resources did not exist. One short 2013 paper provides a superficial introduction and categorization of several of the available resources, but provides little in the way of systematic evaluation, ultimately serving no better than a list on a tech news site [52]. Several recent papers address the tools available for learning particular coding concepts like APIs and Data Structures [53,54]. They do an admirable job covering those resources, but their scope is limited. The majority of resources are not subject to systematic evaluation, and the majority of code learners' resource navigation needs are unmet.

Categorizing Online Code Learning Resources

In order empower students to navigate the thousands of available resources, the resources need to be collected, sorted, and evaluated. Existing collections categorize resources according to several natural dimensions:

- Cost (free/paid)
- Type (video, text, exercises, in browser coding, online course, combination, etc.)
- Language and concepts covered (JavaScript, Ruby, Python, Rails, SQL...)
- Target Demographic (age, skill level, interests)
- Language of instruction (English, Mandarin, Spanish...)

From those lists and my own investigation into the available resources, it seems that most resources fall into one of the following types:

1. In-browser coding lessons & practice
2. Full online courses
3. Tutorial sites
4. Coding platforms
5. Online Books
6. Coding challenges
7. Short tutorials and single topic blog posts
8. Discussion and Q&A Forums
9. Documentation and open source code examples
10. Resource compilations

Appendix 3 is a list of resources for learning to code, indexed by these 10 types and marked according to the above dimensions. These resources were chosen from the thousands available as an illustrative sample of the different types, focusing on the most widely used resources. For an even broader selection of resources, sortable by these and other dimensions, see Resrc.io [55].

Evaluating Online Learn To Code Resources

As the exercise in categorization shows, there are an overwhelming number of resources available. When looking at a particular resource, it is often difficult at first glance to ascertain whether it is appropriate or helpful. It is particularly difficult to discern an underlying theory of learning from a cursory inspection.

In the discussion of the goals and pedagogy of CS education, principles of good educational resources emerged. These principles inform judgments about how well resources align with what we know about teaching and learning. These principles form the basis for the qualitative evaluation tool presented here.

From the CS goals and pedagogy, there are 4 major criteria for evaluating resources, each capturing several different aspects of quality:

Fitness for students	<ul style="list-style-type: none">■ Communicate learning goals clearly■ Low Floor and High Ceiling■ Oriented in broader landscape of computing■ Appropriate to target demographic
Constructivist approach	<ul style="list-style-type: none">■ Engage students in task■ Facilitate sharing and peer learning■ Promote exploration and independence■ Authentic and 'real' problems
Design for Engagement	<ul style="list-style-type: none">■ Immediate, useful feedback■ Short Deliver, Write, Execute loop■ Smooth difficulty curve■ Supports confidence■ Not overwhelming or overpromising■ Delightful, humorous, surprising
Computational Thinking	<ul style="list-style-type: none">■ Balance production and interpretation■ Promote generalization from concepts■ Put thinking ahead of syntax■ Deliver core CT concepts (Abstraction, algorithms, symbols and representation, decomposition, iteration, recursion, parallelism, conditional logic, efficiency, error detection)

The qualities overlap and depend on each other. Resources that are designed for engagement are more likely to support social construction; resources that communicate their goals and place themselves in the larger context of computing will promote Computational Thinking. However, there is enough distinction between the four qualities that any particular resource might excel at one but fail at another. The categorization of resources above and these four qualities of excellent resources form the basis for the categorization and evaluation tool in Appendix 1.

The tool allows for systematic, qualitative comparison of resources. It is subjective, and before it could be used at a large scale, it would need to be tested and revised to establish its validity, both as a predictive model and for inter-rater reliability. As it is, the process of using the tool brings out the apparent strengths and weaknesses of different resources. Appendix 2 demonstrates the use of the tool, and shows how it can be used to form insights about different resources.

Resource Analysis

I selected some of the most popular and well-known resources for learning to code, from a broad range of resource types. Using the evaluation tool, I was able to articulate the relative strengths and weaknesses of each resource (Appendix 2). This analysis or similar application of the evaluation tool could help a student navigate the resources available, matching resources with learning goals and supplementing the weaknesses of one resource with the strength of another. The analysis also suggests opportunities for improvement and creation where these six resources fall short.

No resource got top marks across all dimensions. The creators of many of these resources may have different motives than educators and students; they could agree with the analysis but disagree that they ought to make changes. Project Euler, for example, intentionally occupies the more challenging, less engaging space. The creators of StackOverflow will not change their site to explicitly support Computational Thinking. Still, the creators and users of the resources need a more systematic means of evaluation. This tool is an informed first attempt at creating that system.

Gaps and Opportunities

Common Failings of Resources

In the research and analysis of learn to code resources, there were concepts that few resources addressed and problems that many resources had in common. The consistently weak points in the six resources analyzed with the tool were:

- Lack of a low floor: resources too intimidating
- Concepts not linked to place in broader context of computing
- Long deliver-write-execute feedback loop
- Lack of delightful design
- Prioritization of syntax over thinking and problem solving

Not all of the resources suffered from all of these problems. Scratch and Codecademy provide a low floor and have a short feedback loop, and Project Euler clearly prioritizes problem solving over syntax. Despite the proliferation of low-barrier-to-entry resources like Scratch and Codecademy, learning to code is still intimidating. Intimidating walls of text and long feedback loops are still the norm.

In the big picture, two large, interconnected sets of problems make learning to code intimidating and turn student away from computer science. The first is the information overload mentioned several times already. The second is the more subtle but perhaps more pervasive problem of the second difficulty cliff between the easy tutorials and students' authentic problems.

Navigating the world of online learn to code resources is tricky. There are tens or hundreds of places to turn for the answer to any question, and most collections of resources provide only a brief description of what a resource entails. With few exceptions, lists do not filter or evaluate the resources they link to, and provide no overarching structure or connection between different resources. A few sites, like Bento and Flatiron's pre-work site do provide a structured, filtered walk through the best resources, but many students attempt to sort through lists and collections on their own, without a clear understanding of what they are looking for, or how different concepts connect to each other.

If a new programmer persists through the onslaught of information and finds a resource that has a sufficiently low barrier to entry, such as Codecademy, that programmer will likely be able to make it through introductory sections and get a sense of what coding is like. There are many resources that provide a shallow exposure to coding, easing students into computer science and building their confidence. However, for students of Codecademy, and other introductory sites, students can complete the program but still lack the skills and confidence to create independently.

At that critical point, when students have completed their journey through an introductory resource, they confront the reality that computer science is much harder than those tutorials. The rapid progress they made through the lessons does not translate into abilities to solve

the problems they wanted to solve. The implicit promise of speedy entry and rapid understanding is belied; no one told them they would have to spend years at this to be good. This point, combined with the information overload and the lack of a deep orientation to how computers work, can be demoralizing, especially if there is no clear path to relevant application of the skills they learned in the tutorial.

Under Resourced Subjects and Concepts

Sites like Bento provide an overview and orientation to computers and the available resources for learning to program. They lay out what is out there to learn, and help new coders learn how to navigate the resources. As important as it is to learning computer science to be able to navigate the resources available, there is a deficit of resources teaching how to forage for information. Explicit instruction about how to formulate queries, ask good forum questions, and find and read documentation would be tremendously valuable for new learners.

New coders also struggle to understand how and why their programs do not work. While tutorials on debugging exist, they are not a standard feature of introductory resources. In addition, new coders often do not understand how frameworks, libraries, and APIs can magnify the power of their code. New users might notice that they cannot do the things that other programmers must be doing, but they do not know what piece they are missing.

The piece that programmers might be missing is often design skill. While computer science and design are separated in academics and often in industry, they are both critical parts of real projects, especially the small projects that beginners usually work on.

There are a number of specific computing skills and concepts that curious students might be interested in, but do not know how to look for, with relatively weak available tutorials. These include web scraping, web hosting (and servers and the domain name system), networking and routing, system administration, and encryption and security. Interfacing with hardware is both interesting and tragically absent from the tutorial landscape — if it is not a phone, an Arduino, or a raspberry pi, it is probably hard to get started with. Databases and data manipulation also seem to lack excellent tutorials, but the root problem there might not be the resources, but the difficulty of designing tutorials that help users think in novel ways about data that is meaningful to them.

Early in the course of learning to program, I was perennially frustrated that, despite grasping all of the concepts covered in computer science classes and in online tutorials, I did not understand how the computer really worked. I had a fuzzy picture of wires and chips, but I did not know where to learn about how the software and hardware interact. I still see a gap in the available resources for learning about the full compute stack. While good explanations exist, they are not prominent in introductory materials.

Addressing Gaps and Failures

Free, widely used resources for learning to code represent a paradigm shift in the way people learn. Despite the complex, abstract, technical nature of computer science, many learners are making real strides without the help of an on-hand expert teacher. While there

are many ways that resources can be improved, none of the failures or gaps diminish the stunning democratization that these resources make possible.

What's more, there are many signs that the gaps are filled naturally. Sites like StackOverflow, Scratch, and Bento become popular because they meet a need. Excellent learn to code resources become popular.

To meet the needs exposed by the analysis, current and future resources can:

- Help students navigate and relate resources, a la Bento
- Teach meta-skills like problem solving, debugging, and search techniques
- Link skills to problems authentic to students lives
- Provide orientation and intro to landscape of computing
- Expand definition of content to include blog posts, coding challenges, and forums
- Tell stories to help students form an identity as a coder, and
- Communicate norms and practices without blocking out new students with jargon

New tutorials and resources can cover topics that are covered weakly or not at all by currently available resources, and the resources that cover those topics well can be shared more widely. The Bastard's Book of Ruby, for instance, is a data journalists attempt to teach beginning coders to comb through web data. While the book is not up to the quality standards of other resources, it focuses on practical uses of programming for journalists and citizens interested in making sense of data. This kind of practical use, while not a far stretch from the skills learned through Codecademy or Coursera, has a much greater potential to keep new coders writing useful programs for themselves.

In order to begin to meet a similar need, I wrote two tutorials directly relevant to my peers. "Make A Resume Website From Scratch" and "How to Publish a Website to the Internet" (Appendix 4) walk a new programmer through the steps needed to create a personal resume website and get it to display from a custom domain. Since they were published, they have had some success, with just more than 1200 views. The process of writing them illustrated the amount of work involved in creating resources. Even relatively small resources take considerable effort to design and create.

Future tutorials might include an introduction to the compute stack, an overview of networks and how the Internet works, a guide to navigating the available resources, and an introduction to web scraping and data manipulation.

Conclusions

Many millions of learners are using online resources to learn computer science. There are thousands of resources available, and most of them, including some of the most popular, suffer from several common failings. The theory-based and goal-oriented evaluation tool can help learners and educators make decisions about the resources they use, and help resource creators improve what they make.

There are many ways for resources to support computer science learning and Computational Thinking. While none of the resources evaluated with the tool scored the highest mark in the support of computational thinking category, four out of six achieved the second highest, despite very different approaches and theories of learning. Scratch's and Project Euler's open ended platform have a very different feel than Hartl's Rails Tutorial and the Coursera Python course, but each supports thinking like a computer scientist. This suggests that there will not be a single dominant mode or type of resource universal to all learners and all concepts.

Perhaps most significantly, resources alone will not make a great computer science education. While they form an important part of the computer-learning environment, a great tool cannot make up for inadequate structure or support for learning. Without time, interest, and access, students cannot take advantage of any resource.

Works Cited

1. IDC Study: How Many Software Developers Are Out There? InfoQ. Retrieved December 17, 2014.
2. Ranger, S. (2013, December 18). There are 18.5 million software developers in the world -- but which country has the most?
3. CS Education Statistics. Exploring Computer Science. Accessed December 17, 2014. <http://www.exploringcs.org/resources/cs-statistics>.
4. Bureau of Labor Statistics, U.S. Department of Labor, Occupational Employment Statistics, December 2014. [www.bls.gov/oes/].
5. Dorn, B., & Guzdial, M. (2010, April). Learning on the job: characterizing the programming knowledge and learning strategies of web designers. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (pp. 703-712). ACM.
6. The Hour of Code is here. (n.d.). Retrieved December 17, 2014, from <http://hourofcode.com/us>
7. Means, B., Toyama, Y., Murphy, R., Bakia, M., & Jones, K. (2009). Evaluation of Evidence-Based Practices in Online Learning: A Meta-Analysis and Review of Online Learning Studies. US Department of Education.
8. Bonnie A. Nardi. (1993). A small matter of programming: perspectives on end user computing. MIT press.
9. Why TEALS? Technology Education And Literacy in Schools. <http://www.tealsk12.org/>
10. Exploring Computer Science: Curriculum. <http://www.exploringcs.org/curriculum>
11. Astrachan, O., Cuny, J., Stephenson, C., & Wilson, C. (2011, March). The CS10K project: mobilizing the community to transform high school computing. In Proceedings of the 42nd ACM technical symposium on Computer science education (pp. 85-86). ACM.
12. FIRST At A Glance. FIRST Robotics. Accessed December 17, 2014.
13. TIC Summer Camp. 2014.
14. Eccles, J. S., & Barber, B. L. (1999). Student council, volunteering, basketball, or marching band what kind of extracurricular involvement matters?. *Journal of adolescent research*, 14(1), 10-43.
15. Bucknavage, L. B., & Worrell, F. C. (2005). A study of academically talented students' participation in extracurricular activities. *Prufrock Journal*, 16(2-3), 74-86.
16. Fast Facts: Back To School Statistics. (2014, January 1). National Center For Education Statistics. Retrieved December 17, 2014.
17. Projected Computing Jobs and CIS Degrees Earned. National Center for Women & Information Technology. http://www.ncwit.org/sites/default/files/file_type/state_districtgraphics.pdf
18. Zweben, S. (2011). Computing degree and enrollment trends. Computing Research Association.

19. Bootcamps.in: Programming Bootcamps Compared. <http://www.bootcamps.in/>
20. Toscano, N. (2013, September 6). The Ultimate Guide to Coding Bootcamps: The Exhaustive List. Skilledup.
21. Major League Hacking. <https://mlh.io/>
22. Meetup.com: Programming Meetups. <http://programming.meetup.com/>
23. Jonathan Rees. (2013, July 25). The MOOC Racket. Slate.
24. Hannah Gais. (2014, July 17). Is the Developing World MOOC'd out? Al Jazeera
25. Mercier, E. M., Barron, B., & O'Connor, K. M. (2006). Images of self and others as computer users: The role of gender and experience. *Journal of Computer Assisted Learning*, 22(5), 335-348.
26. Graham, S., & Latulipe, C. (2003, February). CS girls rock: sparking interest in computer science and debunking the stereotypes. In *ACM SIGCSE Bulletin* (Vol. 35, No. 1, pp. 322-326). ACM.
27. Papastergiou, M. (2009). Digital game-based learning in high school computer science education: Impact on educational effectiveness and student motivation. *Computers & Education*, 52(1), 1-12.
28. Anderson, L. W., Krathwohl, D. R., Airasian, P. W., Cruikshank, K. A., Mayer, R. E., Pintrich, P. R., ... & Wittrock, M. C. (2001). *A taxonomy for learning, teaching, and assessing: A revision of Bloom's taxonomy of educational objectives, abridged edition*. White Plains, NY: Longman.
29. Fuller, U., Johnson, C. G., Ahoniemi, T., Cukierman, D., Hernán-Losada, I., Jackova, J., ... & Thompson, E. (2007). Developing a computer science-specific learning taxonomy. *ACM SIGCSE Bulletin*, 39(4), 152-170.
30. Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas*. Basic Books, Inc..
31. Wing, J. M. (2006). Computational thinking. *Communications of the ACM*, 49(3), 33-35.
32. Grover, S., & Pea, R. (2013). Computational Thinking in K–12 A Review of the State of the Field. *Educational Researcher*, 42(1), 38-43.
33. Monroe, R. What if? Retrieved December 17, 2014.
34. McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B. D., ... & Wilusz, T. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin*, 33(4), 125-180.
35. Tittel, E. (2010, November 24). An Ideal Programmer | The 'Must Have' List: Essential Qualifications. Inform IT. Retrieved December 17, 2014.
36. Papert, S. (1987). Computer criticism vs. technocentric thinking. *Educational Researcher*, 22-30.
37. Repenning, A., Basawapatna, A., & Koh, K. H. (2009, May). Making university education more like middle school computer club: facilitating the flow of inspiration. In *Proceedings of the 14th Western Canadian Conference on Computing Education* (pp. 9-16). ACM.
38. Ericson, B., Armoni, M., Gal-Ezer, J., Seehorn, D., Stephenson, C., & Trees, F. (2008). Ensuring exemplary teaching in an essential discipline: Addressing the crisis in computer

science teacher certification. Final Report of the CSTA Teacher Certification Task Force. ACM.

39. Proposal for New Instructional Program: Computer Science Secondary Education Teacher Preparation Programs. (2013, January 1). University of Maryland at College Park, Maryland College of Computer, Mathematical and Natural Sciences And College of Education.
40. Gance, S. (2002). Are constructivism and computer-based learning environments incompatible. *Journal of the Association for History and Computing*, 1.
41. MIT App Inventor. <http://appinventor.mit.edu/explore/>
42. Morelli, R., de Lanerolle, T., Lake, P., Limardo, N., Tamotsu, E., & Uche, C. (2011). Can android app inventor bring computational thinking to k-12. In *Proc. 42nd ACM technical symposium on Computer science education (SIGCSE'11)*.
43. Gruen, J. (2013). *Atlas: Practical Computational Literacy for Designers*.
44. Walter, A. (2011). *Designing for emotion*. New York, N.Y.: A Book Apart/Jeffrey Zeldman.
45. Procter, A. (2012). The need for delight in online education materials.
46. Try R. Code School. <https://www.codeschool.com/>
47. Robert Safian. 10 Lessons For Design-Driven Success. *Co.Design*. 19 Sept. 2013. Web. 17 Dec. 2014.
48. Dorn, B., & Guzdial, M. (2010, April). Learning on the job: characterizing the programming knowledge and learning strategies of web designers. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 703-712). ACM.
49. Boston, W. E., Ice, P., & Gibson, A. M. (2011). A review of paradigms for evaluating the quality of online education programs. *Online Journal of Distance Learning Administration*, 14(4).
50. Deek, F. P., & McHugh, J. A. (1998). A survey and critical analysis of tools for learning programming. *Computer Science Education*, 8(2), 130-178.
51. Garner, S. (2003). Learning resources and tools to aid novices learn programming. In *Informing Science & Information Technology Education Joint Conference (INSITE)* (pp. 213-222).
52. Spina, C. (2013). Learn computer programming and Web design Choosing the best resources. *College & Research Libraries News*, 74(10), 522-525.
53. Kuhn, A., & DeLine, R. (2012, June). On designing better tools for learning APIs. In *Search-Driven Development-Users, Infrastructure, Tools and Evaluation (SUITE)*, 2012 ICSE Workshop on (pp. 27-30). IEEE.
54. Patel, S. (2014). *A Literature Review on Tools for Learning Data Structures*.
55. ReSRC. <<http://resrc.io>> Retrieved December 17, 2014.
56. R/programming. *Reddit.com*. Retrieved December 17, 2014
57. What are the best ways for a complete beginner to learn programming? *Quora*. Retrieved December 17, 2014.
58. Flatiron School Pework. Flatiron School. <http://prework.flatironschool.com/>

59. Bento: A guided tour through curated, free coding tutorials on the web. <http://bento.io>
60. Falcon, A. Top 10 Websites to Learn Coding (Interactively) Online. Hongkiat. Retrieved December 17, 2014.
61. Henry, A. (2014, February 7). The Best Resources to Learn to Code. Lifehacker. Retrieved December 17, 2014.
62. Best Learning Material. Computer Science and Engineering Association Wiki. Retrieved December 17, 2014.

Appendices

- 1) Resource Evaluation Tool
 - a) Evaluation Worksheet
 - b) Evaluation Results Document
- 2) Sample Use Of Evaluation Tool
 - a) Analysis of 6 popular resources
 - b) Side by Side comparison of resources
- 3) Collection of Resources
- 4) Tutorials
 - a) Make A Resume Website
 - b) How to Publish a Website to The Internet

Appendix 1A: Resource Evaluation Tool

Purpose:

- Find best tools for communicating concepts to learners of different levels
- Help learners navigate available resources
- Direct the improvement of current tools
- Shape the design of future tools

Instructions for use:

1. Categorize the resource according to cost, type, target audience, language of instruction, concepts covered - Fill in top part of results sheet
2. For each question, rate from 1 to 5
3. Total score from each category

Fitness for students' needs

Communicates learning goals	Strongly Disagree	1	2	3	4	5	Strongly Agree
Low Floor: easy for a beginner	Strongly Disagree	1	2	3	4	5	Strongly Agree
High Ceiling: room for an expert	Strongly Disagree	1	2	3	4	5	Strongly Agree
In context of the landscape of computing	Strongly Disagree	1	2	3	4	5	Strongly Agree
Tailored to the target audience	Strongly Disagree	1	2	3	4	5	Strongly Agree

Total: _____ Average (Total/5): _____

Pedagogy - Social Construction of Learning

Engage students in task	Strongly Disagree	1	2	3	4	5	Strongly Agree
Facilitate sharing and peer learning	Strongly Disagree	1	2	3	4	5	Strongly Agree
Promote exploration and independence	Strongly Disagree	1	2	3	4	5	Strongly Agree
Authentic, 'real' problems, connected to students' lives	Strongly Disagree	1	2	3	4	5	Strongly Agree

Total: _____ Average (Total/4): _____

Engaging design

Provides immediate, useful feedback on learning tasks	Strongly Disagree	1	2	3	4	5	Strongly Agree
Short Deliver, Write, Execute loop	Strongly Disagree	1	2	3	4	5	Strongly Agree
Smooth difficulty curve	Strongly Disagree	1	2	3	4	5	Strongly Agree
Not overwhelming	Strongly Disagree	1	2	3	4	5	Strongly Agree
Delightful, humorous, surprising	Strongly Disagree	1	2	3	4	5	Strongly Agree

Total: _____ Average (Total/5): _____

Support for Computational Thinking

Supports students to produce working programs	Strongly Disagree	1	2	3	4	5	Strongly Agree
Communicates underlying theory	Strongly Disagree	1	2	3	4	5	Strongly Agree
Promotes generalization from examples	Strongly Disagree	1	2	3	4	5	Strongly Agree
Creates Environment of thinking over syntax	Strongly Disagree	1	2	3	4	5	Strongly Agree

Total: _____ Average (Total/4): _____

Appendix 1B: Resource Evaluation Results

Name of resource:

URL:

Cost:

Language of instruction:

Concepts covered:

Target Age:

Target Skill Level:

Type (check one):

- ☐ In-browser interactive lesson
- ☐ Written tutorial
- ☐ Video tutorial
- ☐ Exercises/Challenges
- ☐ Online Course
- ☐ Documentation
- ☐ Forum
- ☐ Other_____

Description:

Average Scores (out of 5):

_____Fitness

_____Social Construction of Learning

_____Engaging Design

_____Support for Computational Thinking

Notes and Comments:

Comparable Resources:

Appendix 2A: Sample Use of Tool - Analysis of Resources

StackExchange/StackOverflow

The StackExchange network of sites, and particularly StackOverflow, are the most popular forums for asking and answering questions about coding and computer

science. It is free to use, targeted at developers with some experience, and primarily in English. The site facilitates questions and answers across many computer science subjects, from general concepts to minutia of syntax.

The strength of StackOverflow lies in the community. While the site does not engage learners with a quick feedback loop or support generalizing concepts, the discussions are all situated in an authentic problem-solving context. The forum represents peer learning at its finest — asking genuine questions, sharing experience, and allowing others to see those interactions and vote on what is useful and important. StackOverflow has a terrible difficulty curve. Without knowing what to look for or previous experience with coding, it is almost un navigable. For these reasons, it cannot function as a standalone learning resource. However, due to the quality of the peer interaction on the site, it is nearly inevitable that computer science learners will end up there, finding answers to their questions.

This analysis suggests using StackOverflow to supplement other resources whose support for peer interaction may be lacking. In fact, the site already plays that role for many learners. If the creators of the site were interested in helping new learners better, they could do more to explain how the site ought to fit in to the larger picture of learning, and direct them towards resources that address their other needs.

Fit to student needs	2.4
Social Construction of learning	4.25
Engaging Design	2.0
Support for Computational Thinking	2.25

Codecademy

Codecademy is one of the most popular interactive learn-to-code tutorial websites. The free tutorials aim to help beginning coders start their journey into

programming, with tutorials on web fundamentals HTML, CSS, and JavaScript, and introductions to languages Ruby, Python, and PHP. The tutorials involve an ordered series of concepts and programming tasks in the code editor. In order to move to the next concept, the student needs to successfully write code in response to the prompt.

Fit to student needs	3.4
Social Construction of learning	2.75
Engaging Design	4.6
Support for Computational Thinking	3.0

Codecademy aims to be an all-in-one solution for beginners. As such, it focuses on making the introduction to programming as smooth and easy as possible. The smooth difficulty curve, the rapid feedback loop, and the small, non-overwhelming size of the concepts delivered all contribute to the engaging design.

The relentless focus on making the first moments of coding easy also contribute to Codecademy's weaknesses. While the site does provide a forum area, the focus is not on peer sharing and learning. While the examples, projects, and challenges are symbolic of tasks that a new learner might want to know how to do, students can pass all the way through the course without gaining the necessary skills to build their own project independently. Similarly, while the lessons present computational thinking concepts, they do not push students to figure things out on their own.

Scratch

Scratch is MIT's free visual programming platform. Used by millions of younger code learners to make games and animations, the drag-and-drop interface lowers

the barrier to entry for students who are new to programming or who lack the dexterity to type commands. On the website, students can share their projects and see others projects and how they work.

Fit to student needs	3.8
Social Construction of learning	4.75
Engaging Design	4.2
Support for Computational Thinking	3.75

Scratch is one of the best and most popular resources for teaching young students about programming. Aimed at 8-16 year olds, it is easy for young students to get started with the platform - there are examples and a tutorial and many other students' projects to learn from. Users are encouraged to create their own projects based on each others' work and independently, a classic illustration of the use-modify-create cycle.

Some of Scratch's weaknesses may be intentional. The platform never explains what students are supposed to learn. While this may make it more difficult for students to place what they learn in the context of computing landscape, it also keeps the barrier low and promotes a culture of exploration and play rather than one of explicit teaching and learning. Consequently, there is little support for development of concepts and theory, particularly the terminology used by computer scientists. While this is not a problem for most kids, and they still learn to think in algorithms and objects, it limits the transfer of their knowledge beyond the scope of the Scratch platform. Finally, though the games and animations are real and relevant to the target audience, Scratch does not empower students to solve problems in their lives beyond creating toy programs. None of these weaknesses diminish Scratch's ability to lower barriers to programming for young students; the millions of users and programs created attest to the learning, problem solving, and creativity Scratch makes possible.

Mike Hartl's Rails Tutorial

Hartl's free online book is one of the most popular introductions to the Ruby on Rails framework. It is a step-by-step guide through the setup of Ruby and the Rails environment and the creation of

three successive projects, each more complicated than the last. The book provides detailed descriptions of each step needed in order to build a web application, and mentions many best practices and habits of good programmers.

The Rails tutorial is popular because it guides learners through the difficult process of getting started in rails, and quickly gets them to the point where they can build their own web apps. It is designed for programmers who already know that they want to learn web app development with Rails — it communicates its goals clearly and places them in the larger context of computing. The wall of text could be intimidating for beginning programmers, and the difficult curve is sharp — the expectations of the reader are high.

The Rails Tutorial is not an introductory resource; it is meant to be used after establishing some familiarity with programming. It is densely packed with information, but the theory of learning it represents is not one of exploration, experimentation, or discovery. Instead, it assumes that the way to learn is to read from the book and then try out the examples. While readers may also participate in discussions on sites like StackOverflow, the rails tutorial does not directly facilitate sharing and peer learning. Ultimately, the book fits into a particular niche in the landscape of resources, and fills it's space very well. Books like it serve similar purposes: not a smooth introduction to programming or an orientation to what is out there, or even a smooth learning curve through a section of material; instead, it is a challenging but clear guide and reference through a complex subject.

Fit to student needs	3.4
Social Construction of learning	3.0
Engaging Design	1.8
Support for Computational Thinking	3.5

Coursera - Learn to Program: The Fundamentals

Coursera provides free online courses on a wide variety of subjects. This particular course dives into the fundamentals of programming with Python. The lecture videos, exercises, and homework focus on installing and working with Python. The

course also provides a forum for discussing the course, asking questions, and sharing learning with peers. More than 60,000 students took the course as it was offered, and close to 10,000 completed it. The course materials remain open to those who want to follow along at their own pace, but they cannot earn a certificate like they could if they participated while the course was offered.

Fit to student needs	3.2
Social Construction of learning	3.0
Engaging Design	3.2
Support for Computational Thinking	2.5

Similar to the Rails Tutorial, the Coursera course offers detailed explanations of how concepts influence the creation of functional programs. It has a lower floor than the Rails tutorial; it is designed for absolute beginners. Unlike most of the resources listed here, the Coursera course has quizzes and tests that assess concept knowledge, not just functional knowledge of programming. While this focus may help learners connect ideas, it limits learning to an artificial context. The real motive for understanding python syntax is not to pass a quiz, but to be a more effective problem solver.

The course description describes the ubiquity of computing devices and how the course will not only give you a better understanding of how those devices work, but help you learn Computational Thinking. The majority of the material and assessment of the course, however, is focused on Python syntax. While learning syntax is an important step towards expressing ideas with computers, the course does not bill itself as a Python course.

Despite this flaw, the course does meet most of the criteria for effective resources. Learning is social and focused on applications the student might be interested in; the lectures communicate important concepts; students quickly know when they have errors, and have support to fix them; the difficulty curve matches what most beginning students are capable of. While the course has room to improve on all of the categories, the full online course has its place in the world of available learn-to-code resources.

Project Euler

Project Euler is a site with a list of nearly 500 highly challenging math problems.

While a few might be solved with a paper and pen, the majority can only be solved with the aid of a computer. Hundreds of thousands of coders hone their skills by tackling the challenges presented. After entering the correct solution to a problem, users gain access to the forum discussion of that problem, where other solvers post their solutions and reflect on the strategy they employed. Users can also view their statistics and compare themselves to other programmers on the site.

Project Euler is another resource that fits naturally in its place in the landscape of resources. It is designed to be challenging, and its puzzles and social, competitive nature draw many programmers. That same challenge also turns people off: only half of the users have solved more than five problems. The site is good about communicating its purpose - it does not mislead anyone into thinking that the site itself will teach them to code. Instead, it encourages open-ended problem solving, using whatever languages and skills possessed.

Authentic, challenging, interesting problems have their place in learning to code; students are not regularly faced with the massive data sets that professionals deal with all the time.

Fit to student needs	2.8
Social Construction of learning	4.25
Engaging Design	1.6
Support for Computational Thinking	3.75

Challenges may not be the best way for everyone to learn to code. Like StackOverflow, however, Project Euler shows that authentic problems and a community of learning can thrive even with an intense difficulty curve and an unengaging design.

Appendix 2B: Side-by-Side Resource Comparison

Name	Type	Language/Concepts	Target Audience
Stack Overflow	Q&A Forum	All/Agnostic	Practicing Developers
Codecademy	In-browser Interactive Tutorial	Web Development, Python, Ruby	Beginning Programmers
Scratch	Creation & Sharing Platform	Scratch visual language (drag and drop)	Younger Students
Coursera - Programming Fundamentals	Full Online Class	Python Programming	Adult First Time Programmers
Mike Hartl's Rails Tutorial	Online Book	Ruby on Rails	Adult First Time Rails Programmers
Project Euler	Code Challenges	Algorithms, Problem Solving, Complexity	Math-focused programmers

*All six are offered for free and taught in English

4-5		Scratch StackOverflow Project Euler	Codecademy Scratch	
3-4	Scratch Codecademy Rails Tutorial Coursera	Rails Tutorial Coursera	Coursera	Scratch Coursera Rails Tutorial Project Euler
2-3	StackOverflow	Codecademy	Stack Overflow	StackOverflow Codecademy
1-2	Project Euler		Project Euler Rails Tutorial	
Score	Fit to Students' Needs	Social Construction of Learning	Engaging Design	Support of Computational Thinking

Appendix 3: Collection of Learning Resources

In-Browser Interactive Code Tutorials:

Name	URL	Cost	Type	Concepts
Try Haskell	http://tryhaskell.org/	Free	Interactive	Haskell
Try Ruby	http://tryruby.org/	Free	Interactive	Ruby
Khan Academy	https://www.khanacademy.org/	Free	Interactive	Multiple
Codeschool	https://www.codeschool.com/	Free	Interactive	Multiple
GA Dash	https://dash.generalassemb.ly/	Free	Interactive	Multiple
Codecademy	http://www.codecademy.com/learn	Free	Interactive	Multiple
Code.org	code.org	Free	Interactive	Multiple
Tynker	http://www.tynker.com/	Free	Interactive	Visual
CodeSpark	http://codespark.org/	Free	Interactive	Visual
Lightbot	http://lightbot.com/	Free	Interactive	Visual
Git Immersion	http://gitimmersion.com/	Free	Interactive	Git
Try JQuery	http://try.jquery.com/	Free	Interactive	JQuery
Code Learn	http://www.codelearn.org/	Free	Interactive	Multiple
Programmr	http://www.programmr.com/	Free	Interactive	Multiple
Learn Python	http://www.learnpython.org/	Free	Interactive	Python
Repl.it	http://repl.it/	Free	Interactive	Multiple
Code Avengers	http://www.codeavengers.com/	Free	Interactive	Javascript

Full Online Classes:

Name	URL	Cost	Concepts
Udacity	https://www.udacity.com/	Free	Varies
Udemy	https://www.udemy.com/	Varies	Varies
Stanford Online	http://online.stanford.edu/courses	Free	Varies
MIT OCW	http://ocw.mit.edu/index.htm	Free	Varies
Coursera	https://www.coursera.org/	Free	Varies
EdX	https://www.edx.org/	Free	Varies
Open2study	https://www.open2study.com/	Free	Varies
Allversity	http://www.allversity.org/	Free	Varies
Open Learning	https://www.openlearning.com/courses/	Free	Varies
Blackboard OEP	https://openeducation.blackboard.com/	Free	Varies
Future Learn	https://www.futurelearn.com/	Free	Varies

Open Security	http://opensecuritytraining.info/Training.html	Free	Varies
Open SAP	https://open.sap.com/courses	Free	Varies
Miriada X	https://www.miriadax.net/	Free	Varies
Neodemia	https://www.neodemia.com/	Free	Varies
Iversity	https://iversity.org/en/courses	Free	Varies
Saylor	http://www.saylor.org/courses/	Free	Varies
NovoEd	https://novoed.com/	Free	Varies
UW Flexible Option	http://flex.wisconsin.edu/degrees-programs/information-science-technology/	Free	Varies
P2PU	https://p2pu.org/en/groups/list/community/	Free	Varies
Apna Course	https://www.apnacourse.com/	Free	Varies
Edureka	http://www.edureka.co/	Free	Varies

Creation and Sharing Platforms:

Name	URL	Cost	Type	Concepts
Hackety Hack	http://www.hackety.com/	Free	Platform	Ruby
Code Combat	http://codecombat.com/	Free	Platform	Javascript
Scratch	scratch.mit.edu	Free	Platform	Visual
MIT App Inventor	http://appinventor.mit.edu/explore/	Free	Platform	Visual Android
Game Maker	https://www.yoyogames.com/studio	Free	Platform	Games
Fight Code	http://fightcodegame.com/	Free	Platform	Javascript
Simple	http://www.simplecodeworks.com/homepage.html	Free	Platform	Simple
Stencyl	http://www.stencyl.com/	Free	Platform	Games
Open Processing	http://www.openprocessing.org/	Free	Platform	Processing
Code Bender	https://codebender.cc/	Free	Platform	Arduino
Soda Play	http://sodaplay.com/	Free	Platform	Modeling
Alice	http://www.alice.org/index.php	Free	Platform	3D Stories

Written Tutorials:

Name	URL	Cost	Type	Concept
Tuts+	http://tutsplus.com/	Free	Written	Many
Channel9	http://channel9.msdn.com/	Free	Written	Many
Inform IT	http://www.informit.com/	Free	Written	Many
Treehouse	https://teamtreehouse.com/	Paid	Written	Many
Coursolve	https://www.coursolve.org/		Written	Many
Skillshare	http://www.skillshare.com/	Free	Written	Many
Thinkful	http://www.thinkful.com/	Paid	Written	Many
HTML5Rocks	http://www.html5rocks.com/en/	Free	Written	Web

Gymnasium	http://www.thegymnasium.com/	Free	Written	Many
Lynda	http://lynda.com	Paid	Written	Many
Ruby Koans	http://rubykoans.com/	Free	Written	Ruby
JQuery Center	http://learn.jquery.com/	Free	Written	JQuery
Webmonkey	http://www.webmonkey.com/tutorials/	Free	Written	Web
TutorialZine	http://tutorialzine.com/	Free	Written	Many
MDN	https://developer.mozilla.org/	Free	Written	Many
echoecho	http://www.echoecho.com/school.htm	Free	Written	Many
W3Schools	http://www.w3schools.com/	Free	Written	Web
Entheos	http://www.entheosweb.com/	Free	Written	Many
Web Design Tutorials	http://www.webdesign.org/tutorials/page-1.html	Free	Written	Web
After Hours Programming	http://www.afterhoursprogramming.com/	Free	Written	Many
Meteor Academy	http://meteor.academy/	Free	Written	Many
Code Heaps	http://www.codeheaps.com/	Free	Written	Many
Site Point	http://www.sitepoint.com/	Free	Written	Many
Tutorial Republic	http://www.tutorialrepublic.com/	Free	Written	Many

Code Challenges:

Name	URL	Cost	Type
Google Code Jam	https://code.google.com/codejam/contests.html	Free	Challenge
Project Euler	https://projecteuler.net/	Free	Challenge
hackerrank	https://www.hackerrank.com/	Free	Challenge
coder byte	http://coderbyte.com/	Free	Challenge
code eval	https://www.codeeval.com/	Free	Challenge
code chef	http://www.codechef.com/	Free	Challenge
code challenge	https://www.mscodechallenge.net/	Free	Challenge
hackerearth	http://www.hackerearth.com/	Free	Challenge
top coder	http://www.topcoder.com/	Free	Challenge
daily programmer	http://www.reddit.com/r/dailyprogrammer	Free	Challenge
codility	https://codility.com/programmers/	Free	Challenge
sphere judge online	http://www.spoj.com/	Free	Challenge
talentbuddy	http://www.talentbuddy.co/	Free	Challenge
code wars	http://www.codewars.com/	Free	Challenge
rosalind	http://rosalind.info/problems/locations/	Free	Challenge
code condo	http://codecondo.com/coding-challenges/	Free	Challenge

Appendix 4A: Make A Resume Website From Scratch

You've seen those fancy resume websites. Now make your own.

Hey.

Your artistic and web-savvy friends make them. Your instructors might have plain-white ones. If you have no idea what I am talking about, check out [some example sites](#).

This how-to walks you through building your own resume site and getting it published online. If you want to buy a template or pay someone to make a site for you, there are places to do that. This is not that.

Building a personal website will take you some time. If you are like lightning, maybe an hour. If you spend time tinkering and perfecting, it could be a few days. We think it's worth it.

Here is a [preview](#) of what you will build. It's adapted from a theme by [blacktie.com](#) with some backgrounds from [graphicburger](#).

You can go fast or slow. If you want to go fast:

1. Download this [folder](#) (or clone the git repository <https://github.com/knommon/resume>)
2. Unzip the file and drag the folder into the sidebar of Sublime Text (⌘+K ⌘+B to open the sidebar if it is not there)
3. Find all the places where it says EDIT in **index.html** (⌘+F to search)
4. Replace the content with your information
5. Skip to [7. Theming your website](#)

Going slow will help you develop a better grasp of what is going on under the hood.

1. Start an HTML File

Open up Sublime Text. ([download here](#)). If you are a coder then your favorite text editor will work too. If you haven't coded before, welcome to Sublime! It's wonderful. If you want some tips to use it better, try [here](#).

Create a new file (⌘+N). We will put it in a folder later.

Copy and paste this code into your file: [embedded code]

Ignore all the things you don't understand. Edit the title—that's what will show up at the top of the browser.

Change ‘Your Name’ to your own name, ‘What you do’ to the thing you are (author, librarian, social worker, party animal) and add your email both places. Save the file (⌘+S) as **index.html**, in a new folder called **resume**.

Navigate to that folder and open the **index.html** file in your browser.

It should open up and look like [this](#):

Your Name Here

What you are | [You@Email.com](#)

It’s not very good yet, but you made a web site! COOL!

2. The About Section

A quick intro to who you are and what you are about.

Paste this code between the big chunk from part 1 and the closing **</body>**tag.

[embedded code]

Now edit the paragraph so the short description is actually a short description, instead of instructions. If you had an objective statement on your paper resume, this section might draw on it.

Save again, and try it out in the browser. Still ugly, but we are laying the groundwork here. (It should look something like [this](#)).

Your Name Here

What you are | [You@Email.com](#)

ABOUT

This is a short description of who you are. Keep it to about 3 sentences and make sure to highlight your unique qualities as an applicant?

[DOWNLOAD PDF](#)

See that link? You can make it easy for someone to download a pdf of your resume. All you need to do is save the pdf file in the same **resume** folder and make sure that the name of the file is the same as the one referenced by the hyperlink. Like so:

```
<a href="yourname_resume.pdf" target="_blank">
```

3. THE RESUME

The moment you've been waiting for: Adding your resume. Education, Work Experience, Awards—whatever you want to show off to a potential employer. This section has a lot of code all at once, but it's the same story as before: copy, paste, edit.

It's a lot more editing this time. Take breaks. This code goes below Step 2's code, and above the closing **</body>** tag.

Edit the template text to make the resume your own. This code assumes you want three sections: Education, Work, and Awards, in that order. If you don't want one of those sections, you can modify the content to suit your needs. If you want them in a different order, switch them around. If you want a section that isn't included (like Skills, Certifications, or Activities), you can copy and adapt one of these.

Save and check how your site looks obsessively. It should look like [this](#).

4. Contact information in the footer

Your email address has been below your name since the beginning, but you should include other contact info down below everything else (just below the closing **</body>** tag).

Here's the code: [embedded code]

Add your email, address, and phone number. Then, replace each of the #'s with the web address of your social site. You want to make it easy for people to connect with you. Each will look something like:

```
<a href="https://www.facebook.com/medium" target="_blank"><i  
class="icon-facebook"></i></a>
```

The icons will not show up yet when you save and open in the browser. That's okay, we will add them in soon. If you don't want to link to a particular social account, delete the whole line.

With the footer in, it should look like [this](#).

5. Finishing up index.html

We want to get to the styling of the page, making it cool and responsive (adjusts to different size screens like computers, phones, etc.). Just a few things to add to the index.html page before we can do that—don't worry, these are just cut and paste!

Put this below **<head>** but above **<title>** up at the top of your file:

If you want, you can fill in the optional details **description** and **author**. Make sure to add your info inside the quotes in content="".

Save it and open it up—still ugly! Should be something [like this](#).

As you can see, plain HTML is very boring; it just gets the content on the page. Next up: making the whole thing beautiful. On to the styling!

6. Adding Style

This is where we put the *design* in *web design*. If you haven't already, download the tutorial files [here](#). Copy the **css**, **font** and **images** folders into your **resume** folder. The **css** folder contains stylesheets to make the site look pretty. But if you reload the page, it will still look the same! That's because we still need to tell the browser to use the styles that we've created. Here's how:

Put the following code below the **</title>** but above the closing **</head>** tag:

[embedded code]

Now if you reload your page, it should look like a real website. Like [this](#).

Congratulations, you did it!

You built something you should be proud to showcase. Take a break, get some fresh air.

Now on to customizing your website! (Having the same site as everyone else is boring).

7. Theming your website

We'll be working to customize 3 main sections of the resume: the background behind your name, the banner and main font color, and the link color. But as you'll soon see, it's easy enough to change the color for just about anything. Pick 3–4 colors you like from <http://clrs.cc/> or from [Adobe Kuler](#) (hover over a color to show it's code). The color codes you need look like this: **#FFFFFF** or this: **#1CBCAD**.

a. Customize the header

Open up **main.css** and go to where it says **.header {** (on line 148)

If you don't want to use a background image and prefer a plain ol' color, delete the line like:


```
background: url(../images/header-bg.jpg) no-repeat center top;
```

Look at the line directly above and replace **#1CBCAD** with the color of your choosing.

Otherwise, if you'd like to change the header image, you can use a different image or download a new image to the images folder. Then change the **url** to `../images/your-image.jpg`. Save and reload and your new image should show up. If you like the background that's there but want a different color, there are three other images to try included in the images folder.

b. Customize the banner and main text color

To change the about section's background color, open **main.css** and find the following section (*on line 110*):

```
/* About Wrap */
.about {
  background: #2c3e50;
```

Change **#2c3e50** to whatever color your heart desires.

A similar process can be done to change the main text color. Look for the following section and edit it in a similar fashion:

```
.desc t {
  color: #34495e;
  font-weight: 700;
}
```

c. Change the link color

Pick an accent color for the links to make them stand out on the page. To change the link colors (and the social icon colors) find and edit the following section:

```
/* Links */

a { color: #1CBCAD;
```

Replace **#1CBCAD** with the accent color for your links.

Note: **color** changes the text color whereas **background** changes the background color / image.

Now that you've seen how to edit some colors of a few elements on the page, play around and see what else you can change.

8. Challenge: Animate your name

Difficulty: moderate. If you don't like challenges, skip to step 9.

Background

Let's talk a little bit about what's happening under the hood of our website. We've seen that HTML by itself is very bland and CSS allows us to theme our site by changing the colors and background. But how does the browser know what style to apply to each part of the page? Basically, we tell it to with HTML like this:

```
<div class="about">
```

We're telling the browser to apply the **.about** style to this container. (For no rhyme or reason “.” (dot) stands for class.)

Styles can also be applied to types of elements like we saw before with links:

```
a { color: #1CBCAD;
```

The **a** is the HTML link and this style applies to all links on the page.

The Challenge: In `main.css`, there are two classes called **animated** and **fadeInUp**—you'll need both of these. See if you can animate your name by including these classes in the class attribute of the right **div**.

Hint: you can add more than one class per element separated by a space. Example:

```
class="about class1 class2"
```

Ready, set, go!

If you're stuck, here's the [gist](#).

9. Optional: Change the fonts

So you've made it this far, but the default font choice is really not working with your vision. Let's do something about that...

Head over to [Google Fonts](#) and find 2 fonts. One for your headers (i.e. your name, ABOUT, etc.) and a second for body text (the large chunks of text that you read). Make sure to choose a readable body text font; the header font can be anything under the sun.

Scroll through, and if you see any you like, click **Add to Collection** (it's better to add more than 2 here). When you have a few that you like, **Review** them (tab at the bottom right). Narrow it down to your 2 favorites, and click **Use**.

1. Choose the styles you want:

☐ Oxygen	
<input checked="" type="checkbox"/> Light 300	Grumpy wiz
<input checked="" type="checkbox"/> Normal 400	Grumpy wiz
<input checked="" type="checkbox"/> Bold 700	Grumpy wiz
☐ Merriweather	
<input checked="" type="checkbox"/> Light 300	Grumpy wi
<input type="checkbox"/> <i>Light 300 Italic</i>	Grumpy wiz
<input type="checkbox"/> Normal 400	Grumpy wi
<input type="checkbox"/> <i>Normal 400 Italic</i>	Grumpy wiz
<input type="checkbox"/> Bold 700	Grumpy w

For your body font (Oxygen in this example), we need fonts of weight 300, 400, and 700. If the font you pick doesn't have that, don't worry about it. For the header, we only need the 300.

Scroll down to part 3, where it says “Add this code to your website” and copy that code.

Now to replace the old fonts. Open up **index.html** and find the line where the old fonts were linked. It should look like this:

```
<link href="http://fonts.googleapis.com/css?family=...>
```

It is at the top between the <head> tags. Replace this line with the code you just copied.

One last thing: we have to update our stylesheet to use the new fonts we've imported. So head over to **main.css** and go up to the top. Inside the **body** rule, change **Lato** to the name of the font you chose for the body (but keep the quotes!).

```
body {
  background-color: #f2f2f2;
  font-family: "Lato";
```

and then switch **Raleway** to your header font.

```
h1, h2, h3, h4, h5, h6 {
  font-family: "Raleway";
```

Save both of these files and check out your new fonts!

Pro tip: typically web developers work with both HTML and CSS files at the same time, so it makes it easier to have them both open at the same time.

To split screen in Sublime Text, go to View > Layout > Columns: 2 (Option+⇧+2) and then drag your main.css tab to the new right column.

All finished!

You're awesome! You've created something really cool looking, and you learned a bit in the process. Now it's time to celebrate. Print it out and put it on your fridge, show your friends, tell your mom.

Where to go next?

Do you want other people to be able to see this resume online, or even put a link on your physical resume to this website? If so, check out [*How to Publish a Website*](#).

Appendix 4B: How to Publish A Website To The Internet

You've coded a site. Now, put it on the Internet for people to see.

YO.

This was written to follow the [Make Your Own Resume Website From Scratch](#) tutorial, but the steps should be the same for any other site. We'll use that site as the example. Have all your code in a folder on your computer before starting.

The first tutorial could be completed entirely for free. While there are some options for free web hosting out there, you might end up spending a little money on this (probably less than \$20). In this tutorial, I used a [student deal on Namecheap](#) and hosted via Google Drive (ht to [Achim Scarlet](#)). There are lots of other ways to do it, and you are welcome to explore those. (In particular, [Github pages](#) are cool if you are up for using git, and there are purportedly ways to [host via dropbox](#)).

Different domain names cost different amounts. Web hosting has monthly or yearly costs. Read any agreements carefully so you don't get hooked into paying more than you want.

1. Find a Domain

Let's start with a fun part!

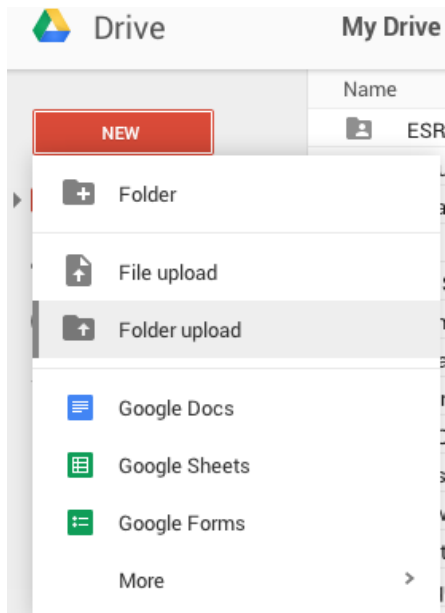
The domain is what appears at the top of the browser, like medium.com or robcobb.me. You want to find a catchy-cool, easy-to-remember domain that identifies the site as yours. At the same time, you have a limited budget, and the popular domains often cost way more. If 'Yourname.com' is available, cool. If not, it might take more creativity. Luckily, there is a great tool for instantly searching for domains: Instant Domain Search!

Spend a little time finding the perfect name, and be sure to test the prices of the different providers by selecting different options along the bottom. When you find the one you like best, buy that sucker! Remember, Namecheap has a [student special on .me addresses](#).

2. Upload your files to Google Drive

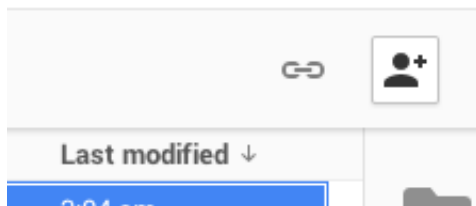
If you have not signed up for [Google Drive](#), do so.

In Drive, click New: Folder Upload

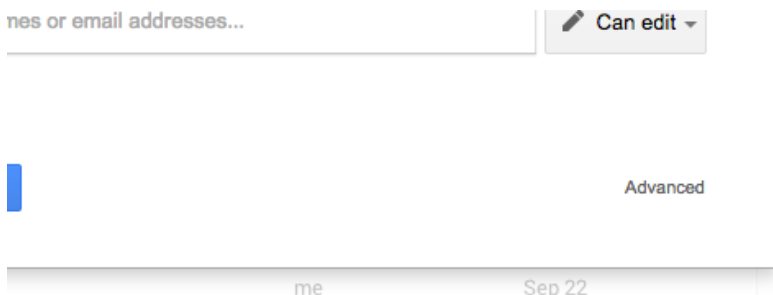


Choose the folder with your website from the menu and click upload. (If you followed the last tutorial, it was the **resume** folder.) It may take a little while to upload, depending on the size of your website.

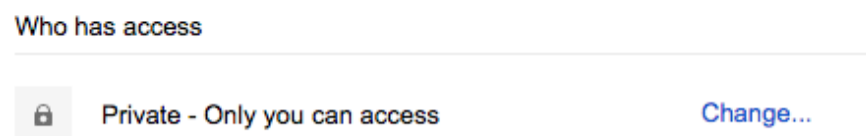
Once it is uploaded, select the folder and click sharing settings.






Click advanced.



Change the sharing settings from private to public.



Link sharing

- ☒  **On - Public on the web**
Anyone on the Internet can find and access. No sign-in required.
- ☐  **On - Anyone with the link**
Anyone who has the link can access. No sign-in required.
- ☐  **Off - Specific people**
Shared with specific people.

Access: Anyone (no sign-in required) [Can view](#) ▼

Note: Items with any visibility option can still be published to the web. [Learn more](#)

[Save](#) [Cancel](#) [Learn more about visibility](#)

And save and done.

Now for the trick: Open [this link](#) in a new window. (It will give you Google's 404 error page). Back in Google Drive, click into your new folder, and copy the part of the url after *drive/u/0/#folders/* (**this is the part you copy**)

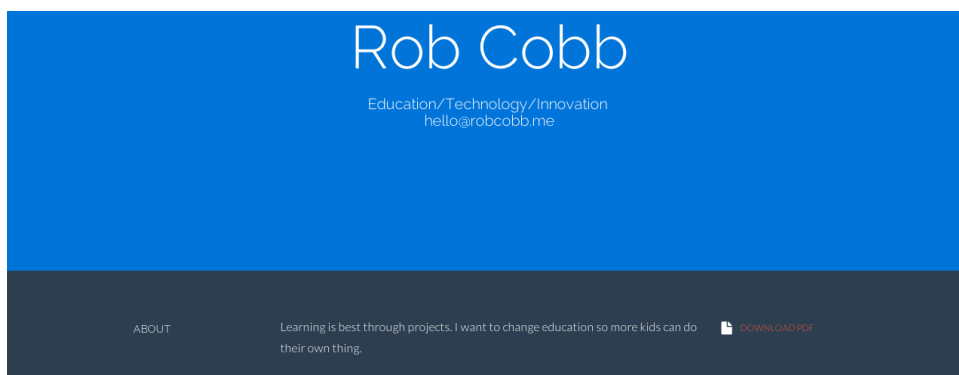
Paste the part you copied at the end of the url in the link you opened. It should look something like:

<http://www.google.com/host/0ByAlIOLoOmVWWVYxdVVGGeUJieDg>

Your browser will redirect you to a different url, with your site! Mine took me to:

<https://ddc1e269165117bd538cfba6fc3ee3eb441f947e-www.google.com/host/0ByAlIOLoOmVWWVYxdVVGGeUJieDg/>

and there's my site!



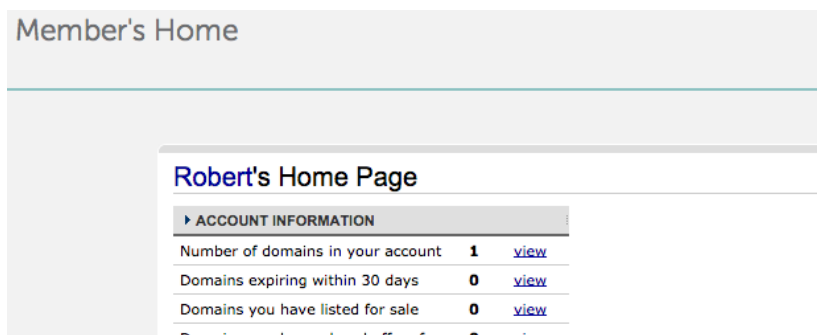
3 Connect Your Domain To Your Site

Now you have your own domain, and you have a site that is up online! All that's left is to connect the two, then you can tell everyone about it.

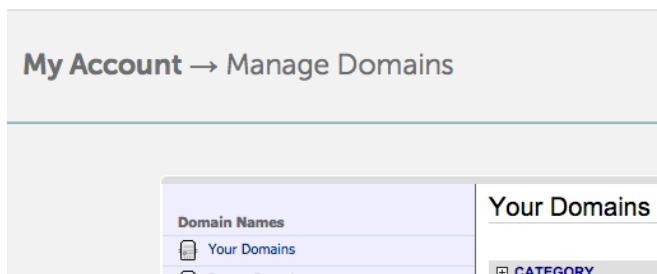
Depending on where you bought your domain, managing the domain will look differently. The goal is to create a URL Redirect to the address of the file you are hosting on Google. I did it on NameCheap, but it looks pretty similar on other domain services.

Log Into Your Account

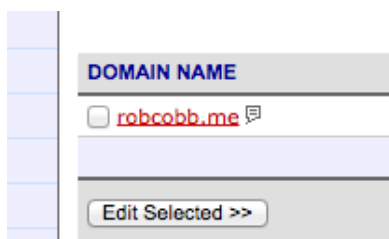
You probably just created an account in order to purchase your domain. Log on in so that you can manage that domain.



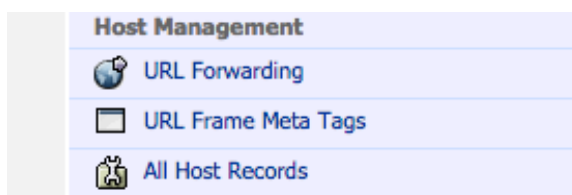
Go to Manage Domains.




Click to edit your domain



And go to URL Forwarding



Create two new URL Frame records, with host name 'www' and '@' and the url from the spot where your site is hosted on Google in the url field.

 **Modify Domain:** robobb.me

[Related Help](#) [Related Video](#)

HOST NAME	IP ADDRESS/ URL	RECORD TYPE ?	MX PREF	TTL
@	<input type="text" value="https://ddc1e26916511"/>	<input type="text" value="URL Frame"/>	n/a	<input type="text" value="1800"/>
www	<input type="text" value="https://ddc1e26916511"/>	<input type="text" value="URL Frame"/>	n/a	<input type="text" value="1800"/>

[Save Changes](#)

Save your changes, and navigate to the url you bought! Hooray! You did it!

Mine is here: robobb.me

Where to go next?

Keep making your website better! Change the pictures, add new content, change the fonts or colors. When you change the files in Google Drive, the website will update automatically. Good luck, and keep on learning!