UNIVERSITY OF CALIFORNIA SAN DIEGO

Instructor-Centered Design of Tools to Support Teaching Programming and Data Science At Scale

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy

in

Cognitive Science

by

Samuel Ethan Lau

Committee in charge:

Professor Philip J. Guo, Chair
Professor James D. Hollan
Professor Ranjit Jhala
Professor Bradley Voytek
Professor Haijun Xia

2023

The Dissertation of Samuel Ethan Lau is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2023

DEDICATION

For my wife, Tina, who fills my life with love and countless cherished memories.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would not be writing this dissertation without the grace of God and the help of my incredible community, and I'd like to express my thanks to the following people for their support through the many ups and downs of my journey.

To Philip, my advisor, thank you for always welcoming my ideas, no matter how far-fetched. Your pragmatic, get-it-done approach to research was incredibly refreshing, and I will certainly miss our "freestyle rap battle" meetings. Even before I started the PhD, you were confident in my abilities even when I wasn't, and I can't imagine reaching this milestone without your help. Thank you.

I would like to thank my parents, Harry and Echo Lau, and my sister, Sophia Lau, for their unyielding love and belief in my potential. I would also like to thank my parents-in-law, Wei and Lucy Ye, for their continual care and support from the day we met.

I'm grateful for my committee members, Jim, Ranjit, Brad, and Haijun, for many insightful questions and advice over the years. I would also like to extend my gratitude to Steven Dow and Nadir Weibel for constantly making the Design Lab a welcoming place to be.

To the graduate students who mentored me in the Design Lab, Vineet Pandey, Tricia Ngoon, Ailie Fraser, and Amy Fox, thank you for sharing with me your knowledge, expertise, and above all, your patience. To my labmates, Sean Kross and Ian Drosos, thank you for many stimulating conversations and laughs. To the other students of the Design Lab, and especially Srishti Palani, Tone Xu, Lu Sun, Hui Xin Ng, and Matin Yarmand, thank you for always humoring my ideas with enthusiasm.

I would like to acknowledge the Cognitive Science department, and especially Marta Kutas and Andrea Chiba for mentoring our second and third year research course. Their passion and commitment to cognitive science students has been truly inspiring.

Finally, to my beautiful wife, Tina Ye, thank you for loving me wholeheartedly. I thank God every day for bringing you into my life.

Chapter 3, in full, is a reprint of the material as it appears in the proceedings of the ACM Conference on Learning at Scale as The Challenges of Evolving Technical Courses at Scale: Four Case Studies of Updating Large Data Science Courses. Sam Lau, Justin Eldridge, Shannon Ellis, Aaron Fraenkel, Marina Langlois, Suraj Rampure, Janine Tiefenbruck, Philip Guo. 2022. The dissertation author was the primary investigator and author of this paper.

Chapter 4, in full, is a reprint of the material as it appears in the proceedings of the ACM Technical Symposium on Computer Science Education (SIGCSE) as How Computer Science and Statistics Instructors Approach Data Science Pedagogy Differently: Three Case Studies. Sam Lau, Deborah Nolan, Joseph Gonzalez, Philip Guo. 2022. The dissertation author was the primary investigator and author of this paper.

Chapter 5, in full, is a reprint of the material as it appears in the proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC) as The Design Space of Computational Notebooks: An Analysis of 60 Systems in Academia and Industry. Sam Lau, Ian Drosos, Julia M. Markel, Philip J. Guo. 2020. The dissertation author was the primary investigator and author of this paper.

Chapter 6, in full, is a reprint of the material as it appears in the proceedings of the ACM Conference on Human Factors in Computing Systems (CHI) as TweakIt: Supporting End-User Programmers Who Transmogrify Code. Sam Lau, Sruti Srinivasa Ragavan, Ken Milne, Titus Barik, Advait Sarkar. 2021. The dissertation author was the primary investigator and author of this paper.

Chapter 7, in full, is a reprint of the material as it appears in the proceedings of the International Workshop on Data Systems Education (DataEd) as Teaching Data Science by Visualizing Data Table Transformations: Pandas Tutor for Python, Tidy Data Tutor for R, and SQL Tutor. Sam Lau, Sean Kross, Eugene Wu, Philip J. Guo 2023. The dissertation author was a primary investigator and author of this paper.

VITA

2017	Bachelor of Science in Electrical Engineering and Computer Science, University of California, Berkeley
2018	Masters of Science in Computer Science, University of California, Berkeley
2023	Doctor of Philosophy in Cognitive Science, University of California San Diego

PUBLICATIONS

The Challenges of Evolving Technical Courses at Scale: Four Case Studies of Updating Large Data Science Courses. Sam Lau, Justin Eldridge, Shannon Ellis, Aaron Fraenkel, Marina Langlois, Suraj Rampure, Janine Tiefenbruck, Philip Guo. ACM Conference on Learning @ Scale (L@S), 2022.

How Computer Science and Statistics Instructors Approach Data Science Pedagogy Differently: Three Case Studies. Sam Lau, Deborah Nolan, Joseph Gonzalez, Philip Guo. ACM Technical Symposium on Computer Science Education (SIGCSE), 2022.

The Design Space of Computational Notebooks: An Analysis of 60 Systems in Academia and Industry. Sam Lau, Ian Drosos, Julia M. Markel, Philip J. Guo. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), 2020.

TweakIt: Supporting End-User Programmers Who Transmogrify Code. Sam Lau, Sruti Srinivasa Ragavan, Ken Milne, Titus Barik, Advait Sarkar. ACM Conference on Human Factors in Computing Systems (CHI), 2021.

Teaching Data Science by Visualizing Data Table Transformations: Pandas Tutor for Python, Tidy Data Tutor for R, and SQL Tutor. Sam Lau, Sean Kross, Eugene Wu, Philip J. Guo. International Workshop on Data Systems Education (DataEd), 2023.

ABSTRACT OF THE DISSERTATION

Instructor-Centered Design of Tools to Support Teaching Programming and Data Science At Scale

by

Samuel Ethan Lau

Doctor of Philosophy in Cognitive Science

University of California San Diego, 2023

Professor Philip J. Guo, Chair

Instructors of technical subjects like programming and data science use a wide array of software tools that enable them to create sophisticated and engaging lessons at scale. Although there are many such tools available, instructors often find themselves repurposing software originally designed for other people, like professional software engineers. This mismatch of intent adds extra logistical complexity to the already-challenging task of designing and delivering effective learning content.

To address these issues, this dissertation takes an instructor-centered approach. It surfaces

previously unmet needs through studies of instructors, their goals, and their software tools. The key findings are that instructors constantly seek to update their learning materials, yet encounter heavy logistical challenges in doing so because the tools they use to help design their lessons were not intended for instructional use.

This dissertation also contributes novel interactive systems that directly support teaching by designing for instructor needs. In particular, this dissertation contributes program visualization tools that enable instructors to show how code transforms data: TWEAKIT helps learners work with unfamiliar code snippets, and the Pandas/Tidy Data/SQL Tutors automatically visualize code that manipulates data tables step-by-step. Together, this dissertation provides the first evidence that the insights gathered from an instructor-centered approach can lead to tools that better support the work of instruction.

Chapter 1

Introduction

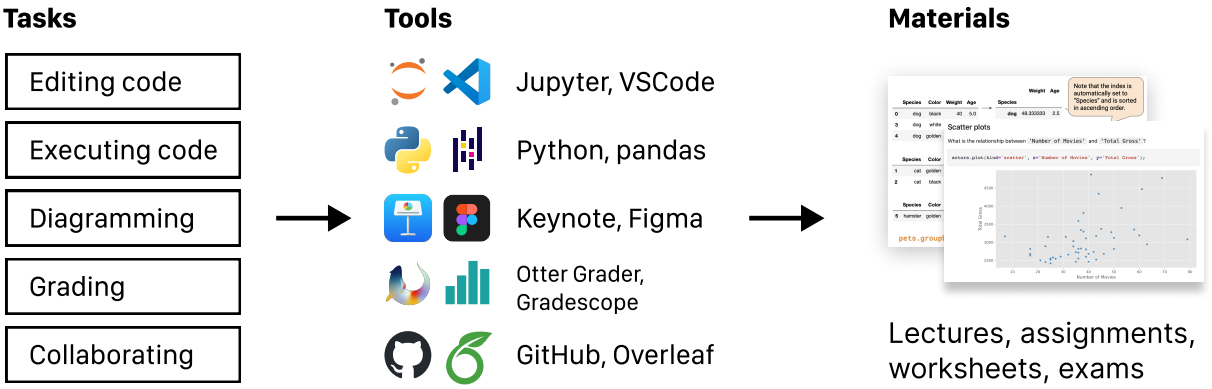


Figure 1.1. For instructors of large technical courses, even a single week of class involves many tasks (left). Each task is completed using a collection of software tools, which instructors must become familiar with (middle). By working with their tools, instructors produce high-quality learning materials that deliver lessons at scale (right).

Instructors today have access to a wide array of software tools which they use to create sophisticated and engaging lessons. Consider Rosa, a hypothetical instructor who teaches data science at a large public university. During her lectures, she flips back and forth between a slideshow presentation in PowerPoint and code in a Jupyter notebook. The code she writes in class uses the Python programming language and a host of packages for data science, including `pandas`, `plotly`, and `scikit-learn`. To prepare assignments, Rosa and her course staff rely on a mix of third-party tools for grading, like Otter Grader [61] and Gradescope [232], which automatically splice out instructor solutions into test cases and check student submissions for correctness. This workflow is stitched together with custom scripts and Makefiles in a `git`

repository. Thus, even a single week of a course requires instructors to work with many tools to produce course materials, as depicted in Figure 1.1.

These tools are powerful and many are freely available; without them, Rosa would be unable to create lessons that can realistically capture the complex nature of analyzing real-world datasets for the hundreds of students in her course. This course infrastructure is especially important to help run large-scale courses in subjects like programming and data science where demand has skyrocketed in the past decade. To provide high-quality pedagogy to hundreds (even thousands) of learners, instructors use course infrastructure that helps them collaborate with course staff, keep course websites up-to-date, catch bugs in assignments before release, distribute assignments, grade exams, and more. At the same time, working with all of these tools can be bewildering, even chaotic. Ultimately, for instructors, especially ones who teach programming and data science, designing a lesson is rarely a process that happens completely in the mind. **Instead, instructors work in tandem with a heterogeneous collection of software tools to create effective learning materials.**

Despite the availability of software tools, instructors still find themselves needing to repurpose tools that weren't originally designed for instructional use. For example, the instructors I collaborate with all use tools like `git`, Python `doctests`, and computational notebooks like Jupyter. These tools were originally designed for professional programmers working on large software projects or data scientists analyzing complicated datasets, not instructors teaching large numbers of students. This mismatch of intent causes added complexity: a typical course employs many custom scripts and infrastructure to work around limitations in existing software tools, because these tools make incorrect assumptions that the user is a software developer, not an instructor.

Thus, instructors who want to improve their lessons must not only face the intrinsic complexity of pedagogical design but also wrestle with extrinsic complexity of software tools that weren't designed with instructor needs in mind. Instructors like Rosa already invest dozens of hours per week designing and updating lectures, assignments, and exams – the essential tasks

of running a course and keeping it relevant. On top of this work, instructors of technical courses also spend much time managing and customizing course infrastructure cobbled together from an ad-hoc mix of software tools.

1.1 Purpose and Thesis Statement

The purpose of this dissertation is to answer the question: *How can software tools help, rather than hinder, instructors as they design, update, and deliver learning material for large courses?* One of the barriers to designing tools that can better serve instructional work is that the work itself is poorly understood. To address this, I take an *instructor-centered* approach – by deeply understanding the needs of instructors and the capabilities of existing software tools, we can chart a path forward for designing tools that directly support the tasks that instructors need to do.

My thesis is that:

> **Instructor-centered approaches enable the design of tools that directly support teaching at scale.**

This dissertation is the first work that views instructors as end-users of software tools and provides the first evidence that this perspective can provide insight for tools that better support the work of instruction.

1.2 A Note on Human-Centered Design

From the outside looking in, research can seem highly impersonal – by its nature, research is presented as a logical sequence of deductions from established prior knowledge. As such, I'd like to take a brief moment here to speak to the reader as a fellow researcher. Much of the work presented in this dissertation was originally motivated by my personal experiences as a long-time instructor of programming and data science, where I've spent my fair share of hours debugging and wrestling with software tools that seemed to hold promise for teaching yet often

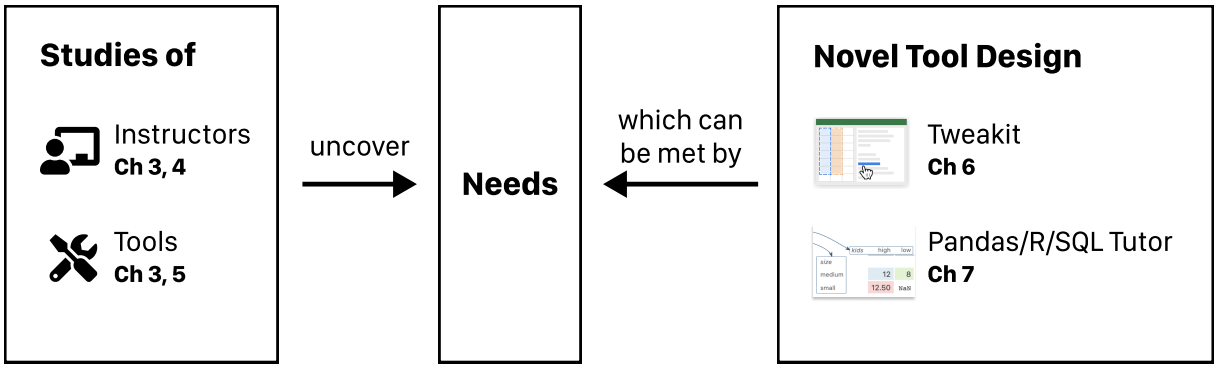


Figure 1.2. This dissertation takes an instructor-centered approach to tool design. Studies of instructors and their tools (left) contribute unmet needs (middle). To address these needs, this dissertation offers novel tools (right).

got in the way instead. In search of solutions, I found much research about *instruction* but very little about *instructors*. There are many studies about teaching methods and strategies, but most research makes the implicit assumption that instructors can trivially implement these findings in practice. Through this dissertation, I hope to convince you that this assumption no longer holds, and that we should design tools that remove obstacles in the way of instructors who want to improve their pedagogy. But even more importantly, I hope to convince you to consider adopting human-centered design methods in your work. If your work will be used by other people, I encourage you to start by deeply understanding your users' goals, desires, and challenges by immersing yourself as much as possible in your users' work. Then, develop an understanding of your users' existing tools by making explicit the assumptions that the tools make about users. It's my hope that by applying human-centered principles, you can gain new insight about how to better serve your target populations through your work.

1.3 An Overview of this Dissertation

This dissertation makes two primary kinds of contributions, depicted in Figure 1.2. First, it contributes needs: previously unstudied challenges that instructors encounter when they hit limitations of their tools. Second, it contributes interactive systems: program visualization tools that support instructors by addressing the needs of their teaching work.

Chapter 2 reviews related work about understanding instructor needs and about tools that instructors use for teaching. It makes recommendations for research to better understand and address challenges that instructors face.

Chapter 3 and Chapter 4 focus on understanding people – in this dissertation, instructors who teach large-scale programming and data science courses. These two chapters present novel research in the form of case studies drawn from my and my collaborators' experiences teaching. The key finding of this work is that instructors constantly wish to update their course materials to improve their pedagogy, but must also work around many limitations in the software tools they use to create and manage their course materials at scale. I frame this finding as a design opportunity to inspire tools that can better meet instructor needs.

Chapter 5 builds on the work of the previous chapters by understanding computational notebooks, a prominent group of tools that instructors use for teaching. This chapter contributes the first definition of computational notebooks and a design space that summarizes 60 notebook systems across academic and industry. The design space motivates recommendations for how notebooks can provide better affordances for teaching, for example by enforcing execution order, allowing for collaboration, and by supporting multiple presentation modalities (e.g. as a script, or as a slideshow).

Chapter 6 and Chapter 7 contribute novel software tools motivated by one recurring set of challenges surfaced in Chapters 3-5: instructors wish to create visual representations of programs to help learners understand how code transforms data. However, current tools including computational notebooks don't directly support this task, leaving instructors to manually instrument code to display intermediate values and draw explanatory annotations. And, whenever instructors update their code, they must also update their annotations by hand, adding yet another logistical challenge to the work of updating course materials. To address this need, Chapters 6-7 apply program visualization techniques to automatically visualize intermediate values in programs and generate annotations to explain data table operations, even ones that transform many data values at once.

Chapter 8 concludes by summarizing the findings of this dissertation and presents an vision for future research to support instructors in their work.

Chapter 2

Related Work

The work of instructors has changed rapidly over the past few decades, especially for technical subjects like programming and data science where instructors face increasingly large enrollments. To handle scale, instructors use a variety of software tools in their courses and collaborate with other instructors and course staff. Data science courses in particular experience frequent iteration, so this chapter begins by reviewing studies that examine data science programs and factors that influence their design (Section 2.1). The chapter continues by providing an overview of tools for managing and teaching large courses (Section 2.2). Lastly, I draw an analogy between learners and end-user programmers, reviewing characteristic end-user programming activities that people perform to understand unfamiliar code (Section 2.3). Throughout the chapter, I point out research opportunities from past work that motivate future studies and tools.

2.1 The Design of Data Science Programs

Data science as a discipline was established in part because of the need for people who can combine computing and statistical knowledge for data analyses. This need is acknowledged by scientific research [142], industry [107], statisticians [108, 98], and computer scientists [77]. To this end, a prominent National Academy of Sciences report calls for undergraduate data science programs to include topics from both computer science and statistics [99]. However,

developing data science curricula is challenging for instructors who need to decide what topics to include from these two disciplines.

This challenge of balancing disciplinary focus is reflected in papers on data science education from the past five years. One theme from this past work on data science curricula design is that each instructor strikes a different balance between computing and statistics content. For instance, computer science instructors might add data science topics to existing computing courses, thus focusing on computing more than statistics. Instructors have taken this approach in introductory courses for computing majors [105], computing courses for non-majors [70], and data systems courses [87]. Similarly, statistics instructors include data science topics into statistics courses that might otherwise not include computing, including introductory courses [72, 186] and more advanced courses [201, 133]. Finally, instructors have designed brand new courses that try to focus equally on both computing and statistics [110, 217, 66]. These courses tend to be listed under a data science department rather than a computer science or statistics department. As a whole, this set of prior work provides examples of successful course designs. In contrast, we take a broader view in Chapter 4 by examining how computer science and statistics instructors approach data science topics differently rather than arguing for a specific course design.

Beyond single courses, data science degree programs also differ in how they balance computer science and statistics content. Some programs place more course hours in computing [116, 216], some programs place more in statistics [223, 218, 75, 67], and other programs have equal course hours in computer science and statistics [65, 137]. Rather than debating what the goals of a program should be, Chapter 4 describes the thinking that computer science and statistics instructors go through when setting learning goals in the first place.

2.2 Tools for Teaching Programming and Data Science Courses at Scale

Technical courses, like programming and data science courses, seek to give learners learning experiences that closely match how engineers and data scientists program in practice. Many of these courses also face the challenge of scale – with hundreds of students, instructors need tools to automate tasks like managing course content, executing student code, and visualizing code for teaching. This section reviews the current state of these software tools and how they motivate the research in subsequent chapters of this dissertation, especially Chapters 3, 5, and 7.

2.2.1 Managing and updating course materials

Large-scale courses motivate instructors to automate as many logistic tasks as possible. To this end, there is much research into the implementation of specific *components* of large-scale courses. For instance, Sharp et al. described the code submission, execution, and autograder framework used in Harvard's large intro. programming course and MOOC (CS50) [229]; lead instructor David Malan also documented its server sandbox environment [188] and the entire suite of open-source software infrastructure [187] used to manage CS50 logistics. Basu et al. [71] and Sridhara et al. [235] presented innovations in automated code grading and feedback systems deployed in UC Berkeley's intro. programming course. Others have documented how instructors use GitHub to manage course materials and to enable students to submit coding assignments [257, 118]. Many such tools are focused on programming-related courses, but some are topic-agnostic: Gradescope facilitates AI-assisted grading of handwritten assignments and exam questions [232]. PeerStudio [175] and Talkabout [174] facilitate peer feedback and small-group discussions, respectively, in large-scale MOOCs with open-ended learning activities. And ProjectLens helps instructors and students manage group project logistics in HCI MOOCs [95]. Similarly, we discovered that assignment submission and grading infrastructure were hard to maintain. But in contrast to these related papers, which each focus on the technical details of

individual software components, we see a research opportunity to understand *instructors' holistic experiences* in maintaining and updating course materials in the face of increasing scale, which we explore in Chapter 3.

The closest analogues to the above teaching activities come from software engineering research. Specifically, software maintenance [91] and evolution [125] are classic topics of study that date back to the birth of computer software in the 1960s. Unlike physical systems, software does not physically degrade over time, but it does need to be updated in light of changes to the surrounding environments that it runs upon (e.g., operating systems, software libraries, hardware). Similarly, course materials do not physically degrade but also need constant updating in light of changes to their 'environment' such as new developments in the field. Also, since our courses involve lots of programming, our materials include software components that must be routinely updated, such as incorporating new versions of Python libraries for data science.

Dependencies also make software maintenance more challenging since it is hard to change one component independently from the dozens or hundreds of code libraries that it depends upon [103]. We faced similar issues with dependencies amongst different intricately-linked components of our course materials, which made updates fraught with unexpected surprises.

More broadly, we can also draw an analogy between teaching large-scale courses and collaborative peer production of online resources, such as groups of writers convening to update Wikipedia [132] and other Wiki knowledge bases [160], and programmers convening to produce open-source software [104, 115]. While not as large-scale as some of these projects, the courses we investigate in Chapter 3 often have a dozen or more staff members collaboratively updating materials that consist of dozens of lecture slide decks, lab handouts, code examples, and homework assignments. This collective work is reminiscent of Geiger et al. describing the "invisible and infrastructural work" involved in keeping open-source software projects running, such as "uncredited" behind-the-scenes tasks like updating documentation [124]. In our case, instructors spend large amounts of time on invisible infrastructural work to keep their course materials maintained and up-to-date.

Thus, to our knowledge, the work presented in Chapter 3 is the first to describe the challenges that instructors face in maintaining and updating materials for large-scale technical courses.

2.2.2 Computational Notebooks

Over the past few years, an increasing number of courses have started to use computational notebook systems for specifying, writing, and executing code because these systems have also become a standard tool by professional programmers and data scientists.

Computational notebooks trace their lineage back to the vision of literate programming that Knuth first articulated in the early 1980s [164]. Knuth envisioned software being written like literature, with code and expository text interwoven into a single document to facilitate a natural human reading order. To implement this vision, he created the WEB system to interweave Pascal code with TeX-formatted exposition. The Mathematica scientific environment extended these ideas by creating the first computational notebook in 1988 [138], followed by a similar feature in Maple in 1989 [51]. Because early notebooks were embedded within proprietary environments, they remained niche products within the scientific community for the next two decades. As free and open-source web technologies matured in the 2000s, Perez and Granger developed the IPython Notebook in 2011 [48], which evolved into the popular Jupyter Notebook in 2015 [205].

The notebook systems we analyze in Chapter 5 embody several facets of programming research: end-user programming, live programming [239], and literate computing. Notebooks are widely-used environments for exploratory [153] and end-user programming, which Ko et al. define as coding as a means to an end (e.g., to produce data science insights or research findings) rather than to create artifacts for broader public use [166]. That said, some notebook systems have affordances for creating public artifacts such as web dashboards or reproducible software packages. Also, modern notebooks embody the spirit of literate *computing*, which is a generalization of Knuth's literate programming vision that mixes code with both exposition and rich outputs such as images, videos, and interactive widgets [121].

Aside from publishing academic papers on new notebook systems (summarized in Section 5.3), researchers have also studied how data scientists use notebooks and what obstacles they face. Rule et al. discovered a pervasive tension between Jupyter notebooks used for exploration (i.e., prototyping analyses) and explanation (i.e., sharing research results) [222]. Kery et al. found that data scientists often cleaned up exploratory notebooks into explanatory ones by using ad-hoc methods like alternately expanding and consolidating code cells [156]. A broader study discovered pain points for notebook users along the entire workflow spectrum ranging from setup to exploration to sharing [94]; the 60 systems that we analyze in this paper are often attempts to address specific pain points along that spectrum. Lastly, CSCW researchers investigated how data science teams collaborate using notebooks and discovered the limitations of current notebook systems for both asynchronous and synchronous collaboration [244, 195, 258].

To compare and contrast the designs of literate computing systems, a PPIG workshop paper [121] surveyed 12 literate computing projects, 7 of which were notebooks. Similarly, Merino et al. surveyed 12 notebooks and 4 other systems as a formative study to inform the design of their Bacatá system, which generates notebook UIs for DSLs [190]. Both of these analyses were small in scope and did not focus on mapping a design space of technical notebook features. In comparison to the prior work in this area, the work presented in Chapter 5 is to our knowledge the first comprehensive design study of dozens of notebook systems across academia and industry.

2.2.3 Program Visualization Tools

To explain how code works, instructors frequently wish to provide visual representations of programs. Chapter 7 introduces a set of tools that automatically generate diagrams to explain how Python, R, and SQL code transform data tables.

The two closest related projects to ours – Data Tweening [158] for SQL and Datamations [212] for R – use animations to show how data tables get reshaped. We took a complementary design approach by rendering static diagrams that can be used in screenshots and

presentation slides, and we also support a larger set of operators necessary to cover what is taught in introductory data courses.

Interactive data wrangling systems such as Wrangler [149], its proactive extension [130], and Unravel [231] show inline visual previews of table reshaping operations. These were designed to assist working data scientists; in contrast, our tools were made specifically for teaching, so they also show side-by-side before-and-after comparisons and fine-grained mappings to Python/R/SQL code.

Our work also extends the line of computing education research on program visualization tools [233]. These tools, such as Jeliot [194], UUhistle [234], and Python Tutor [128], visualize the step-by-step run-time state of code written for introductory programming courses. Our tools are inspired by Python Tutor and extend its reach to introductory data courses. This required us to design a different set of visualizations focused on annotating table transformations instead of on variables, objects, pointers, stack frames, and heaps.

Lastly, tools such as QueryVis [182], SQLVis [191], and SQL EXPLAIN [198] can help users to understand SQL query plans. These tools show the query plan and the *schemas* of affected data tables; but they do not show the concrete data that is being transformed, since their goal is to emphasize a higher level of abstraction. We build upon those ideas by showing both the query plan and the actual data tables on-demand when the user clicks on nodes in the query plan tree. Our intuition is that showing actual data can help instructors explain SQL execution more concretely to students.

2.3 Learning Activities as End-User Programming

In introductory programming and data science courses, instructors present numerous code examples for learners to understand, reuse, and modify for their own work in their courses and beyond. Learners who encounter these initially unfamiliar code snippets can be viewed as a kind of *end-user programmer* while they progress towards expertise. This section reviews the literature

that characterizes end-user programming activities, how people reuse code opportunistically, and how live programming environment can help people understand code examples. The tools presented in Chapters 6 and 7 draw upon this prior work in particular.

2.3.1 End-user programming

End-user programming typically refers to activities where a person with no formal programming education writes code to accomplish a task; examples include biologists writing code to analyze data, or accountants building spreadsheet macros. Prior work has aimed to understand and support their programming and debugging activities in various domains such as spreadsheets, CAD [199], web mashups [252], home automation [88] and hobby electronics [80]; approaches range from visual languages, programming by example [113], mixed-initiative programming [129], help seeking [159, 243] and natural languages to formal engineering (e.g., testing, verification and versioning) to ensure quality [196, 165].

The challenge of defining end-user programmers exclusively as people with no formal programming education is that it conflates behaviour with expertise—although they are appear correlated, they are in fact orthogonal. On one hand, expert programmers regularly find themselves in the position of a non-expert, needing to code in an unfamiliar language, API, or project. They often need to prioritize working code for a specific short-term aim (e.g., a script to generate plots for a presentation) over comprehensibility and maintainability. On the other hand, there are non-expert programmers such as novice spreadsheet users who want to build well laid-out and well-tested spreadsheets for long-term use by their team. Consequently, end-user programming is better viewed as an activity that both experts and non-experts engage in, and end-user programmers can then be defined as people who engage in end-user programming behaviors [165].

In the formative interviews of Chapter 6, we found that data analysts who tweak code often do not self-identify as having programming expertise. Those who do may nonetheless lack expertise in the data scripting language they are trying to reuse—in our case, Python. Of the

small fraction who do have Python expertise, an even smaller fraction are aware of, let alone have expertise in, the APIs of specific data manipulation libraries, such as `pandas`. Thus, code tweakers are better viewed as end-user programmers than as software developers, even if they have programming expertise.

2.3.2 Opportunistic code reuse

To help our target end-user population, it is necessary to characterize the kind of end-user programming activity they are doing when code tweaking. This is a blend of authoring, debugging, and information seeking that is best captured by the term 'opportunistic code reuse.' This is a specific instance of opportunistic programming [85], which is defined by prioritizing getting things done over code comprehension and maintainability. Opportunistic programmers reuse snippets of code from prior versions of the same code [236], from others' code or in specialized code repositories [219] and from the internet [83]. They piece them together using glue code and use ad-hoc debugging techniques such as commenting code and print statements over formal debugging tools.

A closely related, but distinct activity, is exploratory programming [154, 151]. Here users write code to experiment or prototype with different ideas: the goal is open-ended and evolves through the process of programming, and the programmer is not attempting to match a specification. Although exploratory programming can involve opportunistic behaviors, it is clearly a different phenomenon.

The term 'tinkering' also appears in the literature. Burnett et al.'s GenderMag framework [89] uses the word 'tinkerer' to refer to an intrinsic personality trait of some end-users that causes them to be more inclined to program, edit, and customize software. Our data did not suggest that our users were universally tinkerers—rather, they were usually only motivated to do the minimum tinkering necessary to get their work done. Terms such as 'customization', 'configuring', and 'tailoring' [114, 148, 240] describe setting parameters of existing programs, but not direct modification of a program's source code.

The problems of opportunistic code reuse can be characterised in terms of Ko's learning barriers [168]. In particular, they might face difficulties selecting the right code snippet to reuse (a *selection* barrier), understanding how to use a code snippet and adapting it to the task at hand (a *use* barrier), putting together different code snippets to seek the desired output (a *coordination* barrier), and understanding how the code snippet or the put-together code works (or the functions or the lines in the code snippet) works (an *understanding* barrier).

Many prior tools for improving opportunistic code reuse from the internet are aimed at helping people find relevant code examples easily [83]. Such tools may additionally capture the source of the reused code, in case it needs to be revisited [135]. Still other tools are suited towards expert programmers for whom reading code is sufficient for comprehending it; these simply list candidate snippets that a programmer can evaluate by reading (and users seek additional information only when they need). However, in the formative studies of Chapter 6, we did not get the sense that finding relevant code snippets was the biggest bottleneck; rather, the difficulty was in understanding each snippet and how various snippets may be combined to solve the task at hand.

Non-experts additionally need help understanding code. Previous work has explored summaries of code snippets [255], or integrating additional information available on the web (e.g., examples and explanations) as part of code search results [143], or enriching webpages containing code snippets with comprehension tools [259]. During reuse, programmers don't just look at code, but the context in which the code is used: they look at examples of code use and adapt the entire usage instance to their task's context; Rosson calls this "reuse of uses" [219]. Community guidelines on websites such as Stack Overflow recommend that code snippets are accompanied with examples,[1] signalling the usefulness of examples in code reuse, even for experts. Thus, the approach we took in Chapter 6 was to improve the intelligibility of the code through its output, since we are likely to be able to successfully run the code (as opposed to code summarization or retrieval of additional information, which would have variable rates of

[1] https://stackoverflow.com/help/minimal-reproducible-example

success).

Programmers might want to retrace their steps when working in an opportunistic manner, and thus need support for fine-grained backtracking [256], or runtime event-centric explanations (e.g., why did or didn't a method get invoked?) [167]. But existing debugging tools don't make it easy to inspect arbitrary statements, especially for non-experts. Although traditional debugging tools offer sophisticated ways of pausing code execution to inspect values, non-expert programmers often do not know how to use them. Moreover, in opportunistic programming, users cannot or do not want to invest effort in learning, instead prioritising finishing the task at hand in any way possible. Even expert programmers tend to use more ad-hoc methods for debugging, e.g., print statements and code commenting, rather than using the debugger—which we postulate is due to the high interaction costs of setting up a debugger and adding breakpoints. Although rich in feedback, computational notebooks suffer from similar pain points with high expertise requirements and interactional costs [93, 139, 155]. TWEAKIT addresses this limitation by allowing users to inspect output simply by placing their cursor in the relevant piece of code.

2.3.3 Live programming

The importance of interactional costs to opportunistic end-user programming cannot be understated. When code visualizations are always-on, as opposed to manually triggered, users develop and adopt unique strategies for code comprehension and navigation [183]. Let alone manual interaction costs—even slow output can be an issue: in data analysis, a consciously imperceptible response latency can unconsciously act as a significant deterrent for exploration, reducing the user's coverage of the data set, as well as the rate at which they make observations and hypotheses [185].

Live programming environments allow users to edit a program as it runs, or automatically recompiles and executes the program as the user edits [238]. Such real-time, interactive feedback can be beneficial support for novice end-user programmers. But as Ko et al. found [168], sometimes people want to see what the "factory" is producing, and at other times, they want to

inspect what each machine does and what each machine produced. Live programming is typically concerned with rendering the output of the entire program, and this is suited to understand what individual bits do when writing code from scratch; but it is not very well suited for reusing code, which requires the programmer to hypothesize potential edits, edit the code and then see the live output to confirm the hypothesis. Some live environments allow developers to inspect what the compiler evaluated each statement to during live execution [76]. This interaction, like runtime event-centric explanations, is well suited for debugging, but doesn't directly address the problem of understanding what a single statement or method call does, because to do so requires the ability to compare the system state before and after the statement was executed, not just the output of the statement. As this was a priority for non-expert end users, we built this into the comparison feature of TWEAKIT, the tool presented in Chapter 6, which naturally extends the cursor preview interaction to allow for ad-hoc comparisons between arbitrary different steps of the program. This design goal also informed the design of the tools presented in Chapter 7, which display and annotate a full sequence of intermediate values for code expressions.

Live programming environments typically lack granularity, and granular debugging tools typically lack liveness (in the sense that they incur interaction costs to configure). Thus, the ability to inspect the output of individual code expressions with little or no interactional cost is a core novelty in the interaction design of the tools presented in Chapters 6 and 7. Although the individual ideas are not by themselves new, their combination is.

Chapter 3

The Challenges of Evolving Technical Courses at Scale: Four Case Studies of Updating Large Data Science Courses

Instructors who teach large-scale technical courses, especially on data science and programming, must do a large amount of logistical work when updating their courses. All of this behind-the-scenes labor takes time away from the pedagogically-meaningful work of teaching students. Over the past five years, I and seven other instructor collaborators have created and updated eight courses for an undergraduate data science program that serves over 2,000 students per year. We present four case studies from our teaching experiences that highlight major challenges in maintaining and updating technical courses: 1) There were intricate dependencies between course materials, so making updates to one part of the course would require updating many other parts. 2) We needed to maintain several variants of course materials such as assignments. 3) We wrote large amounts of ad-hoc custom software infrastructure to manage logistics. 4) We could not easily reuse software written by others. Our case studies point to design ideas for instructor-oriented tools that can reduce the logistical complexities of teaching at scale, thus letting instructors focus on the substance of teaching rather than on mundane logistics.

3.1 Introduction

Suppose Alli is an instructor teaching an introductory data science course. She first created this course four years ago, and enrollments have doubled each year. This term, Alli wants to move the lecture on data tables earlier in the course since last year students were confused by this concept. Although this seems like a simple change from a pedagogical standpoint, she runs into major logistical challenges. After moving the lecture, Alli also needs to update many other pieces of course materials to keep them up-to-date, such as discussion worksheets and assignments. But, making these updates requires knowledge of multiple custom infrastructure scripts and third-party software tools that her TA staff uses to generate those course materials. At the scale of Alli's course, she must not only ensure good pedagogy but also pay close attention to variations in hundreds of assignment files, manage autograder software configurations, and debug programming assignments as the underlying software libraries evolve.

This example scenario highlights the reality that large courses, especially on technical topics such as programming and data science, require instructors to manage a tremendous amount of *invisible behind-the-scenes logistical work*. All of that work takes valuable time and energy away from what they actually want to do: teach students. For instance, one collaborator said the following during a discussion of the challenges we faced in maintaining and updating our large courses:

> *"I had 14 to 15 hour work days, every day. It was a lot of work, and NOT the fun kind of work because I didn't get to interact with students during any of it."*

Over the past five years, I and seven other instructor collaborators on this study have created and updated a set of eight courses for an undergraduate data science program that serves over 2,000 students per year. We have witnessed firsthand the tremendous growth in enrollments to data science and computing majors over the past half-decade [68, 147, 171] – some of our course enrollments have *nearly doubled every year.*

To handle such increasing scale, instructors of technical courses like programming and

data science adopt automated workflows to support large class sizes, such as creating software infrastructure to automatically distribute and grade assignments [229]. While these innovations have benefits, they also require lots of work to manage and debug. ***What logistical challenges do instructors face when maintaining and updating large technical courses?***

This question is important because dealing with logistical challenges takes time away from the pedagogically-meaningful work that instructors want to do. By understanding the complexities that instructors face in their daily workflows, we can move toward streamlining the process so that they can focus on the substantive parts of actually teaching well.

To investigate this question, we present four short case studies drawn from our teaching experiences over the past five years. Each highlights a specific challenge we faced when updating technical courses. Table 3.1 summarizes each challenge along with representative examples: 1) There were intricate dependencies between course materials, so making updates to one part of the course would require updating many other parts; and oftentimes these dependencies were invisible, which led to inconsistencies that frustrate students. 2) We needed to generate several variants of course materials, such as versions of assignments with and without solutions, then keep those consistent with related materials. 3) We wrote large amounts of ad-hoc custom software infrastructure to manage course logistics such as running student code on servers, accepting homework submissions, doing both automatic and manual grading, and maintaining variants of course materials. 4) We could not easily reuse software written by others, such as off-the-shelf learning management systems or even code written by prior terms' instructors.

Research contributions: This study is, to our knowledge, the first to characterize the challenges that instructors face when doing maintenance and update work for large technical courses. We focused on data science courses in particular since they combine concepts from multiple technical disciplines – mathematical theory, applied statistics, computer programming, data visualization, and the social sciences (e.g., the ethical and social context of data use). Thus, we believe that parts of Table 3.1 can generalize across other types of technical courses. More broadly, we hope

Table 3.1. From our experiences teaching large data science courses, we discovered four main sets of challenges when maintaining and updating course content. This logistical complexity hindered our ability to make pedagogically useful improvements.

Challenge	Description	Representative Example
Intricate dependencies between course materials	Course material depends on each other; updating a single lecture can cause worksheets and assignments to become out-of-date.	Adding hypothesis testing concepts to a course created a cascade of unforeseen updates, even for material that was not directly related to the topic (Section 3.4.1).
Maintaining consistent variants of course materials	Even single pieces of course content require careful management of multiple file variants and computing environments.	For many assignments, we have an instructor version, a version released to students, and student submissions. However, student Python versions sometimes differed from the autograder's Python version, causing subtle bugs and student confusion (Section 3.4.2).
Writing ad-hoc software infrastructure to manage scale	Each course has its own scripts and software written by course staff to automate tasks like autograding. However, every piece of infrastructure code requires work to maintain.	One course's infrastructure, which started out as a single file of LaTeX macros, grew to become a collection of scripts that stitched together multiple programming languages (LaTeX, Python, bash) and software tools (Section 3.4.3).
Cannot easily reuse software written by others	Most software development tools do not fully enable instructors' desired workflows, so instructors still need to write custom code and pay close attention to software updates.	When a third-party software tool released important bug fixes, it also added backwards-incompatible changes that broke our existing course infrastructure (Section 3.4.4).

our case studies open up a dialogue about how to support instructors of large university courses, who are often temporary lecturers, teaching-track professors, and graduate students [237] who must bear the brunt of this invisible behind-the-scenes labor.

In sum, this study's contributions are:

- Case studies that highlight four sets of logistical challenges in maintaining and updating large technical courses.
- A call to develop better practices and domain-specific tools to support instructors in managing these challenges so that they can focus more of their time on teaching.

3.2 Methods

This study investigates two main research questions:

- Why do instructors need to make updates to large-scale technical courses that are already well-established?
- What challenges do instructors face when maintaining and updating these large technical courses?

We addressed these questions by reflecting on our own experiences as data science instructors and synthesizing them into four short case studies. Each case study presents a challenge we faced along with a reflection of its broader implications for our research questions. The main benefit of this case study method is that we can candidly introspect on our own experiences and discuss them as co-authors in a way that is not as feasible with external interviews or surveys. (But see Section 3.2.3 for limitations of this method.)

3.2.1 Case Study Participants

For a case study to be effective, its participants must be representative of the general population that its research questions aim to target. In our case, the 8 co-authors of this study

Table 3.2. The courses that this study's authors created for a data science major at a U.S. university. Students = approximate enrollment per term, TA = number of TAs this term, Terms = how many terms has this course been taught. Instructors = how many different people have taught this course.

Course Topic	Students	TA	Terms	Instructors
Intro. to Data Science	280	16	16	5
Data Structures	250	5	11	2
Theory for Data Science I	150	9	14	5
Theory for Data Science II	150	9	13	2
Algorithms for Data Science	150	10	13	3
Applications of Data Science	100	9	12	3
Data Science in Practice	550	10	17	4
Capstone Project	220	6	7	1

collectively helped create the data science major at a large public Ph.D.-granting institution in the United States. We designed and taught large data science courses for the major, including *all six required lower-division courses*, a popular elective course, and a capstone project course. These courses cover foundational topics in data science, including data manipulation, data structures, visualization, and statistical modeling. We mostly use widely-adopted Python data science tools like `pandas` [189], Jupyter notebooks [206], and `scikit-learn` [203]. Thus, we believe that we are well-positioned to reflect on our firsthand experiences to highlight the challenges of updating technical courses.

Methodologically, our approach follows the precedence set by prior research papers that are *case studies of the authors' own firsthand experiences*. Some notable examples include lessons learned from simultaneous deployments of a MOOC and residential course [169], reflecting on five years of Georgia Tech's online masters program [146], a case study of an HCI MOOC with group-based design projects [95], and six years of reflections on an internationalized CS curriculum [210].

3.2.2 Overview of Courses

Table 3.2 summarizes the eight data science courses that we created and updated over the past five years. Average course sizes ranged from 150 to 550 students. We hired large teams of teaching assistants (TAs) to help manage these courses, with three hiring 10 or more TAs each term. TAs held a wide variety of responsibilities, including hosting discussion sections, holding office hours, attending staff meetings, updating and releasing assignments, updating the course website, and grading assignments. We hired both undergraduate and graduate students as TAs, although the majority were undergraduates who did well in prior offerings of the course [193].

Seven of these courses are required for data science majors at our university, and one is a popular elective. Typically, each week we presented three hours of lecture and an hour-long discussion section. Weekly homework assignments consisted of a mix of math, computer programming, and open-ended narrative write-ups.

Moreover, all these courses were well-established, with the majority having been offered in at least ten past terms (see 'Terms' column in Table 3.2). There are four quarter-long terms per year at our university. Most courses had multiple instructors teaching it in the past ('Instructors' column); for instance, our intro course had 5 instructors teaching 16 offerings over the past five years.

These courses are large-scale and complex to manage not only due to high student enrollments but also because of large staff sizes (often with over 10 TAs), and because they are taught by many instructors over multiple past offerings. Thus, we feel they are appropriate to use as the settings for our case studies in this study.

3.2.3 Limitations

The usual limitations of a self-reflective case study apply here. This study's co-authors all work at the same large public university in the United States. This left out other settings where data science is taught, such as bootcamps, MOOCs, and online workshops [171]. On the other

hand, having multiple instructors within the same data science program captures complementary perspectives from different instructors of the same courses. For instance, we often rotate between teaching each other's courses. During an academic year, an instructor typically teaches between 3-5 courses within the program. We acknowledge the limitations of this self-selected sample, so to compensate we tried to estimate how often the issues we encountered came up for peers who taught at other types of universities and reported issues that were more likely to generalize. In the future, interviewing or surveying our colleagues who work at other institutions can help reduce these limitations.

Another limitation is that we only teach data science and programming courses, but other technical courses across STEM fields, such as math and physics, also have large enrollments. We chose to focus on these fields since data science and intro programming are amongst the fastest-growing and highest-enrollment courses at major U.S. universities and MOOCs [68, 147, 171]. We also believe that data science courses are good representatives of technical courses *since they combine concepts from multiple STEM domains*: our courses in Table 3.2 cover mathematical theory, applied statistics, computer programming, data visualization, and written narrative assignments about the social and ethical context of data. Thus, studying these courses could shed light on problems that courses from other technical domains may encounter at scale. However, our findings may not generalize to all types of technical topics, especially those that do not involve as much math or programming (e.g., biology).

3.3 What kinds of updates are needed for large technical courses?

First we describe the kinds of updates that are typical for large-scale technical courses such as our data science courses (Table 3.2).

We all taught established courses that have each existed for at least three years. Thus, every term we all had the option of "replaying" the course verbatim—we could have reused the

material that already existed without changing any of it. In fact, this material was often written by ourselves in a prior offering of the course. Yet, all of us shared a common desire to update our course materials despite the effort required to do so. Here are the three main kinds of updates and common reasons for making them:

1) Improving lecture materials: The most common reason for updating existing lecture materials was observing students struggle with an important concept from the prior term. In response, we often trimmed, expanded, and moved topics. For example, at times we allocated more lectures for basic statistics concepts or decided to move them earlier in the course to give students time to practice. Larger classes often involve more frequent lecture edits to accommodate more students with varying levels of prior experience.

We also updated the examples used in lectures to keep up with current trends. For example, most of us felt students were more engaged when lectures incorporated personal interests or data related to current news events. Related, since data science and programming technologies (e.g., software libraries and APIs) change rapidly year-to-year, the code examples used in lectures must be continually updated or else we risk teaching outdated content.

2) Updating assignments to keep them fresh: In established courses, assignment solutions inevitably get leaked online after several offerings, which increases the odds of cheating. Table 3.2 shows that most of our courses had been taught over ten times before. Many of us felt that students were negatively impacted by assignment solution leaks, to the point that we avoided repeating an assignment in consecutive offerings. Thus, assignments must be constantly updated to help maintain academic integrity.

Another reason for assignment updates is that data science courses often rely on the latest data or real-time data streams to provide timely and motivating assignments for students. Some assignments used election data or real-time Twitter feeds, so they had to be updated to incorporate the latest data changes.

3) Adding more content as enrollments grow: Student demand for our courses has been at

an all-time high due to more people wanting data science and programming jobs. Most of our courses in Table 3.2 have doubled or tripled in size over the past few years.

We had to create more content as enrollments increased. First, we needed to write more internal documentation for our TAs. To ensure that even new TAs (who are mostly undergraduates) can deliver high-quality instruction, we had to write more detailed TA guides to provide structure for TA sections and create more detailed grading rubrics. We felt that these additional teaching materials were required to maintain a consistent student experience at larger scales. For instance, when our data structures course grew from 60 to 250 students, we made discussion worksheets so that students could get useful practice no matter which section they attended.

Larger courses also mean a greater diversity of student backgrounds; in our experience, this manifested most prominently in the amount of prior computer programming experience that students had when coming into our courses. When we taught the programming-heavy parts, we found that many students either found programming too easy or too difficult. To address this wide variance, we created supplemental course content to help students who had less programming background. For instance, one of us created a separate problem-solving guide for the data science theory course. We also created a separate track of content for more advanced students who wanted additional depth for a particular topic, for instance by inviting guest lecturers or by adding more lectures that covered recent advances in the field.

3.4 What challenges do instructors face when updating their courses?

We distilled our collective experiences of updating our course materials over the past few years into four main challenges, summarized in Table 3.1. For each challenge, we use a small case study to illustrate a representative example from our experiences. Each case study provides background context, the difficulties we faced in making that course update, and a reflection on

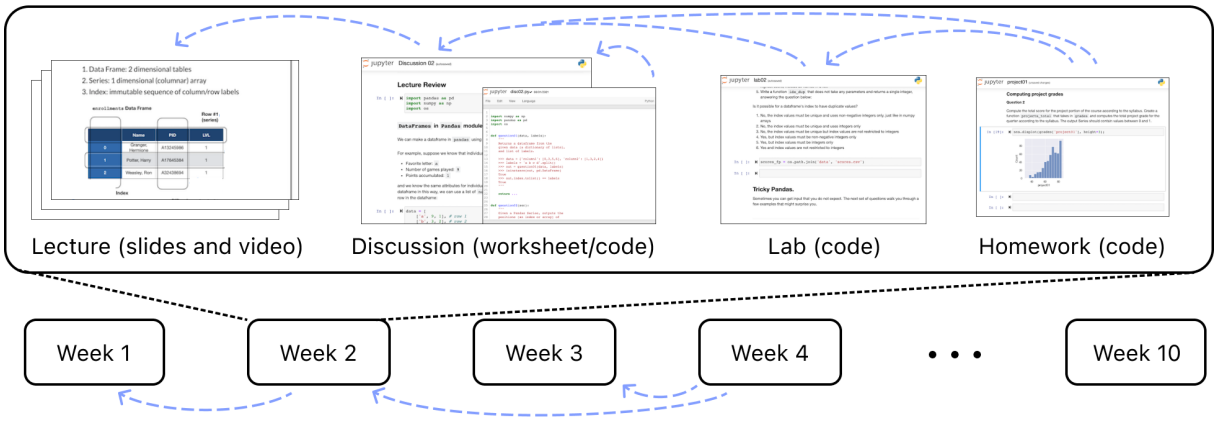


Figure 3.1. Even within a single week in a course, instructors must manage multiple pieces of course content with intricate dependencies (depicted as dotted blue arrows in this figure). In this example from one of our courses, the discussion Jupyter notebook reinforces topics from lecture. The worksheet is accompanied by a separate Python script which must stay up-to-date with the content in the notebook. Lab and homework also depend on the discussion and lecture content. At a higher level, dependencies exist between subsequent weeks. An instructor who wishes to update a single lecture must ensure that all the dependent material is also updated across the entire course. Instructors now do this manually, which is tedious and error-prone.

broader implications. We conclude each one with a list of other kinds of course updates that resulted in similar challenges.

3.4.1 Challenge 1: Intricate dependencies between course materials

Figure 3.1 shows that in technical courses even a single week can involve many pieces of course material that must remain consistent with each other. For example, a typical week might contain two to three lectures. Each lecture has both a set of slides and a Jupyter computational notebook [206, 176] that the instructor presents to show live code and data examples in class. Students also attend discussion and lab sections during the week to reinforce lecture content; those sections each require their own Jupyter notebooks or printed paper handouts. Each assignment also requires its own Jupyter notebook and supplemental files like datasets.

Due to the intricate dependencies shown in Figure 3.1, if an instructor wants to update any piece of course content (e.g., to fix a bug or improve an example), they must also update a batch of related content to keep it all consistent. And although each piece of content depends on

others, these dependencies are hidden and must be maintained manually by instructors and TAs.

Case Study: Substituting or Moving Course Topics

The first course that all students take in our degree program is an introduction to data science. For this intro course, we adapted UC Berkeley's openly-available Data 8 curriculum [111]. However, Data 8 is a 15-week course, so we had to modify it for our university's shorter 10-week academic term. Data 8 is split into three parts: computation (Python), inference (hypothesis testing), and prediction (machine learning). To prune this course down to ten weeks, we did not include inference when we first offered our version. However, a year later we updated our course to make the opposite choice—we removed topics in prediction to teach inference instead.

Why make a change? The motivation for making this change actually came from other instructors in our data science program. Specifically, one of the more advanced data science courses teaches students to do end-to-end data analyses, including hypothesis testing. However, the instructors for that course felt that teaching hypothesis testing from the ground up took too much time away from their core topics, which made us realize it was important for students to learn it first in our intro course. Thus, we decided to replace prediction topics with inference (hypothesis testing) since other courses in the program already covered prediction.

Why did we not include hypothesis testing in the original version of our intro course? Because it was the very first course we created for our data science program, so these later courses did not even exist back then. Although we knew the basic high-level descriptions for later courses, it was not possible to know exactly what would be most important to cover until those courses were actually taught for the first time and we received student and instructor feedback. Thus, during the first offering we decided to teach prediction instead of inference because we felt that a prediction project related to machine learning would be more interesting for students. But once the subsequent courses were launched, we found that having inference in our intro course was better for students. This is an example of updating our course in the face of changing external circumstances such as new downstream courses being created later.

How did we update our course? We adjusted the lecture schedule to emphasize inference instead of prediction. As we updated course materials, we repeatedly encountered the challenge of intricate dependencies: switching each lecture's topic required updating the discussion worksheet, lab assignment, and homework for that week. We also realized that we could adjust the first half of the course to better prepare students for inference. For instance, our old content did not teach students how to draw random samples from a data table, an important skill for inference. Thus, we updated content not only for the weeks that were directly focused on inference but also previous weeks in the course so that students could be better prepared for inference later. This is an example of an update triggering additional updates of dependencies.

Reflection. This case study illustrates how intricate dependencies exist between materials within a course. But at a higher level, it also illustrates how dependencies exist *between courses*. In our case, an advanced downstream course motivated us to swap out prediction topics for inference. Once we made this change, other courses in the program also needed to adjust since their students would have more practice with inference content but less practice with prediction. For instance, courses that previously assumed that students understood how to fit a classification model in Python could no longer do so. This poses a big challenge for instructors: Although we want to make pedagogical updates to improve our courses, it is difficult to predict how much work these updates will actually take to implement. One update could cause cascading changes to many other pieces of material, but we do not know which other pieces of material will become out-of-date until we review them manually.

Other examples.

Here are other times where we encountered the problem of intricate dependencies between course materials.

- Updating references to past assignments in text (e.g. "In the last assignment, we covered these topics.")

- Changing a core software package for working with data tables in Python.
- Moving graph algorithms from an earlier course to a later course in our data science program.
- Removing geospatial data from a course to cover natural language models in more depth.

3.4.2 Challenge 2: Maintaining consistent variants of course materials

In the prior section we showed how updating one piece of course content often results in needing to update many other pieces of content that depend on it. Here we present a related problem: even maintaining a *single* piece of content in isolation (without worrying about dependencies) requires instructors to manage multiple *variants* of that content. For instance, a typical assignment in one of our courses has at least three variants: 1) an instructor version that contains both questions and solutions, 2) a student version that contains only the questions (not the solutions), 3) each student completes the assignment to produce their own personal variant (a completed assignment) to submit for grading.

Case Study: Reacting to Unexpected Python API Changes

One of our courses contains lessons on text data, regular expressions, and natural language models. We give students an assignment where they write Python code to download books from Project Gutenberg [59] and tokenize the text. However, some students ran into unforeseen problems after we upgraded the course Python version from 3.6 to 3.7. This led us to update the assignment.

Why make a change? In the midst of the term we got unexpected reports from students and TAs about an issue where for some students, their code would run correctly on their personal computers but not on the autograder server that grades assignments. After looking into this, we realized that this problem affected only students who had *not* upgraded to Python 3.7. In 3.7, Python changed the behavior of regular expressions. Before, splitting the string `'ba t'` using `r'\s*'` would result in `['ba', 't']`. But in 3.7, the same regular expression produces

`['', 'b', 'a', '', 't', '']`. The challenge here was that some students had installed Python 3.6 from a previous course while others had installed Python 3.7. Our autograder system used Python 3.7, so students who wrote perfectly correct code for Python 3.6 could still fail the tests for 3.7.

Unfortunately, we did not catch this subtle issue until we had already released the assignment. Why not? Our staff-written solution to the assignment did *not* rely on the specific regular expression syntax affected by the Python update, so the staff solution still produced the correct output using either Python version; in contrast, many good student solutions (that were still correct) used a regular expression that broke in 3.7. Also, the webpage that documented the changes in Python 3.7 buried this tiny backwards-incompatible change within a long list of other changes [56]—if it were printed out, this change would be on page 38 within a 42-page document.

How did we update our course? When we found out about this issue, we wanted to update the assignment so that the test cases would detect and notify students when they were using the problematic syntax. To do this, the course staff added test cases to the instructor version of the assignment and regenerated the blank student version of the assignment. However, we then ran into the problem that many students had already downloaded and made progress on the old version of the assignment—some students had even finished the assignment before we discovered this issue. If a student wished to use the new version of the assignment, they would have to download a blank copy of the assignment and copy all of their code over. However, we felt that this was too burdensome for students since there were many pieces of code in this assignment, and we only needed to update one question.

Instead, we sent an email announcement to students to upgrade to Python 3.7 immediately. We also instructed our teaching assistants to make sure students in their discussion sections ran their code using Python 3.7, not 3.6. For future iterations of the course, we emphasized that students needed to install and use the same Python environment as the autograder.

Reflection. This case study highlights the challenges of having multiple variants of course materials. In this example, a single assignment had an instructor file (with both the questions and the solutions) and a blank student file (with no solutions) that needed to stay consistent with each other. But each time a student downloaded the assignment and worked on it, they created yet another file that could become inconsistent with the instructor version. When the course staff fixed a question in the instructor version, all previously-downloaded student copies of that assignment (which could be hundreds in a large course) became out of sync.

Another related challenge with variants of course material is that these variants might be used in different computing environments. In this case study, one version of the assignment ran on the instructor's computer and another ran on each student's computer. And once students submitted the assignment, their code would run on the autograder server, which had its own distinct Linux environment. In other words, instructors must not only ensure that variants of course materials stay consistent but also that the *computing environments for these variants stay consistent as well*. One way to cope with this problem is to have all students use a cloud-based coding environment. Some courses do this, but others give students the flexibility to work locally on their own computers.

Other examples.

Here are other times when we faced the challenge of maintaining consistent variants of course materials:

- Changing the dataset used for an assignment often requires rewriting many questions and accompanying autograder tests, then testing those tests to make sure they are robust.
- Creating alternate versions of programming assignments after noticing that the current solutions were posted online.
- Updating lecture and assignment code because a package version updated, for example `pandas` releasing version 1.0.

3.4.3 Challenge 3: Writing ad-hoc software infrastructure to manage scale

Another challenge of teaching large technical courses is that we find ourselves spending lots of time working as software developers and sometimes as software development managers, even though our primary job is supposed to be teaching. We have to either write lots of *software infrastructure* ourselves or supervise TAs who write the code to manage the complexity of course logistics. See Figure 3.2 for an example of some software infrastructure that we created.

Case Study: Handling Math and Programming Problems

After students take the introductory courses in our data science program, they then take a theory course on algorithms. When we first launched this course, students completed a weekly assignment in LaTeX. To make an assignment, we created a single LaTeX file that contained all the problems and solutions. Then, we used a library of LaTeX macros to strip out solutions in the student version of the assignment. This system was relatively simple, since we could create the student versions of the assignment using a single shell command. However, this course infrastructure software grew significantly more complex over time as we iterated on the course.

Why make a change? As is typical for new courses, the topics we taught in this course changed many times over its first few years. For instance, the course initially spent several weeks on statistical concepts. However, we later decided to move those topics to another course in the program and focus on algorithm analysis instead. Whenever we added a new topic or changed the order of existing topics, we also needed to update the course assignments to match the lecture schedule (as discussed in Section 3.4.1). The process of updating assignments was prone to introducing syntax errors or pedagogical bugs since we had to manually copy-paste LaTeX problem descriptions between files.

Also, after several course offerings we accumulated a growing "library" of assignment problems for every course topic. For instance, we wrote problems on big-O notation for each offering, so after a few years of running the course we had a number of possible problems to

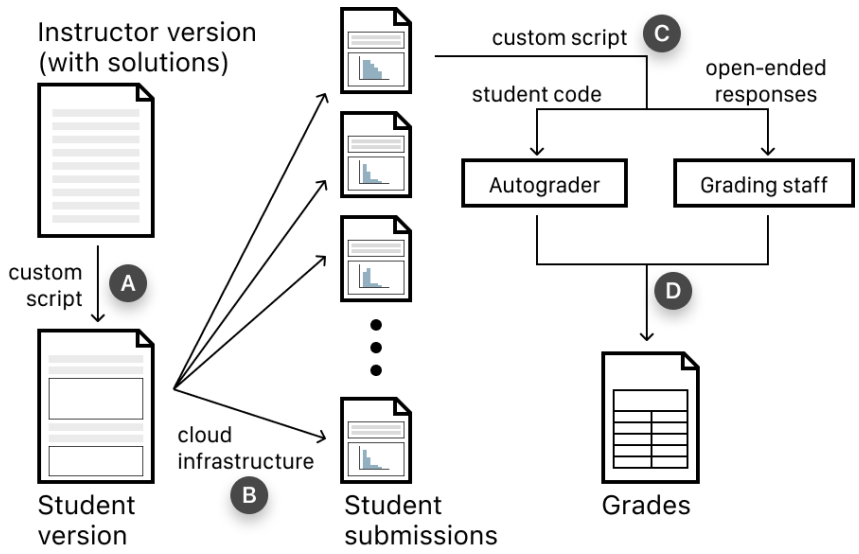


Figure 3.2. We wrote custom software infrastructure to manage assignment workflows. (A) Instructors use custom scripts to strip out solutions and test cases to generate a student-ready assignment file. (B) Students write and submit assignments to a cloud-hosted computing environment. (C) Another custom script runs an autograder on student code and splits out the open-ended responses for manual grading. (D) Grades are collected into a single score and released to students.

include in an assignment. We wanted a way to choose a few problems from our problem library for each assignment, but there was no existing tool available to do this task.

How did we update the course? Updating course topics meant that we also had to update the ad-hoc infrastructure code that we had written earlier to manage assignment creation, submission, and grading (summarized in Figure 3.2).

First, we refactored our existing assignments by moving every problem to its own file. We then wrote a script that could concatenate problem files together into a complete assignment. This script started as a simple set of shell commands but grew more complex as we augmented it to handle other tasks. For instance, we added a feature to automatically generate a blank LaTeX starter template for students to write in their solutions so that submissions had a consistent style to make them easier to grade.

We also wanted to give students programming questions in the course, which required a series of extensions to the existing infrastructure that managed our lecture and assignment

materials, which were based on Jupyter notebooks. We wrote scripts that could remove our solution code from Jupyter notebooks, concatenate notebooks together, bundle both code and LaTeX files together for students, and generate configuration settings for the course autograder server. At the time of writing, this course infrastructure consists of an ad-hoc mix of shell scripts, LaTeX macros, and Makefiles that manage LaTeX files, Jupyter notebooks, and Python scripts for assignments.

Reflection. Our software infrastructure plays an important role in helping us manage the scale of our courses—every task that the infrastructure currently handles used to be manual work for the course staff. For instance, we no longer worry about accidentally leaving solutions in the student versions of assignments since our scripts handle this automatically. This is especially useful for helping us manage the many variants of course materials (Section 3.4.2).

However, this software infrastructure itself requires constant maintenance and debugging since it is a complex piece of ad-hoc software created by educators, not professional software developers. Many courses in our program hire TAs who specifically monitor the course infrastructure so that problems can be quickly addressed.

This infrastructure can grow complex over time. For instance, Figure 3.2 depicts a typical part of our infrastructure for generating assignments, handling submissions, and autograding them on a server. This software not only needs to be written but must also be fine-tuned every term as assignment details get updated and the underlying run-time environment (e.g., operating systems, library versions, cloud hosting services) change year-to-year.

Other examples.

We have also used our infrastructure for:

- Extracting free-response questions from student submissions for manual grading.
- Hosting a cloud-based computing environment so that students all use the same Python and package versions.

- Handling grading special cases like late submission penalties.

3.4.4 Challenge 4: Cannot easily reuse software written by others

The prior section described how we spend significant amounts of time writing and maintaining custom software infrastructure for our courses. A reasonable question is: *Why don't we purchase off-the-shelf software or use freely-available software written by others?*

First, some of us do use a generic LMS (learning management system) to keep track of logistics like student enrollment lists and gradesheets. But such software is best-suited for static content like posting PDFs of homework assignments. Technical courses like the ones we teach have lots of *dynamic* content such as coding assignments that need autograding infrastructure and different variants of course materials that interleave code, data, and exposition (see Section 3.4.2). Generic LMS software such as Blackboard, Canvas, or Moodle cannot handle that kind of dynamic content on their own.

We also use free tools such as GitHub and software development toolchains like the classic Unix 'make' [117] for programmatically compiling different variants of course materials. However, we still have to write custom code to wrap around these tools since they were not originally developed with educational use cases in mind. And even though we use tools designed for educational use like Gradescope [232], we still need custom code since each tool uses differently formatted data.

In sum, we have found it challenging to reuse instructional software written by others, as illustrated by the following case study.

Case Study: Handling Software Tool Updates

In several of our courses we rely on an open-source software package for autograding student code in Jupyter notebooks called Otter Grader [61]. This tool was specifically made for educational use, so it has several convenient features. For instance, it can automatically generate test cases from `assert` statements in Jupyter notebooks. However, using this package comes

with its own maintenance requirements—when we update this package to use the latest release, it often requires us to also update our course infrastructure and materials.

Why make a change? Since Otter Grader receives regular bug fixes and updates, we prefer to use the latest package version when possible. But these updates also come with their own backwards-incompatible changes. In one notable instance, an old package version could take over 45 minutes to generate student variants of assignments; the new package version fixed this bug so it ran much faster. But when we updated to the new version, we found that other parts of our infrastructure code stopped working since our code depended on old functionality that was no longer available.

How did we update our course? To make our infrastructure code work again, we needed to edit files in every single assignment for the course. We also needed to teach the course staff how to use the changed infrastructure since we updated our existing scripts in the process of updating to the latest Otter Grader package version.

As another example, we added special markup to every assignment that tells Otter Grader what parts of the assignment to convert into test cases. However, in an upcoming package version the markup format will change in a backwards-incompatible way. Thus, when new versions are released and we update our infrastructure, we expect that we will need to manually go through every single assignment file and change the markup to match.

Reflection. This case study highlights the challenges of relying on external software tools when running a course. Even software tools that are specifically designed for instructional use require manual effort to update. Since we lack a single tool that handles every single instructional use case, we cobble multiple tools together in our course software infrastructures. For instance, we use Otter Grader alongside scripts that we wrote ourselves, and student code is graded in another third-party tool called Gradescope that also needs to be configured and hooked up to our other software.

In theory, we could have designed a one-size-fits-all infrastructure that works for all

the courses in our data science program, since presumably programming-based courses involve similar sorts of workflows comprised of code submission, autograding, and assignment file variants. But despite these surface-level similarities, in our experience it was still more effective to develop our own software that could be customized for the needs of each course rather than reuse another course's software infrastructure.

In fact, several of us have actually tried to reuse another course's software, only to encounter mismatches that were time-consuming to fix—for instance, a course infrastructure for Jupyter notebooks might not be able to handle LaTeX files. When these moments arose, we did not want to ask other instructors to spend even more time adding features to their custom infrastructure since it would unfairly take time away from their course.

Also, making software generally reusable by others requires significant time investment beyond simply making the software functional—for instance, software without good documentation is hard for others to use. Since our primary jobs are to teach, not to write and maintain software, our code ends up being highly specific to our course and is not designed for others.

Finally, writing software ourselves makes it much easier to adjust it on-the-fly in response to surprises that arise throughout the term. Many of us value flexibility in our software since our courses change frequently. And when the course's TA staff has expertise in their own software it is much more feasible to fix bugs quickly, which we value especially during the run-up to assignment deadlines.

Other examples.

Here are other cases where we encountered difficulties in reusing software written by others:

- Running into merge conflicts when using the Git version control tool, which then prevent Jupyter notebooks from loading.
- Needing to write scripts to transfer grades between multiple grading systems, like the

course autograder, manually-graded exams, and our university's gradebook.

3.5 Discussion

Our case studies shed light on the fact that running, maintaining, and updating large-scale technical courses involve a tremendous amount of invisible infrastructural work [124] that is not directly related to pedagogy. In light of these findings, we now discuss how to design better tools to support instructors in coping with such complexities.

3.5.1 Instruction at scale lacks the tools that make open-source software projects successful

To reflect on our findings as a whole, we draw an analogy between instruction at scale and open-source software development [115]. In both, multiple stakeholders collaborate to produce a useful product that is shared with a large audience.

Successful open-source projects gain more users and maintainers but must also deal with a growing code base that fills with technical debt [173]. This progression mirrors our experience as instructors since our courses are rapidly-growing not only in numbers of students but also in numbers of TAs and amount of course material that we need to maintain. Open-source software projects also face the challenges that we as instructors encounter with intricate dependencies, infrastructure, and distribution of knowledge. Like instruction at scale, large software projects depend on other libraries that are frequently updated. A single set of source code may generate multiple output variants, for instance to build executables for different operating systems. They also rely on ad-hoc infrastructure via custom build scripts and Makefiles. And complex software contains too many features for one person to track in detail, so project knowledge is distributed among multiple maintainers.

People who maintain open-source software are all-too-familiar with the problem of cascading updates due to chains of dependencies [103], which we also faced when updating our course materials. But instructors lack analogous tools that software projects rely on. For

instance, package managers [249] enable automatic dependency tracking—a developer might update a dependency by changing a single metadata file that tracks package versions, running its test cases, and then using code inspection tools to ensure the project still functions as expected. However, these off-the-shelf tools do not account for all the different settings where instructors might use code, like as screenshots embedded within lecture slides or as snippets inserted in LaTeX or MS Word assignment files. More broadly, instructors lack an way to automatically track semantic dependencies based on pedagogical concepts.

3.5.2 Toward instructor-centered tool design

We now synthesize our findings into ideas for future tools to address the logistical challenges of instruction at scale. As a whole, we advocate for ***instructor-centered tool design***: rather than force instructors to adopt entirely new systems and workflows, tools should acknowledge that courses already have existing workflows, that teaching staff changes frequently from term to term, and that instructors desire flexibility in handling course updates.

As a representative example, consider the dependency problem again from Section 3.4.1. For instance, an instructor who removes a lecture slide introducing *pivot tables* [250] needs to ensure that no future course content relies on pivot tables. These dependencies extend beyond just keeping software package versions up-to-date; thus, they are not supported by current package manager tools. A future tool might address this by explicitly representing the dependencies between pieces of content in a course. Such a tool could represent dependencies in a fine-grained way, so that removing a lecture slide on pivot tables would alert the instructor to the specific homework questions, discussion worksheets, and exam questions that also need updating. This will make it easier to keep multiple pieces of course content in sync.

However, it is unrealistic to expect that instructors will explicitly set aside time to record every single content dependency, for the same reasons that they currently do not set aside time to document their course updates. To take this into account, future tools could embed themselves directly within instructor workflows without requiring them to switch applications to record

dependencies. For instance, we noticed that we often open multiple pieces of dependent material in separate windows to manually ensure that materials remained consistent. A tool for automatic dependency tracking might observe this activity at the operating-system level [197] and provide a prompt to allow the instructor to mark these materials as linked together, or even automatically record the links based on how often two files were opened side-by-side.

Relatedly, instructors do not wish to break out of their workflows in the process of making course updates and thus do not often use outside tools to document their changes. They also hesitate to create yet another piece of content with metadata that they must keep up-to-date. In software development, maintainers often rely on email threads, chat channels (e.g., Slack), and GitHub issues/commit messages rather than monolithic documentation pages. Can future tools support similar kinds of lightweight contextual documentation for instructors who work in multiple applications and diverse file formats? Such a tool could record down useful context that the instructor provides tacitly [97] as they are working.

3.6 Conclusion

We used four case studies to present some of the complexities of updating large-scale courses related to data science and programming. These challenges include managing intricate dependencies between course materials, content variants, software infrastructure, and software reuse. The main implication of our findings is that large technical courses are like complex engineering systems with hundreds of 'moving parts' that depend on one another in subtle ways. Similar to how programmers have developed a rigorous methodology of software engineering over the past few decades, we call for the research community to move toward an analogous discipline of *courseware engineering* [162]. This would involve not only better design practices but also building domain-specific tools to help manage the logistical complexities of maintaining and updating courses at scale. These efforts will hopefully free up instructor time and energy to focus on what matters most: *teaching students.*

Chapter 3, in full, is a reprint of the material as it appears in the proceedings of the ACM Conference on Learning at Scale as The Challenges of Evolving Technical Courses at Scale: Four Case Studies of Updating Large Data Science Courses. Sam Lau, Justin Eldridge, Shannon Ellis, Aaron Fraenkel, Marina Langlois, Suraj Rampure, Janine Tiefenbruck, Philip Guo. 2022. The dissertation author was the primary investigator and author of this paper.

Chapter 4

How Computer Science and Statistics Instructors Approach Data Science Pedagogy Differently: Three Case Studies

Over the past decade, data science courses have been growing more popular across university campuses. These courses often involve a mix of programming and statistics and are taught by instructors from diverse fields ranging from computer science to other STEM disciplines. I and two other instructors collaborated to help launch a data science program at a large public U.S. university over the past four years. We noticed one central tension within many such courses: instructors must finely balance how much computing versus statistics to teach in the limited available time. In this study, we provide a detailed firsthand reflection on how we have personally balanced these two major topic areas within several offerings of a large introductory data science course that we taught and wrote an accompanying textbook for; our course has served several thousand students over the past four years. We present three case studies from our experiences to illustrate how computer science and statistics instructors approach data science differently on topics ranging from algorithmic depth to modeling to data acquisition. We then draw connections to deeper tradeoffs in data science to help guide instructors who design interdisciplinary courses. We conclude by suggesting ways that instructors can incorporate both computer science and statistics perspectives to improve data science teaching.

4.1 Introduction

It is widely acknowledged that both computer science and statistics are fundamental to data science [98]. The computer science perspective generally focuses on "how"—how to do data analysis in practical real-world settings. Data scientists write programs using languages like Python and use software packages like `pandas` [189] to load, manipulate, and clean data. Computer science also provides tools to analyze the runtime and memory usage of code. These tools have led to the development of systems for working with large datasets, including relational databases and distributed computing (e.g. MapReduce, Hadoop, Spark). Finally, data scientists rely on algorithms for numeric optimization to fit their models. Without computer science, data scientists are limited to small datasets and cannot make use of the vast amounts of data being generated today.

On the other hand, the statistics perspective generally focuses on "why"—for instance, why an analysis on a relatively small sample can generalize to a larger population. Statistics provides fundamental tools for inference like hypothesis testing and confidence intervals. These statistical tools let data scientists say precisely *why* they have (or don't have) confidence in the results of an analysis. And statistical learning gives basis to the models that data scientists use to make predictions. Without statistics, data scientists can only draw conclusions about their samples—they can't use their samples to make inferences or predictions about the unobserved population.

Domain experts acknowledge that data science courses should demonstrate how both computer science and statistics can be applied to do useful data analyses [99]. However, data science instructors then face the practical challenge of providing students with an effective balance of these two fields, given the limited time available in a quarter- or semester-long course. In practice, different instructors strike different balances between computing and statistics content. This tension is reflected in data science programs as a whole as well—some programs have more of an emphasis on computing, and others have more statistics, as discussed in past panels and

reports [116, 216, 223, 218, 75, 67, 65, 137].

This study is, to our knowledge, the first to reflect on why and how computer science and statistics instructors approach teaching data science differently. The authors of this study come from both computer science and statistics departments at a large public U.S. university. We have worked together over the past four years to design curriculum and write a textbook for an undergraduate data science course that serves over 2,500 students a year. This paper presents three case studies of where we made critical pedagogical changes along the way.

On the surface, these discussions were about changes to a specific course. However, we found that our negotiations actually revealed fundamentally different approaches to data science. Each case study explains how our approaches differed, and how we eventually integrated both perspectives. In the first case study, we discuss programming abstractions—should our students implement gradient descent themselves or use a Python package? In the second, we discuss how to approach statistical modeling for different levels of mathematical maturity. In the third, we discuss the tension between working with clean scientific data versus messy data in the wild.

We reflect on our case studies as a whole to show how the different approaches of computer science and statistics also mirror deeper tradeoffs in data science education: the tradeoff between emphasizing inference and prediction, and the tradeoff between theory and practice. We conclude by making suggestions for data science instructors who bridge the gap between the two disciplines. This study's contributions to computing education are:

- Three case studies from our experiences that highlight the different approaches that computer science and statistics instructors take to teaching data science.
- A discussion of fundamental tradeoffs for data science pedagogy, and suggestions for other topics that can benefit from a close coupling of computer science and statistics.

4.2 Setting: A Large Undergraduate Data Science Course and Textbook

Three authors of this study helped design and instruct an upper-division undergraduate data science course at a large public U.S. university; two authors are from computer science and one is from statistics. This course was launched in Spring 2017. It is now a requirement for both data science majors and minors at our institution, serving over 2,500 students during the 2020–2021 academic year. In this course, students learn standard computational tools for working with data, including relational databases, Python-based programming tools (e.g., `pandas` [189], `scikit-learn` [203]), and algorithmic techniques such as gradient descent. Students also learn statistical techniques for modeling, including statistical techniques for visualization, regression, and bias-variance tradeoff. Our course prepares students for real-world analyses by showing how *both computer science and statistics* are used throughout data science.

Since our course includes topics from both computer science and statistics, it is co-taught each term by one instructor from the computer science department and one from the statistics department. We are not the only instructors involved with the course; other faculty from our departments have also instructed and made helpful contributions to the curriculum. Although we do not speak for all the instructors of the course, we have directly instructed in seven out of the eleven offerings so far. We are also the authors of a textbook used in this course [178].

In the past four years of working together, we have made many changes to the course and its accompanying textbook. Most of these changes needed discussion amongst this study's co-authors before implementing. In the following sections, we share three case studies where these discussions highlighted how our backgrounds approach data science differently, and how we reached agreement by acknowledging the perspectives of colleagues in different fields.

Caveats: We acknowledge that other institutions have different arrangements for teaching. For instance, some courses alternate instructors between computer science and statistics, and data science programs have students take separate courses from both computer science and statistics.

To provide experiences that remain relevant across different teaching arrangements, in this study we focus on more foundational data science concepts rather than course-specific logistical details. Although our case studies happened to take place in one course, we expect that they provide insight for instructors who must strike their own balance between computer science and statistics content when designing and iterating on their courses.

4.3 Three Case Studies

This section reports on three case studies from our course and textbook design experiences. For each case study, we summarize what actually happened by presenting a back-and-forth 'simulated conversation' between a computer science (COMPSCI) instructor and a statistics (STAT) instructor. These conversations are meant to capture where each side started from, how each presented the benefits and tradeoffs of their pedagogical approach, and eventually how we came to a resolution[1]. Since our reported conversations are 'simulated', they do not directly quote individuals, although we consulted our old meeting notes where possible. Instead they are meant to capture the high-level points-of-view that emerged from us jointly reflecting on actual pedagogical negotiations we had with each other throughout the past four years of working together.

4.3.1 Do We Need to Delve into Algorithms?

One of the most foundational algorithms in data science is *gradient descent*, which is used to fit all kinds of models ranging from simple (e.g., linear regression) to complex (e.g., neural networks). The gradient descent algorithm can be taught at multiple levels of abstraction. But, which level of abstraction is best for teaching data science? Or, do we even need to teach it at all? We tried several approaches over different iterations of our course and its textbook, summarized in Figure 4.1.

[1]This explanatory approach of a simulated dialogue is inspired by business school case studies that show how stakeholders discuss and make business decisions [226].

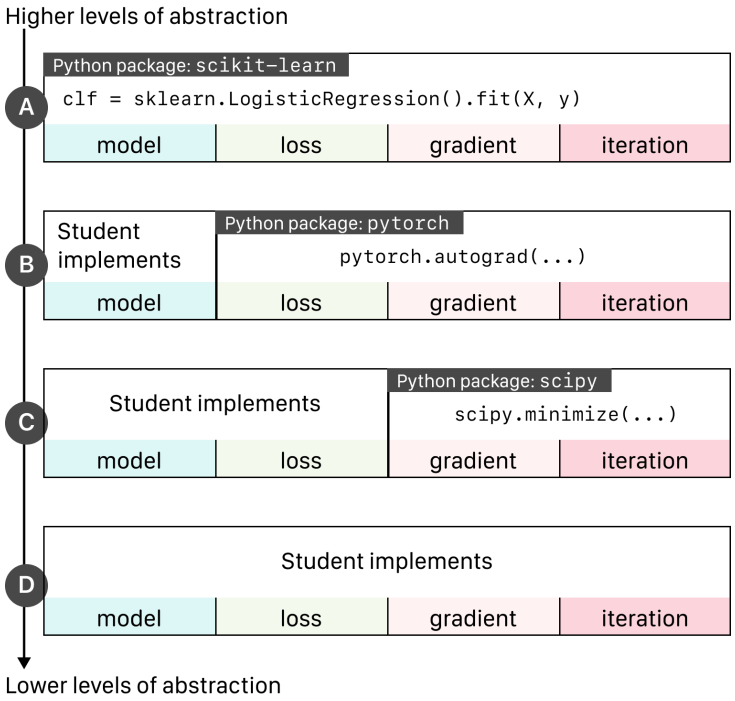


Figure 4.1. We considered multiple ways to teach students to fit models using Python. Each approach works at a different level of abstraction, and our internal debate centered around how much implementation students should do themselves.

Starting Points:

STAT instructor: The main goal of our course is to teach students how to do sound work with data. It's key that our students learn the importance of using well-tested and well-implemented code when they optimize and fit models. The Python `scikit-learn` package implements all the models we teach, and our students will likely only use the `fit()` method from `scikit-learn` once they start working on real data problems (Figure 4.1a). Teaching them how to implement models and gradient descent *from scratch* (Figure 4.1d) is not necessary since packages are well-tested and optimized.

COMPSCI instructor: Well, I think numeric optimization is an important topic in data science. Even students who only use packages like `scikit-learn` will still need to understand gradient descent in order to fully use its API. There are interesting teaching possibilities here that connect the theory of optimization with its implementation! For example, we can use automatic differentiation software like the `autograd` module from `pytorch` (Figure 4.1b). Or, we could

have students calculate and implement their own loss functions, then pass those into gradient descent optimizers like `scipy.minimize` (Figure 4.1c).

Discussion:

STAT instructor: Teaching students to implement gradient descent seems great for a specialized course on numeric optimization. But it's less clear how implementing gradient descent from scratch helps them understand and draw conclusions from real-world data. How do you see it fitting into the larger data science picture?

COMPSCI instructor: Optimization is an important topic in data science because it lets us determine what models are feasible to use given real-world constraints on time and memory usage. For instance, neural networks could only be applied to realistic problems, like image recognition, after we had more powerful computers and better gradient descent algorithms. Implementing gradient descent lets us teach students about the runtime and memory needed to fit models. For example, we can have students implement both batch and stochastic gradient descent, ask them to fit models using both algorithms, and then have them explain why one might converge faster than the other. It's important that our students can say, *"I can still fit this model if the training data doubles in size, but I'll need to change my approach if the training data is ten times larger."*

STAT: Ok, I agree that runtime is an important tradeoff for data scientists to consider. Runtime is especially important for large, complex models like neural networks. But, teaching gradient descent at levels of abstraction lower than `scikit-learn` (Figure 4.1a) has drawbacks, especially if students need to learn other complex Python packages to do so. We've seen that students already find it hard to learn a package specifically for gradient descent—although they wrote fewer lines of code, each package has specific idioms that they had to learn. That's a lot of cognitive overhead! It's useful to teach students how to work with multiple Python packages, but we only have so much time in one course.

COMPSCI: Point taken. Though there's another aspect to what I mentioned earlier: to

fully use the `scikit-learn` API, our students will need to have a deep understanding of gradient descent. For instance, `scikit-learn` models let data scientists change the stopping criteria and the maximum number of iterations, and they need to experiment with these parameters every time they try to fit a model. I want our students to identify when they need to change these parameters and what the tradeoffs are—for example, if their model doesn't converge, they should know they can increase the cap on the number of iterations if they're willing to run for longer.

STAT: I agree that's important, but can't we just teach the gradient descent algorithm using pseudocode to get those concepts across?

COMPSCI: I think students develop better intuition about gradient descent when they implement and debug a basic version of it themselves. For example, they can add custom logging statements into their own implementations to look at their program's behavior at a fine-grained level. `scikit-learn` also implements other optimization algorithms, including second-order methods like BFGS [184]. Even though we don't have time in our course to explain these algorithms in detail, implementing gradient descent from scratch gives students a basic framework to start understanding when other algorithms can be useful.

Resolution:

Our discussion eventually converged, and we agreed that gradient descent was important for our students to understand well. Data scientists make many tradeoffs when choosing and fitting models—they consider how much data they have, whether the model might underfit or overfit the data, and how accurate and interpretable the model is. At each step in modeling, data scientists must also think about the computational resources that they have access to. We found gradient descent to be a natural place to introduce these central considerations in modeling.

Thus, we resolved to take a hybrid approach. We teach students to use `scikit-learn` (Figure 4.1a) but also teach them to implement a basic version of gradient descent completely from scratch in Python (Figure 4.1d). Since our students have some programming background, we found they could implement basic versions of gradient descent after one lecture's worth of

explanation through pseudocode, which we felt was reasonable for our course's time constraints. We also used this hybrid approach in our textbook, and took care to point out that the book contains simple implementations for teaching purposes only, not for production use.

4.3.2 How to Approach Statistical Modeling?

Data scientists use statistical models to draw inferences and make predictions using data. In this case study, we discuss various approaches for teaching the foundations of modeling.

Starting Points:

STAT: The likelihood principle offers a unifying approach to modeling that extends to many settings. In a traditional statistics course, we define a simple linear model as $Y = \theta_0 + \theta_1 x + \varepsilon$, where the errors, ε, are independent normally distributed with mean 0 and constant variance. From this perspective, fitting a model means finding the model parameters that maximize the likelihood of observing the data, i.e. maximum likelihood estimation (MLE) (Figure 4.2a).

When we work through MLE derivations, we motivate different loss functions. For instance, the simple linear model described above naturally connects to squared loss (Figure 4.2b). And, at the end of the derivation, we can get closed-form solutions for model coefficients, which have meaningful interpretations. For instance, the 'regression effect' clearly appears in the slope of the regression line—the sample correlation scaled by the standard deviations of x and y (Figure 4.2c). The beauty of the likelihood principle is that it shows how a data-generating model leads to both loss functions and parameter estimates. We can also use this approach to make inferences for model parameters.

COMPSCI: OK, but my focus is on prediction, not inference—I'm not as concerned about the exact form of the model and its parameters compared to getting a predictor with high accuracy. Although the likelihood principle leads to useful interpretations of model parameters, I often have so many parameters in my models that there isn't a useful interpretation for the parameters anyway. An over-parameterized model isn't a big concern for predictive accuracy

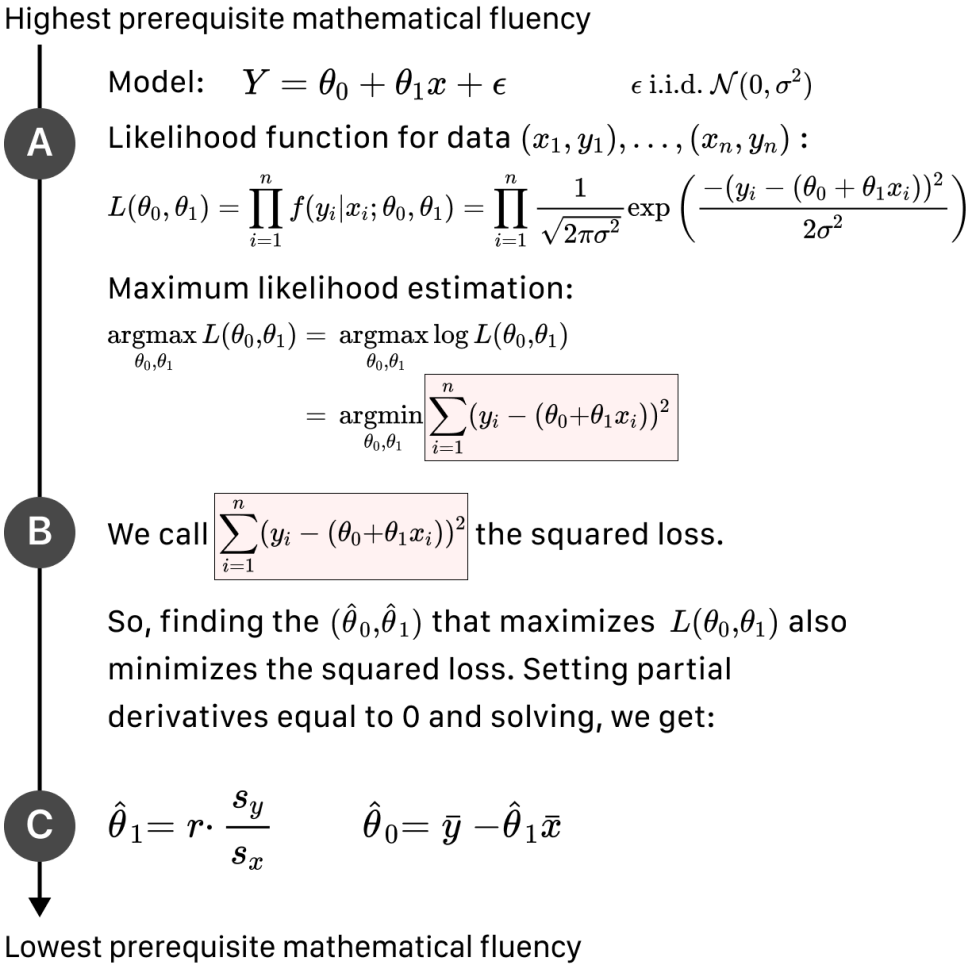


Figure 4.2. This sketch of a derivation for the simple linear model parameters highlights several possible pedagogical approaches. Instructors can start with (a) likelihood principle, (b) loss minimization, or (c) a closed-form solution.

since we can control over-fitting through regularization. Moreover, since we can fit models using loss minimization via gradient descent, we don't need to find closed-form solutions for the MLE. And, the likelihood principle for widely used models like logistic regression gets complicated quickly. The advantage of using the MLE in this case is not clear.

Discussion:

COMPSCI: When students go into the world, it's unlikely that they will need to consider the likelihood principle when they fit models. Most models we use in this course have closed-form MLE solutions because they are relatively simple, but more complicated models like neural networks don't have nice analytic solutions. When faced with more complex problems, students might get easily stumped because they can't cast the problem in terms of likelihood.

STAT: MLE shows explicitly how a data scientist's assumptions affect how we fit the model. Important extensions are well-motivated by this approach, such as weighted regression and the generalized linear model. The likelihood approach I'm proposing will require us to state model assumptions upfront (Figure 4.2a). And when assumptions are violated, the likelihood approach can provide a path forward. Without MLE, I'm concerned the course sends the message that data scientists can pick any arbitrary model and loss function they want without considering the assumptions behind them.

COMPSCI: I agree that it's important for students to understand whether a model is appropriate for their data. However, to understand MLE, students need to draw on knowledge of random variables, expectation, probability distributions, and calculus. We've tried to teach MLE before, but students felt unprepared, and they fixated on the math rather than the underlying concepts we really wanted to teach. In hindsight I think we underestimated how hard it is to develop fluency with probability.

STAT: I agree the technical machinery used with MLE can become a burden for students, especially when they first see multivariate modeling. But even if we don't cover MLE in this course, it's important to help students build mathematical maturity. The stepping stones to

MLE—expectation, variance, and bias—are useful when we teach general modeling principles. For example, in our course we make a distinction between empirical loss (or training set average error) and expected loss (based on model error). When we teach that a model's test set error is a statistical estimator for the expected loss, we reinforce statistical ideas that they can use later.

Resolution:

We converged to agree that the likelihood principle is an important concept in modeling. However, we could only realistically cover modeling from a loss minimization approach based on the time constraints of our course and prerequisite student knowledge (Figure 4.2b). Thus we postponed the likelihood principle and MLE to a more advanced theory course. Then in our course and its textbook, we included the conceptual stepping stones to the likelihood principle by teaching model fitting through empirical loss minimization.

We now teach a broad array of loss functions—e.g., squared, absolute, and cross-entropy loss—but do not show their derivations from a likelihood standpoint. We also encourage students to consider how prediction errors should be penalized based on domain knowledge. For instance, if over-medication in surgery can lead to serious health problems, then an asymmetric loss function that penalizes overestimates more than underestimates is preferable [192]. By taking the middle ground and focusing on loss and optimization, we give a balanced perspective that covers both inference for model parameters and prediction of future observations.

4.3.3 How to Approach Real-World Data?

It takes a lot of work to find useful data for teaching, so we had many discussions on what data to use in our course and textbook.

Starting Points:

STAT: Data analyses begin by asking how the data were collected. One important distinction is whether or not the data were collected according to a chance mechanism. With probability sampling, we can teach the notion of *representative data*. For instance, with a

simple random sample (SRS) we know that every potential sample has the same chance of being selected. More broadly, in the real world, alternative probability methods (e.g., stratified, cluster, systematic sampling) are used in scientific surveys. If we ask students to carry out probability calculations for small populations, then they can better understand the importance of random samples.

COMPSCI: These sampling methods are a nice-to-have rather than a necessity. The SRS is all we need to get the point across to students. I'm not sure why we need to discuss multiple random sampling methods when nearly all of the datasets in the course are *not* collected using random sampling. For example, we work with datasets of social media posts (e.g., tweets), housing prices, and emails. All of these are examples of *found data*: data scientists usually "find" data that are already available online rather than collecting data themselves. And none of these are probability samples. This also reflects a current trend in data analysis: Traditional scientists start with a research question and then collect data *specifically* to answer that question. Today, data are more plentiful and accessible, so data scientists often start with available *found data* and explore interesting questions that data might be able to answer.

Discussion:

COMPSCI: Probability calculations with various sampling schemes can get complex real fast, and I don't see the point of teaching counting arguments and combinatorics. It's also easy to spend too much time on fun probability problems with dice and cards. While this develops intuition for probability, it can get to be too much theory and feels very disconnected from real-world data analysis problems.

STAT: I agree it's tempting to teach too many fun probability problems and that counting arguments have little value in data analysis. But, sampling methods *do* have important real-world implications, especially in recent times: for instance, treating margins of error in election polling as if the data were from a SRS winds up under-estimating the true margin of error [230]. And more importantly, for nonrandom samples it doesn't make sense to compute p-values or

confidence intervals—these methods assume random draws from a population, but *found data* break that assumption.

COMPSCI: Yes, ignoring the sample design can lead to major mistakes. There are many famous examples where ignoring this led to major problems with the analysis, like the wrong call in the Dewey-Truman election [58]. Still, even the most careful surveys aren't true probability samples because of issues like non-response [102]. Should we teach students random samples when in reality there are many caveats? Few samples are truly random in practice.

STAT: That is indeed the case, but the ideal scenario is worth knowing about and it gives students a useful comparison point against the data they do have. And, there are statistical methods for adjusting for, say, non-response [246]. Even if we don't have time in this course to talk about specific methods, data scientists should know that these methods exist. More importantly, data scientists need to be aware of common ways that their samples may not be representative. That way, when they do assume that their sample is representative, they should be able to justify it.

COMPSCI: I do think what you are saying is important; our students should acknowledge when they make these assumptions and point out potential sources of bias for their analyses. But, I want to return to an earlier point because I think it is an important one. So much of today's data are large administrative data, such as digital traces. It doesn't make sense to ignore these data sources.

STAT: I agree we need to expand beyond an idealized view. But we also want our students to avoid "big data hubris" [180]—thinking that having more data automatically leads to sound analyses. They need to leverage big data where it makes sense. They also need to understand how to analyze scientifically-collected data. We shouldn't only teach one or the other, because they will use both on the job.

Resolution:

Although our starting points were about sampling methods, the discussion quickly went

to the core of how to treat representative and non-representative data. We decided to reduce the emphasis on specific sampling methods, but kept the stratified sample as a comparison to the SRS (simple random sample). The main change we made to the course and textbook was to develop a framework to help data scientists reason about whether or not their data are representative. We extended the traditional way of teaching survey sampling to introduce the target population, access frame, and sample. This framework gives us the tools to discuss accuracy, including various sources of bias (e.g., coverage, selection, non-response), and has a natural extension for introducing measurement error. The framework also allows us to situate and link together different kinds of data, including survey samples, controlled experiments, administrative data, and instrumental measurements.

4.4 Discussion

In this section, we draw common themes from our three case studies, focusing on the places where computer science and statistics approach data science differently. These case studies show how these two academic disciplines developed with distinct goals. The difference in goals reveals pedagogical tradeoffs that instructors make when designing data science courses. We discuss these tradeoffs and suggest ways that instructors might acknowledge and incorporate both perspectives into data science teaching.

4.4.1 Inference and Prediction

A central tradeoff in data science pedagogy is balancing the emphasis on inference with emphasis on prediction. This is illustrated in the second case study of this chapter (Section 4.3.2). In our experience, statistics focuses more on inference while computer science focuses more on prediction. For instance, a data scientist can construct confidence intervals to infer the slope of a regression line in a model. In contrast, when data scientists focus on prediction, their goal is to find a model with high predictive accuracy. This tradeoff is analogous to the contrast between the data modeling and algorithmic modeling cultures in statistics [86]. An overly extreme focus

on inference might only allow relatively simple models. On the other hand, an overly extreme focus on prediction might select models without regard for interpretability or robustness.

The experiences from our case studies have led us to the view that teaching both inference and prediction together helps students appreciate the difference and learn when to emphasize one over the other. For instance, one well-known economic study uses an inferential view to determine which social factors (e.g., segregation, income inequality) are significantly correlated with economic mobility [96]. Another study uses a predictive view to estimate income from phone call records—the significance of each covariate is less relevant than the model's overall predictive ability [78]. We recommend that data science instructors bridge the gap between traditional computer science and statistics views by including both predictive and inferential views for statistical modeling.

4.4.2 Theory and Practice

Both computer science and statistics contribute methods and theory for data science. For instance, computer science teaches practical software tools and theoretical analysis of runtimes. Statistics teaches practical modeling techniques and bias-variance theory that applies across models. When we first launched our data science course, each of our lessons tended to focus heavily on either theory or practice. This divide sparked pedagogical discussions amongst this study's co-authors as we co-developed our course and textbook. For instance, in our third case study (Section 4.3.3) we debated how much to emphasize the theory for random sampling versus the practical reality of using found data in-the-wild. We moved individual lessons towards a closer balance of both theory and practice as we discussed and iterated on our course. We expect that other instructors can anticipate similar discussions based on the fact that theory and practice both have clear roles in data science—for instance, a lesson that strictly focuses on one may be improved by incorporating a bit more of the other perspective.

4.4.3 Both Perspectives in Data Science Pedagogy

In our view, integrating both computer science and statistics perspectives improves our curricula. Although our three case studies cover three specific examples, we constantly discuss how we might better balance these two disciplines across multiple courses.

For example, data scientists often work with tabular data using the concept of a dataframe. The dataframe was originally designed by statisticians for exploratory data analysis but also serves as a computational data structure. Instructors can contrast the dataframe with its database analog, the *relation* (e.g. [208]).

Relatedly, there is a distinction between the term *data type* as used in programming versus in statistics. For instance, survey responses can be recorded using numbers (e.g., 0 for "no", 1 for "yes", and 2 for "maybe") or as strings. A statistician would describe this data abstractly as categorical, and would not compute the average even when stored as numbers. Instructors can discuss how computer science and statistics view data types differently.

As another example, in data visualization, statisticians and psychologists have established principles for scale, color, and shape [241], but a scatter plot with too many data points presents challenges with over-plotting [106]. Here, instructors can discuss tradeoffs between techniques like sampling and smoothing [213].

4.5 Conclusion

This study shares three case studies from our time spent working closely together on a data science course and textbook. The recurring themes we present highlight the different approaches that computer science and statistics take in teaching data science. These differences are captured in tradeoffs between explaining the "how" and "why" of each concept and reflect ongoing debates amongst experts in the field. We advise data science instructors to seek interdisciplinary views but also understand that it's difficult to please everyone—rather than searching for an immediately "perfect" balance of computing and statistics, instructors can instead welcome

different points of view and improve their courses over time.

Chapter 4, in full, is a reprint of the material as it appears in the proceedings of the ACM Technical Symposium on Computer Science Education (SIGCSE) as How Computer Science and Statistics Instructors Approach Data Science Pedagogy Differently: Three Case Studies. Sam Lau, Deborah Nolan, Joseph Gonzalez, Philip Guo. 2022. The dissertation author was the primary investigator and author of this paper.

Chapter 5

The Design Space of Computational Notebooks: An Analysis of 60 Systems in Academia and Industry

Computational notebooks such as Jupyter are now used by millions of data scientists, machine learning engineers, and computational researchers to do exploratory and end-user programming. In recent years, dozens of different notebook systems have been developed across academia and industry. However, we still lack an understanding of how their individual designs relate to one another and what their tradeoffs are. To provide a holistic view of this rapidly-emerging landscape, we performed, to our knowledge, the first comprehensive design analysis of dozens of notebook systems. We analyzed 60 notebooks (16 academic papers, 29 industry products, and 15 experimental/R&D projects) and formulated a design space that succinctly captures variations in system features. Our design space covers 10 dimensions that include diverse ways of importing data, editing code and prose, running code, and publishing notebook outputs. We conclude by suggesting ways for researchers to push future projects beyond the current bounds of this space.

5.1 Introduction

Over the past decade, computational notebooks such as Jupyter [205] have become popular programming environments for data scientists, machine learning engineers, computational

researchers, and business analysts [242]. As modern embodiments of Knuth's *literate programming* vision [164], notebooks enable programmers to rapidly prototype and share explorations by interweaving expository prose, executable code, and rich program outputs (e.g., data tables, images, and interactive widgets). In addition, modern notebook systems are situated within complex end-to-end workflows that involve data ingest, computing environments, collaboration, and distribution.

Computational notebooks embody several themes that are of interest to the VL/HCC community: end-user programming [166], exploratory programming [153], live programming [239], and literate computing [121]. Both academic researchers [94, 156, 222, 244] and industry practitioners [127, 20] have studied how people use notebooks in their work. These findings have informed the design of new systems that reduce common user frustrations: For instance, Verdant [150, 152] helps users manage the abundance of code and output versions created within notebooks, and Stitch Fix's Nodebook [34] eliminates out-of-order cell execution to improve reproducibility of results.

At present, dozens of such notebook systems have been developed in both academia and industry, but most are one-off-designs meant to address a specific user need or, for academic projects, to prototype a novel user interaction technique. Despite this recent proliferation of notebook systems and the gradual maturing of this field, we still lack a systematic understanding of how these dozens of individual designs relate to one another and what their tradeoffs are. To provide this holistic overview, we developed the first comprehensive design space of computational notebooks by analyzing dozens of notebook systems across academia and industry.

In HCI research, a *design space* succinctly captures multiple dimensions of variation in possible system features within a given domain; each individual system covers a specific range in its design space. Researchers have mapped out design spaces for systems in domains such as end-user programming [136, 84], information visualization [228, 144], and tangible user interfaces [120]. In this study, we extend this analysis technique to the domain of computational notebooks.

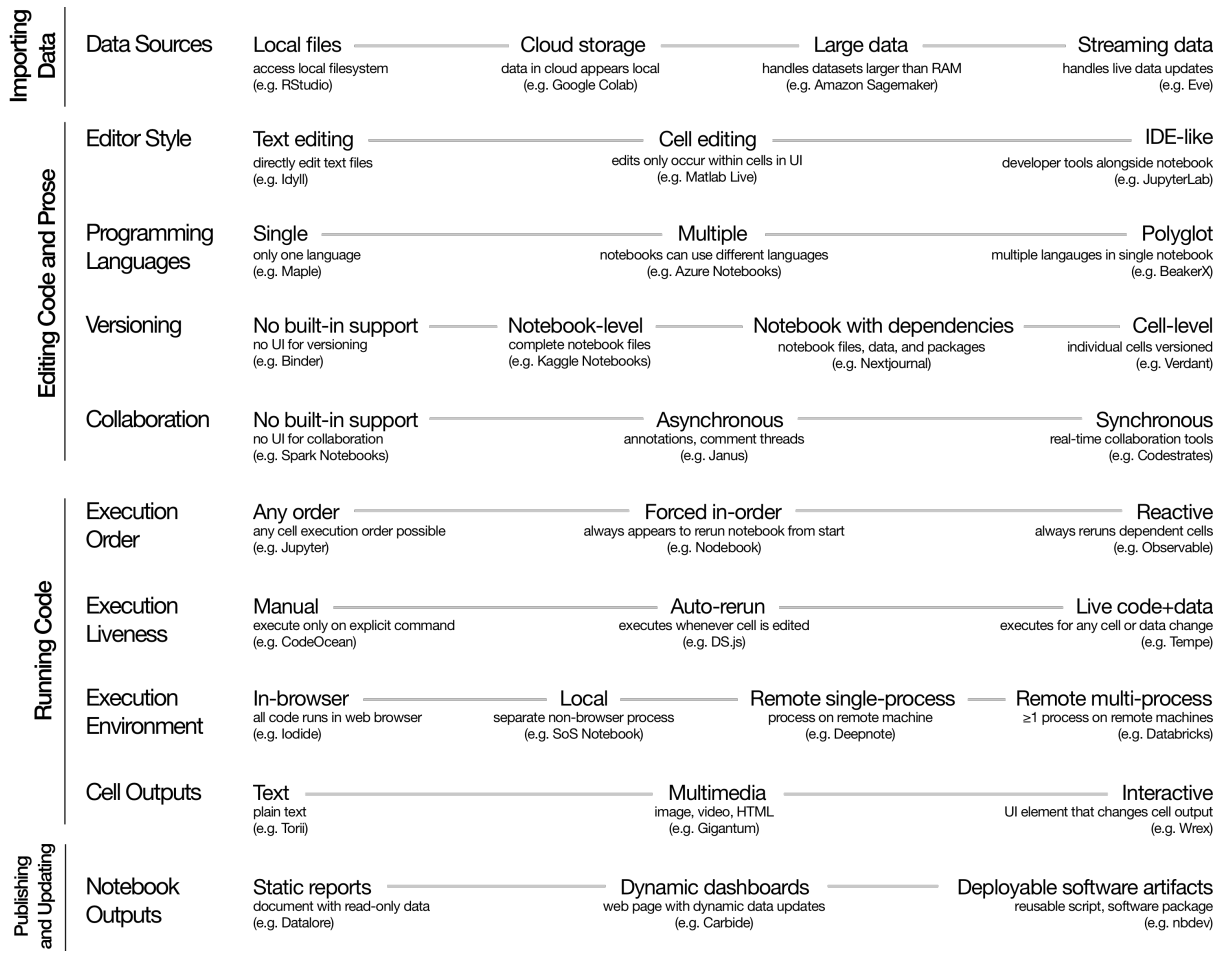


Figure 5.1. The design space of computational notebooks, which we formulated by analyzing the features of 60 notebook projects across academia and industry. As Table 5.1 shows, each individual project can occupy multiple points along each dimension of this design space.

Figure 5.1 shows the design space we created by analyzing the features of 60 notebook systems (16 academic papers, 29 industry products, and 15 experimental/R&D projects). We grouped our 10 design space dimensions into four major stages of a data science workflow: importing data into notebooks, editing code and prose (editor style, supported programming languages, versioning, collaboration), running code to generate outputs (cell execution order, liveness [239], execution environment, and cell outputs), and publishing notebook outputs.

We found that industry products often prioritize widespread adoption, so their designs do not deviate much from that of the widely-used Jupyter Notebook format. In contrast, academic and R&D projects can afford to experiment more with different kinds of cell execution orders, live programming, and interactive outputs. That said, most designs are still restricted by people's current data science workflows, which involve interfacing with desktop computers using keyboards and mice. To innovate beyond this frontier, we suggest supplementing future notebook systems with non-desktop computing devices such as mobile and ubiquitous computing devices. Another way to push beyond current designs is to focus on user groups other than professional data scientists, such as educators, artists, or those in low-resource computing environments.

In sum, this study's contributions are:

- A definition of "computational notebook" (Section 5.2A)
- The first comprehensive design analysis of computational notebooks, identifying 60 notebook systems across three categories: academic, product, and experimental/R&D.
- A design space that summarizes notebook features, and ideas for how researchers can innovate beyond its bounds.

5.2 Methods

We performed a qualitative analysis of 60 computational notebook systems across academia and industry.

5.2.1 Defining the Term "Computational Notebook"

Our first task was to define "computational notebook" to determine what projects to include in our study. On one hand, we did not want to restrict ourselves to only variants of Jupyter Notebooks, which is now the most popular format [205]. On the other hand, we did not want to expand our definition too much to include projects that were too distant. Based on prior studies of notebook use [222, 220, 156, 94, 195, 258, 244] and our own experiences as notebook researchers, ***we define a computational notebook as a system that supports literate programming [164] using a text-based programming language (e.g., Python, R, JavaScript, or a DSL [190]) while interweaving expository text and program outputs into a single document.***

This definition excludes "non-computational" notebooks such as electronic lab notebooks [131] used by scientists to track their daily work, general notetaking tools such as OneNote and Evernote, and modern notetaking apps with spreadsheet-like computational capabilities (e.g., Airtable [1], Coda [6], Notion [36]). It also excludes projects like Distill.pub [28] and Explorable Explanations [13], which provide widgets for users to interactively tune parameters but which do *not* support writing text-based programming languages in the notebook.

Since many notebooks are implemented as Jupyter extensions, it can be hard to tell what counts as a standalone system. In general, we counted academic contributions that are published papers and industry contributions that are marketed as either standalone products (both open-source and proprietary) or substantive R&D efforts. We excluded smaller extensions like postprocessing scripts that only transform notebook outputs (e.g., nbinteract [179], ThebeLab [49], Voila [50]).

5.2.2 Finding Notebook Systems Across Academia and Industry

We sought to be comprehensive in finding all publicly-known notebook systems that fit our definition. For academic projects, we performed a literature search starting with research papers at venues including VL/HCC, CHI, UIST, and CSCW, and then branched outward via bib-

liography crawls and Google Scholar searches for notebook-related terms (e.g., "computational notebook", "literate programming"). For industry projects, we consulted with data scientist colleagues, performed web searches for relevant terms, watched talk videos at practitioner conferences such as JupyterCon [21], and searched through discussion forums and blogs. We iterated on our list by showing drafts to other notebook researchers and practitioners, including core members of the Jupyter project team [55], to receive additional suggestions to include.

5.2.3 Data Overview and Analysis

We found 60 total systems (circa Feb 2020), summarized in Table 5.1. For each, multiple members of our research team independently enumerated its features by reading relevant papers and user guides/documentation, watching demo and talk videos, and trying out notebooks that were publicly available. We focused our qualitative content analysis purely on technical features rather than business-oriented features such as pricing, licensing models, or target markets. The research team met multiple times to merge our notes, categorize them together into themes using an inductive analysis approach [101], and iterate until we could not find additional features. To formulate a design space from these features, we followed a similar methodology as Segel and Heer [228], who made a design space of narrative visualizations from content analysis of 58 visualization webpages. Specifically, we sought to distill generalizable concepts from specific notebook features and find what concepts multiple notebooks shared in common. For instance, whether a notebook supports Python or R is an implementation detail, but the fact that a single or multiple languages can coexist within a notebook is a more generalizable design concept. We made several iterations as a team before finalizing our 10 dimensions, which are grounded in the phases of a data science workflow. We originally considered a larger number of more specific dimensions (e.g., notebook distribution) but merged them into four main workflow phases.

5.2.4 Study Design Limitations

Although we sought to be comprehensive, there is no guarantee that we found all relevant computational notebook systems; we do not have access to internal company projects nor to unpublished academic projects. As the first study of this scope, though, we believe our set of 60 is sufficient for surveying trends and forming the basis for future analyses.

Our goal was to use these 60 systems to map out a design space, so we do not describe *all* features of each one. Since we organized findings around the 10 design space dimensions rather than describing each notebook, we omit specific feature details such as the programming languages supported by each. Also, there is subjectivity in how we selected our dimensions, so other researchers may have picked different dimensions.

Since Jupyter is now the most popular platform, systems built on top of it are overrepresented in our data, accounting for around half of our 60 projects. That said, we included other ecosystems such as R, Mathematica, and Spark; we also found experimental and academic research projects that built their own bespoke notebook interfaces separate from Jupyter.

5.3 Results Overview: Sources of Notebooks

We identified three sources of notebooks, summarized in Table 5.1: academic, product, and experimental/R&D.

Academic: The first source includes 16 research projects that have been published as academic papers. Most of these are from university labs, but MessyNotebooks [140], Tempe [119, 109], and Wrex [112] are from Microsoft Research.

Each academic project usually makes one distinct and precise innovation. Some are implemented as Jupyter extensions: Wrex [112] adds programming-by-example to Jupyter by enabling users to interact with data tables and having the system synthesize data wrangling code; Callisto [245] adds inline chat and deictic references to facilitate anchored discussions around notebook cells; Janus [221] adds cell folding and annotations; Verdant [150, 152] adds automated

cell-level micro-versioning and development history visualizations; MessyNotebooks [140] generates slices of notebook cells that produce a given output; Bacatá [190] synthesizes Jupyter UIs for DSLs; Dataflow [170] and SoS [204] notebooks track provenance for reproducibility.

Many academic projects are not tied to Jupyter since their goal is to demonstrate novel ideas, not necessarily to gain wide adoption. Three such systems are built atop the Webstrates [161] platform: Codestrates [215, 81], Vistrates [69], and Labbook [202]. Also, PolyJuS [200], Tempe [119, 109], and Torii [141] implement their own custom UIs, Idyll [100] provides a literate programming [164] markup format, and DS.js [259] turns existing data-rich webpages into computational notebooks.

Product: In contrast to precisely-targeted innovations of academic research, notebook products are monolithic solutions aimed at broad adoption. We define *product* broadly to mean any system intended to serve a sizable user population; by this definition, products can be free or paid, open-source or closed-source, and maintained by for-profit companies or non-profits. Many of these are built on top of Jupyter. Note that since Jupyter is a platform (and accompanying nonprofit organization) rather than a specific product, in Table 5.1 we separately show the two official products maintained by this organization: the original Jupyter Notebook UI and the newer JupyterLab IDE [163, 205]; for these two, we count only their features from default installations without any extensions.

Many products are companies hosting Jupyter in the cloud to provide "Jupyter-as-a-service." They add features such as access to large-scale datastores, fast compute engines, real-time collaboration, or integration with other cloud services. General-purpose systems include Binder [3], Databricks [8], Gigantum [14], IBM Watson Studio [18], Kaggle Notebooks [23], Microsoft Azure Notebooks [31], and Mode [32]. Some are specialized for collaboration: Google Colab [15], Datalore [9], Deepnote [10], and Kyso [25]. Others have a domain-specific focus, such as CoCalc [5] (mathematics), Kogence [24] (supercomputing), Quantopian [40] (finance), and CodeOcean [7] and Nextjournal [35] (reproducibility).

Besides systems built atop Jupyter Notebooks, there are several other major ecosystems: RStudio [42] provides an open-source IDE for R where users can write RMarkdown Notebooks [41] and make interactive dashboards with the Shiny framework [44]. Mathematica (now renamed to Wolfram) has maintained their own proprietary notebook format since 1988 [52, 138]; Maple [29] and MATLAB [30] have similar embedded notebooks. Spark Notebooks [45] and Zeppelin [54] provide notebook interfaces to access the Apache Spark big data ecosystem using Scala and SQL. Spyder [46] provides a MATLAB-style IDE for Python with notebook support. Finally, startups such as Observable [38], RunKit [43], and Streamlit [47] have created their own notebook-based environments for rapid prototyping of data-driven web apps.

Experimental / R&D: This final source contains 15 systems that fall in between academic and product types. They differ from academic research in that they are not formally evaluated or published as papers, and they differ from industry products in that they are less polished. Note that the line between experimental/R&D and industry products may be blurry. One distinction is that experimental/R&D projects do not feel as "standalone" as products do, and their websites are often just a GitHub code repository with some technical documentation.

Nonprofits have created experimental systems such as Iodide [19] from Mozilla, nbdev [33] from fast.ai, Livebook [27] from Ink & Switch, and the nteract notebook UI [37] and Papermill [227, 242] production scheduler from NumFOCUS. Independent creators have started projects such as Carbide [4], Leisure [26], and Eve (a startup that was in R&D phase) [12].

Some companies also release their R&D projects as open-source software: Stitch Fix altered Jupyter with forced in-order cell execution in their Nodebook [34] system. Stripe shared their reproducible production notebook workflows [122]. Finance firm Two Sigma extended Jupyter with polyglot JVM and Spark support in BeakerX [2], which is similar to Netflix's Polynote [39] project. Another finance firm, Capital Fund Management, released Jupytext [22] to provide a Markdown-like text editing experience for notebooks. Several companies also developed plug-ins to integrate notebooks into traditional IDEs, such as Hydrogen [17] for the

Atom IDE and Microsoft's Jupyter mode for Visual Studio Code [53].

5.4 The Design Space of Computational Notebooks

Table 5.1 summarizes how all 60 notebooks we analyzed fit into our design space from Figure 5.1. We grouped our 10 design space dimensions by their typical order in a computational workflow: importing data into notebooks, editing code and prose, running code to generate cell outputs, and publishing notebook outputs. Note that our design space covers more than just the core UI of the notebook itself; it captures the entire end-to-end *system* that the notebook resides within.

5.4.1 Importing Data into Notebooks

1) Data Sources: The baseline is accessing only *local files*, which means that it is up to the user to manage their data sets and import them using a programming language and libraries (e.g., the Python CSV reader module). Default Jupyter Notebooks and most non-cloud variants fall into this category.

Cloud storage means that the system exposes data sets stored in a cloud service as though they were local files. For example, Google Colab [15] lets users mount a Google Drive folder and then access its files in the notebook.

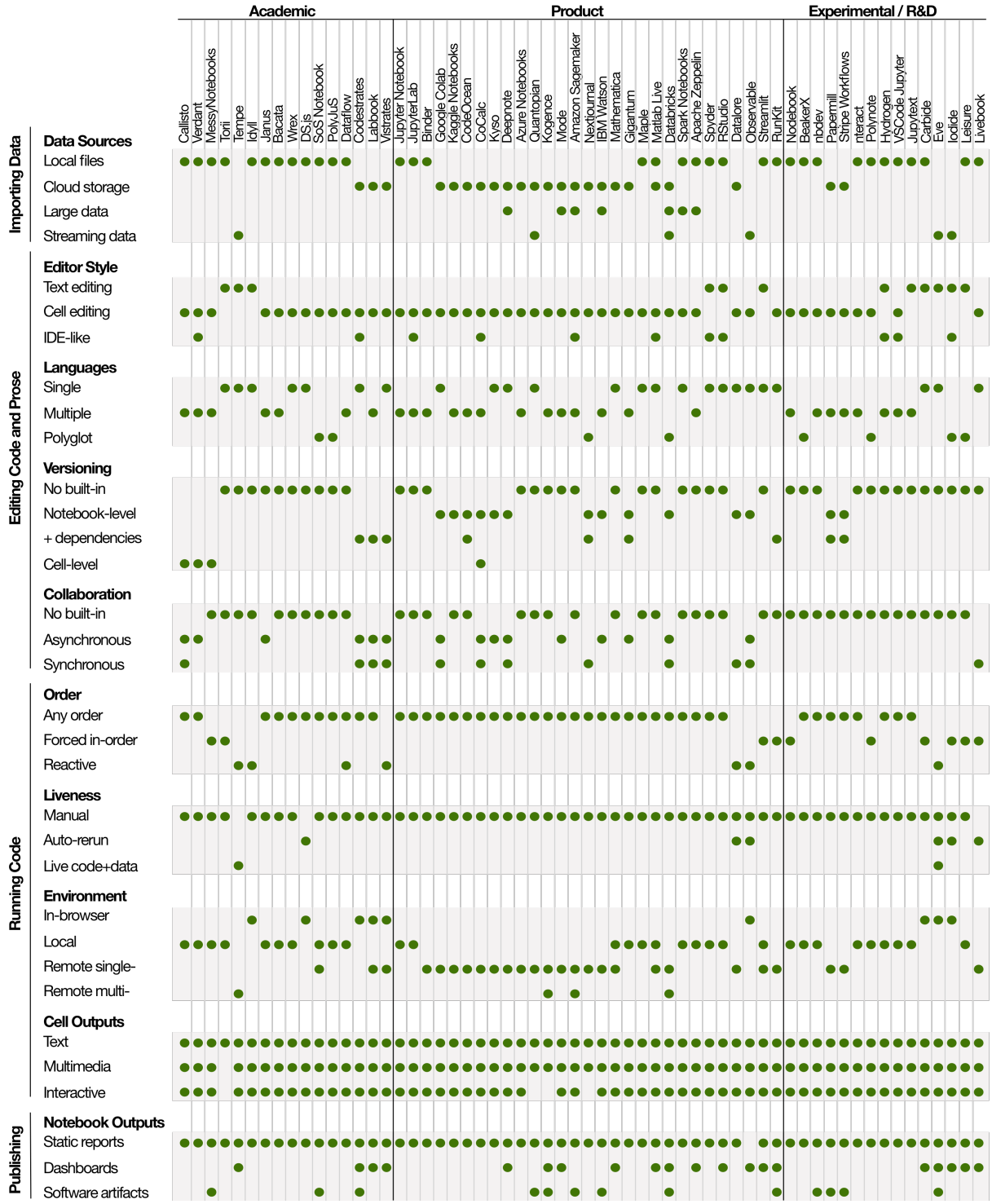
Large data means that the system has built-in support for accessing data that is too large to fit into RAM; these datasets cannot be fully imported into a running notebook session. For instance, Databricks [8] and Spark Notebooks [45] allow users to write SQL queries within notebook cells to access selected slices of terabyte-scale data within a session.

Finally, *streaming data* means the notebook allows users to both connect to real-time streaming data sources *and* automatically update its computed outputs as new data arrives.

5.4.2 Editing Code and Prose

Four design space dimensions relate to editing interfaces:

Table 5.1. How the 60 notebook systems in our study (columns) fit into the design space dimensions (rows) we illustrated in Figure 5.1.



2) Editor Style: At one end, systems like Iodide [19] and Streamlit [47] let users write code and prose in an enhanced text editor; those systems compile text into interactive notebook-like webpages. In the middle, most notebooks provide a more structured cell-based editor where users can independently edit, run, and rearrange cells within a notebook file. At the other extreme are fully-featured IDEs (e.g., JupyterLab, RStudio) that embed notebooks alongside data inspectors, debuggers, terminals, and other developer tools. Codestrates [215] is the most unique since it allows users to write JavaScript to customize its own UI; although it starts as a basic cell-based editor, it can be reprogrammed into an IDE.

3) Programming Languages: The baseline here is support for coding in a single language (e.g., JavaScript for Codestrates [215], a JavaScript variant for Observable [38], a Datalog-like DSL for Eve [12]). Next is support for multiple languages, but each notebook file can only run a single language; default Jupyter works this way. Polyglot notebooks such as Polynote [39] and PolyJuS [200] enable users to natively mix multiple language within the same notebook, which is useful when the most convenient libraries for different stages of a data analysis are in different languages (e.g., web scrapers in Perl, machine learning in Python, statistics in R).

4) Versioning: Studies found that notebooks are often used for exploratory workflows [153] involving lots of trial-and-error [222, 156, 94] and that users struggle to keep track of many versions of analysis code and outputs. The baseline here is no built-in support, which means that it is up to users to track their own versions by, say, committing notebooks to Git. Next is *notebook-level versioning*, where the system automatically saves versions of entire notebook files so users do not need to learn about version control. Next is *notebook+dependency versioning*, where systems like Nextjournal [35] save not only raw notebook files but also accompanying data sets and environment dependencies (e.g., 3rd-party libraries and packages). This enables notebooks to be reliably re-run to reproduce the same results, which is important for replicability of scientific results. At the most extreme is *cell-level versioning*, where each code and output cell can be separately auto-versioned. For instance, Verdant keeps track of individual cell edits at the

code AST level so that "each syntactically meaningful span of text in the code can be recorded with its own versions" [150].

5) Collaboration: Studies also found that notebooks are often used in collaborative data science workflows [244, 195, 258]. The baseline here is no built-in collaboration support, so users must coordinate via a mix of third-party tools (e.g., workplace chat with Slack, code review and issue commenting with GitHub's web UI). Next is built-in support for asynchronous collaboration, most commonly via annotations and comment threads next to notebook cells. Finally, some support synchronous collaboration by embedding real-time chat and allowing multiple users to concurrently edit the same notebook cells, akin to Google Docs. (Note that Google Colab was originally designed for synchronous collaboration but its real-time sync API is currently down [16].)

5.4.3 Running Code to Generate Cell Outputs

We identified four design space dimensions related to code execution: order, liveness, environment, and cell outputs.

6) Execution Order: Notebooks contain a series of code cells, each of which can be run independently to produce outputs. Most allow cell execution in *any order*, which means users can run cells out of order and multiple times as they iterate. However, both academic studies [222, 156, 94] and industry reports [127, 20] show that any-order execution is a major source of frustration for notebook users; specifically, it is hard to tell which exact series of cell executions led to the notebook's current state or how to reproduce that state. A recommended best practice is for users to clean up their cells and run them all in-order to create a final shareable notebook.

To reduce these frustrations, some notebooks implement *forced in-order execution*. This means when the user clicks "Run" on a particular cell, the notebook will reset its global state and then run all cells from the top until that cell. (That is the user's mental model, but in reality these systems such as Nodebook [34] track dataflow dependencies so that not all cells need to be

re-run each time.) This design eliminates the problems of out-of-order cell execution, of cells being run multiple times and altering global state in unexpected ways, and of cells being deleted after being run but their outputs still remaining in the notebook's global state. But the tradeoff is lack of flexibility, since some users appreciate rapid prototyping by running cells in any order. As a middle ground, systems like MessyNotebooks *retroactively* turn an existing notebook into a forced in-order one by backward-slicing only the cells that lead to a certain desired output [140].

Finally, some notebooks implement a *reactive* execution model, much like spreadsheets do. In a reactive model, running a cell triggers the notebook to automatically re-run all other cells that depend on values defined or altered in that cell. This model gives users the flexibility to write their code in any notebook cell regardless of order, but run-time value dependencies are automatically tracked by the notebook so there is still a predictable (albeit non-linear) execution order.

7) Execution Liveness: Most notebooks run a cell only when the user clicks "Run" or uses a keyboard shortcut. Some have simple forms of live execution where a cell is automatically re-run whenever the user momentarily pauses editing or moves their cursor to another cell; this type of liveness (level-3 in Tanimoto's taxonomy [239]) provides immediate visual feedback, which can speed up the iteration and debugging cycles. Systems like Tempe implement a more sophisticated form of level-4 liveness [109] by updating the output live not only after the user edits a cell but also in real time when the data stream that the cell accesses has new data arrive in it.

8) Execution Environment: Notebook systems also vary in *where* they execute code. JavaScript-based notebooks such as Codestrates [215] and Observable [38] run code directly in the user's browser, which is a convenient way to eliminate complex installation and setup issues; Iodide [19] compiles a Python environment to WebAssembly to run directly in the browser. Next, most desktop Jupyter variants run code locally on the user's computer in a separate non-browser process that communicates with the notebook's browser-based UI. Cloud-hosted Jupyter systems

run notebook code in a single process on a remote machine with access to more computational power and GPUs than the user's machine. Finally, systems like Databricks [8] and Kogence [24] automatically set up notebooks to run in parallel and distributed computing environments involving *multiple* remote machines at once.

9) Cell Outputs: When each cell is executed, it produces output directly underneath or beside it. This output can range from plain text (akin to a terminal or REPL) to multimedia (e.g., data tables, images, sound clips, videos) all the way to interactive widgets. These widgets allow users to set parameter values and re-run cells in order to quickly prototype variants of data analyses. Note that the systems in Table 5.1 that are built atop Jupyter support all types of outputs since Jupyter comes bundled with an interactive widget library. Finally, some research systems go even further by allowing interactive widgets to alter the code in the accompanying cell: Carbide [4] exposes sliders whose values are synchronized with numeric literals in code cells, and Wrex allows users to do programming-by-example [112] by entering example values in output tables and having the system automatically synthesize Python data wrangling code to insert into nearby code cells.

5.4.4 Publishing and Updating Notebook Outputs

Notebooks are often used for personal exploration and end-user programming workflows [153, 222], but users sometimes intend to share notebooks with others. They do so by publishing it in various formats, which have different levels of support for updating in response to post-publication feedback.

10) Notebook Outputs: The baseline here is for notebooks to generate *static reports* that are read-only documents. For instance, Jupyter Notebooks can be exported as static HTML files, but those might contain JavaScript-powered interactive visualizations (e.g., using D3 [82] or Vega [225]). Next, some can be exported as *dynamic dashboards* that (unlike static HTML) automatically update and recompute cell outputs as new streaming data arrives. Both static and

dynamic formats facilitate common explanation use cases [222] for notebooks.

Lastly, some have stretched notebooks beyond data science use cases to turn them into *deployable software artifacts*. For instance, nbdev [33] exports notebooks as self-contained Python libraries that can be imported into other software projects; IBM Watson Studio [18] deploys notebooks as machine learning models with live monitoring; Codestrates [215] and Eve [12] allow users to create standalone web applications.

5.5 Discussion

High-Level Trends: Industry products are geared toward scalability and adoption, so their designs tend to be standard Jupyter Notebooks integrated into a cloud platform (e.g., for data sources, automatic versioning, execution environments, and distribution). Although one could in theory emulate these features by installing open-source software and manually provisioning cloud resources, these products provide a level of convenience that fosters widespread adoption.

In contrast, academic and R&D projects are more experimental, adopting less conventional designs for execution order, liveness, and interactive outputs. They also contain properties that are hard to fully capture in our design space: For instance, Codestrates [215] and systems built on it emphasize malleability where the entire notebook programming interface can be adjusted on-the-fly by editing its code. Systems like DS.js [259], Idyll [100], and Torii [141] target educational use cases rather than being for professional data scientists, so their interfaces differ from conventional programming-oriented notebooks.

Design Tradeoffs: Each dimension in Figure 5.1 has associated tradeoffs. 1) Data sources: moving to the cloud may sacrifice privacy and impose more friction for working with legally-protected data (e.g., COPPA, HIPAA), 2,3) Editor style and languages: IDE-like interfaces and polyglot notebooks make the UI more complex for novices to learn, 4) Versioning: finer-grained versioning again requires greater UI complexity to let users organize and sift through versions, 5) Collaboration: integrated collaboration may not always be better since users may prefer to

use their own organization's external tools (e.g., Slack), 6) Execution order: in-order is simplest but lacks flexibility; reactive can be hard for novices since it is not clear which cells execute when a particular one is edited (*hidden dependencies* in Green's cognitive dimensions [126]), 7) Liveness: again it may not be clear what code executes if the user does not initiate those actions, 8) Environment: remote execution is higher-performance than local but can be harder to debug, 9,10) Cell/notebook outputs: the more interactive, the less of a *closeness-of-mapping* (in Green's cognitive dimensions [126]) there is between code and outputs.

There are also design tradeoffs *across* dimensions: For instance, an in-browser JavaScript notebook may have a hard time supporting polyglot programming (though not impossible as more languages now compile to WebAssembly); and notebooks that rely on remote multi-process execution may be hard to package up as portable/versioned software artifacts, since they often depend on the specifics of their execution environment (e.g., supercomputer infrastructure from Kogence [24]).

Notebook Toolkits: As a thought experiment, what about a "kitchen-sink" approach where a hypothetical system covers the *entire design space* of Figure 5.1 by providing *all* these features as options? While that might seem appealing since it would foster customization, from a usability perspective its interface would get overly complex with too many modes. And from a software engineering perspective it would be hard to get such a monolithic system working robustly. Perhaps the most practical path along this direction is to take a toolkit approach like D3 [82] by turning each dimension of Figure 5.1 into a well-scoped software module and then allowing developers to compose them together using a "grammar of notebook systems" akin to the grammar of graphics implemented by the Vega [225] ecosystem. That way, software developers can easily build bespoke notebooks for their intended user group.

Charting the Future of Computational Notebook Research: Our study provides a map of the current state of the notebook world, as summarized in Table 5.1. In the short term, industry products will continue to fill in more points along our design space. Meanwhile, researchers in

both academia and industry can expand the boundaries of this map into uncharted territory. What are possible ways forward? Here are a few directions:

There are still many opportunities for needs-based research of the sort embodied by some of the academic projects in Table 5.1. The HCI approach of understanding the needs of a user group, building a novel notebook system to address those needs, and evaluating it on that user group will continue to yield research innovations. To stretch the bounds of our current design space, we encourage researchers to broaden out to target user groups beyond professional data scientists, such as educators, artists, designers, journalists, digital humanities scholars, young children, older adults, people with accessibility needs, or those in low-resource computing environments.

Though important, we believe that needs-based research will mostly continue to fill in our design space but not expand much beyond it. That is because people's needs are still tied to their existing workflows; for computational notebooks that means using keyboards to write textual code on desktop or laptop computers. Thus, one way to go beyond the current design space is to think about the abundance of ***non-desktop computing devices*** available to us in our daily lives and how we could use those for data-oriented work. For the price of a laptop computer we can now buy several low-cost tablet- and phone-sized devices. Weiser's ubiquitous computing vision is already here since we have plentiful access to pads (e.g., iPads) and tabs (e.g., phones/watches) [247] but are not yet taking full advantage of them for programming. Practically, we do not envision these devices replacing keyboard+mouse+monitors but rather *supplementing* them with auxiliary displays, dynamic magic lenses [74] (e.g., hovering a phone over a monitor to peek into code or data), and touch and voice-based inputs. Beyond pads and tabs, what about other ubicomp devices such as wearables, smart glasses, augmented/mixed-reality interfaces, portable projectors with 3D depth cameras, and large-scale tabletop and wall displays? Specifically, Table 5.1 shows that there is not much variation in cell outputs; a way to innovate along this dimension is by adopting different sorts of displays beyond ordinary computer screens. Similarly, many systems do not support synchronous collaboration, and when

they do, it is often simple forms of multi-user text editing; once again ubiquitous computing techniques can point the way toward more expressive forms of synchronous collaboration.

The above was a technology-centric approach (start with novel devices and prototype outward), so a complementary approach is to be task-centric. What do people actually want to do with notebooks? Examples include deriving business insights, conducting scientific research, scholarly communication, collaboration, education, and personal creative expression. What future interfaces that mix code, data, and multimedia might aid these tasks, regardless of whether they "look" like the cell-based notebooks of today? We encourage researchers not to have their thinking restricted by current notebook formats simply due to the zeitgeist of recent academic publication trends. One metric for success here is how *different* a new project looks from all the systems in Table 5.1.

A more radical way to expand beyond current notebook designs is to ban the term "computational notebook" altogether and generalize it into interfaces for literate computing (i.e., literate programming + interactive media) [121]. One could argue that Figure 5.1 is actually a design space of literate computing, of which notebooks are a subset. This approach may mean reviving classic lines of work embodied by Smalltalk, Boxer, and HyperCard [121], which Bret Victor et al. are doing at Dynamicland [11]. One challenge of such ambitiously generalizable research is to balance expressiveness with usability/learnability in order to avoid the Turing tar-pit, "in which everything is possible but nothing of interest is easy" [207].

5.6 Conclusion

We identified three main sources of computational notebooks (academic, industry products, and experimental/R&D) and presented the first comprehensive study of 60 notebook projects, which resulted in a ten-dimensional design space that spans their technical features. In closing, we encourage researchers to stretch the bounds of this space by pursuing project ideas that go beyond fulfilling current user needs.

Chapter 5, in full, is a reprint of the material as it appears in the proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC) as The Design Space of Computational Notebooks: An Analysis of 60 Systems in Academia and Industry. Sam Lau, Ian Drosos, Julia M. Markel, Philip J. Guo. 2020. The dissertation author was the primary investigator and author of this paper.

Chapter 6

TweakIt: Supporting End-User Programmers Who Transmogrify Code

End-user programmers opportunistically copy-and-paste code snippets from colleagues or the web to accomplish their tasks. Unfortunately, these snippets often don't work verbatim, so these people—who are non-specialists in the programming language—make guesses and tweak the code to understand and apply it successfully. To support their desired workflow and facilitate tweaking and understanding, we built a prototype tool, TWEAKIT*, that provides users with a familiar live interaction to help them understand, introspect, and reify how different code snippets would transform their data. Through a usability study with 14 data analysts, participants found the tool to be useful to understand the function of otherwise unfamiliar code, to increase their confidence about what the code does, to identify relevant parts of code specific to their task, and to proactively explore and evaluate code. Overall, our participants were enthusiastic about incorporating* TWEAKIT *in their own day-to-day work.*

6.1 Introduction

Data analysts across a variety of diverse disciplines—chemists, molecular biologists, material scientists, and cognitive psychologists—routinely find themselves in situations where they must intersect their specialization with programming to accomplish their day-to-day work.

These data analysts are end-user programmers who conduct data analyses but don't see

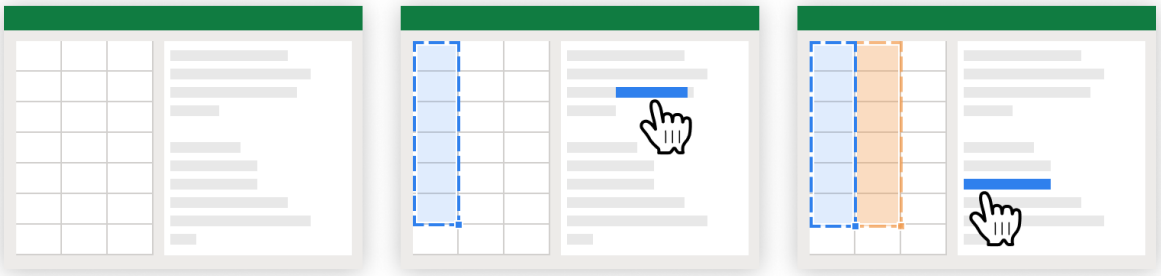


Figure 6.1. TWEAKIT is a system that enables end-user programmers to collect, understand, and tweak Python code within a spreadsheet environment. (Left) Users collect Python code snippets and insert these snippets into the scratchpad. (Middle) Users can select Python expressions and TWEAKIT previews the outputs directly in the spreadsheet. (Right) Users can compare two outputs; after users are satisfied with their program they can save the current selection as a table in the spreadsheet.

themselves as professional software developers. They often have little-to-no background in computer science at all; for them, code is rightfully just one of many tools in their toolbox that helps them to transform and analyze their data, explore hypotheses, and evaluate their findings. Through our formative interviews, we found that their coding workflow consists less about actually *writing code* and more about *transmogrifying* it: in a typical workflow, our data analysts opportunistically cobbled together various snippets of code from colleagues or online sites like Stack Overflow, tweaking these snippets with trial-and-error incantations, and applying educating guesses and makeshift heuristics about what lines of code to permute along their journey. Through a combination of grit, superstition, and serendipity, a "working" code snippet eventually emerged (well, sometimes).

Essentially, our data analysts reflect the "paradox of the active user" [90], where users are motivated to get their immediate task done and bypass conceptual resources—such as programming tutorials—in favor of directly applying and manipulating the objects of study to their work. Consistent with prior work on this "stable but suboptimal preference" [123], our analysts preferred actions with fast and incremental visual feedback like pasting and tweaking code. For better or worse, guess-and-check remained their interaction mode of choice. Rather than asking users to change how they behave, how can we design tools that support the highly

goal-oriented coding workflows they prefer?

This paper aims to address the needs of end-user programmers who transmogrify and tweak code through two contributions. The first contribution is the design and implementation of a prototype tool called TWEAKIT to support data analysts as they guess their way to working programs. In contrast to end-user programming tools that abstract code behind user interfaces, TWEAKIT allows analysts to work directly with code in their existing data workflows by situating code alongside spreadsheets. By applying a live interaction to a familiar affordance, TWEAKIT enables analysts to preview and compare code outputs without pausing their code-tweaking work.

Our second contribution is a set of insights gained from a first-use usability study of TWEAKIT with 14 data analysts. We found that analysts valued TWEAKIT's support for their preferred guess-and-check coding workflow. As analysts tweaked code, they relied on live output previews to narrow the search space of possible edits and used preview comparison as lightweight explanations for what code did. TWEAKIT's affordances encouraged code exploration and increased confidence without decreasing participant effectiveness and efficiency. As a whole, analysts reported many day-to-day tasks that they felt could be better addressed using code and were enthusiastic about support for code tweaking within their existing workflows.

6.2 Example Usage Scenario

Interaction with TWEAKIT is summarised in Figure 6.2. Robin is an expert molecular biologist working with viral RNA. After conducting her experiments, she opens her data using Excel—her preferred spreadsheet application—and explores the data using a combination of operations in the graphical user interface and spreadsheet formulas. After performing basic data cleaning, she has a column of RNA sequences stored as strings consisting of characters like `A`, `U`, `G`, and `C`. Robin needs to calculate the RNA complement of each sequence by replacing each letter with another. For example, every `A` should get converted into a `U`. Since her software does

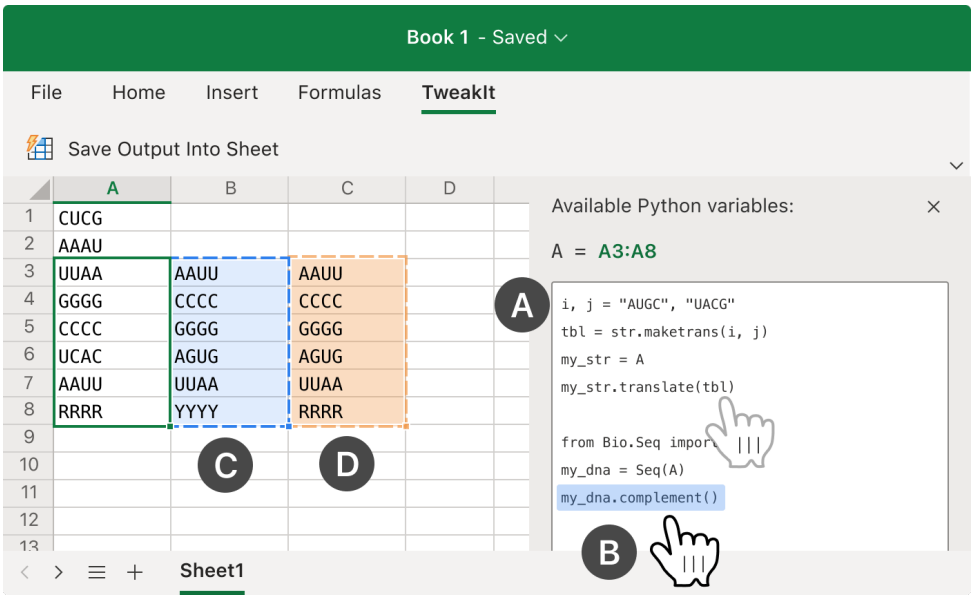


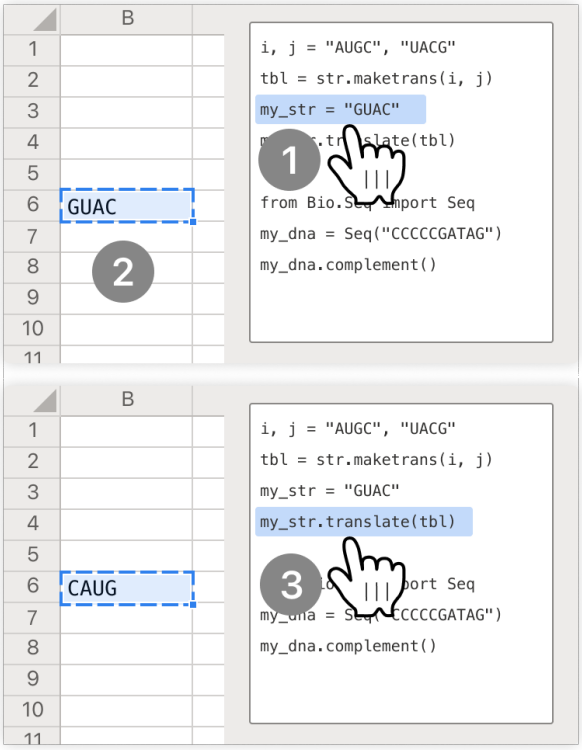
Figure 6.2. TWEAKIT supports data analysts who guess their way towards working code. (A) Users open a code editing pane directly within their spreadsheet application. (B) Users can click on any code expression to highlight it. (C) The selected expression output is displayed directly in the spreadsheet grid as data. (D) The previously selected output (from the expression under the light gray mouse icon) is retained in the grid to facilitate comparison.

not support this calculation via a simple formula, Robin performs a web search. She finds two Python examples, one using the Python `str.translate` method and one using the `complement` method from the Biopython package.

Robin begins working with these two example snippets using TWEAKIT. To get started, she clicks a button in her spreadsheet application to open a scratchpad for code directly inside her application. Robin pastes both examples into the scratchpad (A). On pasting, TWEAKIT automatically corrects minor syntax errors like missing quotes and parentheses at the start or end of the code. TWEAKIT also detects that Robin doesn't have the Biopython package installed and automatically installs this package so that the snippets run without error immediately after pasting.

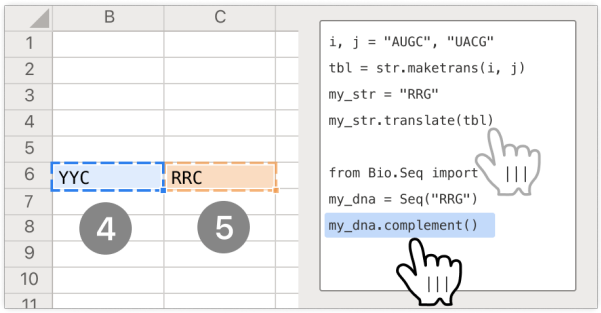
Next, Robin uses TWEAKIT's live output preview feature to understand what the example snippets do. Instead of looking up the documentation for each function call in the snippets, Robin can immediately see what each function does by clicking it in the code scratchpad—TWEAKIT

detects and highlights the complete code expression that surrounds the current cursor location (B) (1). TWEAKIT captures the output of this expression, then overlays this output as a provisional table directly in the spreadsheet (C) (2). This allows Robin to preview every expression by clicking through the example code (3).

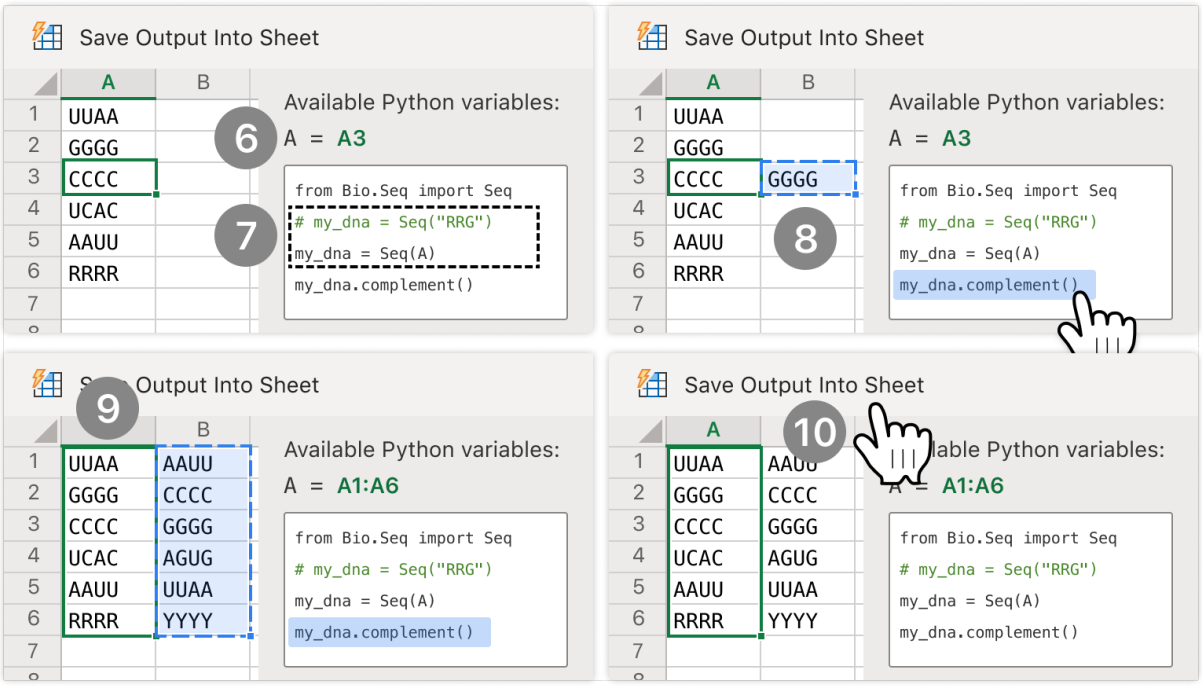


Robin uses TWEAKIT's output comparison feature to understand how the two example snippets differ. As Robin clicks through the example code, TWEAKIT displays both current and previous selected expression outputs in the spreadsheet (D). To directly compare the outputs of the two snippets, she clicks on the `str.translate` call, then clicks on the `complement` call. She sees that both examples produce the correct result for RNA sequences that use the characters `ACGU`. Robin also wants to check that the character `R` is converted to `Y`. She edits the code, adding `R` characters to the examples' RNA sequences. Robin compares the outputs of both snippets and finds that the BioPython example produces the correct output `YYC` (4) but the `str.translate` example does not (5); so, she deletes the `str.translate` example. Robin treats the live output

previews as explanations, using a combination of editing and comparing to deduce what the code does.



Finally, Robin uses TWEAKIT to apply the code to her spreadsheet data. When Robin selects cells in her spreadsheet application, TWEAKIT initializes Python variables that store the selected data 6. Robin selects cell A3, then edits the Python snippet to use the TWEAKIT variable `A` 7. When she previews the last expression of this code, she finds that it correctly calculates the RNA complement for cell A3 8. To apply this code on all her sequences, she selects all the data in column A of the spreadsheet 9. TWEAKIT automatically detects that the Python code needs to be run once per cell and computes all of Robin's desired RNA complements. To save this result into the spreadsheet, Robin clicks the "Save Output Into Sheet" button 10. Having completed this task, Robin can now proceed to further analyze her data through the spreadsheet.



Without TWEAKIT, Robin lacks a built-in method to use example code within her spreadsheet application. She would have to figure out how to export her data out of her spreadsheet, import the data into Python, paste the example script into the Python interpreter, edit the script to operate correctly on her data, then import the data back into her spreadsheet. Instead, TWEAKIT enables Robin to use example Python code as a single step within her desired workflow, and makes the code concrete through live output previews and comparisons so that Robin can figure out what edits to make.

6.3 Formative Interviews and Design Goals

We conducted interviews with 10 data analysts from a broad range of industries: finance, biotechnology, software, real estate, medical devices, environmental engineering, and IT services. All participants used formulas in a spreadsheet application like Microsoft Excel daily. None of the participants were expected to use programming for their work, and most participants (7/10) considered themselves beginners at programming. In our interviews, we focused on tasks that analysts found difficult to complete using their spreadsheet application, the workarounds they

used to complete their tasks, and their desires for improving their workflows. We transcribed the interviews, then used thematic analysis to develop and organize themes. These data analysts (F1-F10) provided several insights that guided the design goals for TWEAKIT.

All ten analysts regularly encountered data manipulation tasks that they felt lay beyond the capabilities of their spreadsheet application. Under constant pressure to meet deadlines, analysts felt the need to "just get it done somehow" (F1, F5). Thus, analysts edited data and formulas manually (F1, F3, F4, F5, F7, F8, F10). They felt this was "inefficient" (F2, F4), "annoying" (F3, F5), and "time-consuming" (F1, F7), especially for recurring tasks like generating a weekly report. To find better solutions for these tasks, all participants used web search, which often produced simple code examples that leveraged software packages. However, participants did not know how to begin using these code examples for their data since their spreadsheet application did not provide a visible method for doing so. F5 lamented that she spent "hours" cleaning data every week when "I know that in Python it's just three lines of code, but I just don't know where to start [putting the code in my spreadsheet]." F2 had experience using Python and used Python examples to automate his data tasks, but reported that "my coworkers all want to use [my script] but I definitely don't want to help them install [Python and its packages]." This feedback led to our first design goal:

D1. Code tweaking tools should enable users to paste and run code with as little additional setup as possible.

Without a method to use code examples directly within their spreadsheet, analysts turned to standalone coding tools like VBA macros (F1, F7, F9, F10), Jupyter notebooks (F2), SQL (F5), and JMP (F6, F8). However, analysts experienced disruptions in workflow using these tools, reporting that tools forced them to remember how to use a "completely different user interface [than Excel's]" (F2, F6, F8). Analysts faced friction importing data, running the tool, then exporting data every time "one little thing changes" in the sheet (F3, F8). Workflow disruptions occurred even when using VBA macros, a built-in scripting system in Excel. F1 explained that

"the VBA editor takes up your whole screen with code [...] when I'm working with VBA, I can't think in Excel anymore." These observations led to our second design goal:

D2. Code tweaking tools should embed themselves within workflows that are already familiar to users.

Code snippets from the Web presented challenges for reuse. Analysts reported that although code examples were readily available online, these examples "never do exactly what I want" (F3, F9) and "don't work out of the box" (F2). Even if an example performed the right calculation on toy data, analysts still had to edit the code to operate on their spreadsheet data. Analysts attempted a variety of code tweaks. For example, they edited column names (F2, F3, F5), changed arguments to function calls (F6, F8, F9), and duplicated lines (F3, F8, F9). Analysts described this process as "trial-and-error" (F3), "guess-and-check" (F2), and "fiddling" with the code (F5).

Although analysts were sometimes able to tweak code successfully, they found it challenging to understand unfamiliar code. F3 explained that "I never took a VBA class, [...] so it's hard to know what [code] does what [change]." F1 added that "it's scary to work with [code] because it can mess up my data without me even realizing it." To overcome this challenge, analysts wished to introspect code examples. F2 explained that "a blob of code is like a black box [...] I have to break it into pieces to figure it out." F9 developed a technique to send VBA output to an spreadsheet cell, explaining that "it's time-consuming, but I can at least narrow down which [parts of code] aren't right."

Overall, analysts expressed a desire to tweak code but faced barriers in understanding unfamiliar code examples and feared making irreversible mistakes to their data. These observations led to our third design goal:

D3. Code tweaking tools should reify unfamiliar code by showing how changes in code cause changes in data.

6.4 System Design and Implementation

TWEAKIT is implemented as an extension to Microsoft Excel using the Office JavaScript API. We modified a prototype version of Microsoft Excel implemented in TypeScript for ease of integration and to allow user study participants to open a URL rather than install software on their personal computers. [1]

Opening the TWEAKIT sidebar initializes a Python interpreter on the server that executes Python code for the remainder of the open browser session. When a user pastes in Python code into the scratchpad, TWEAKIT parses the Python code for package import statements and attempts to install packages if not already installed by running `pip install` with the package's name in the code. TWEAKIT also attempts to automatically import packages based on a hard-coded list of canonical package abbreviations (e.g. `np` for the `numpy` package). Finally, TWEAKIT checks for unmatched parentheses and quotation marks in pasted code and attempts to fix these errors by prepending or appending the missing marks to the code. If the code still produces an error after these attempted fixes, TWEAKIT falls back to leaving the code in its originally pasted form. While these heuristics are relatively simple, they are nevertheless useful and handle a variety of common situations.

To implement output previewing, TWEAKIT parses the abstract syntax tree (AST) of the Python code and keeps track of where each expression is located in the code. TweakIt uses the AST to find expressions to evaluate. For example, if a line contains

```
df.query().groupby().mean()
```

and a user clicks on the `groupby()` subexpression, TweakIt uses the AST to know to run `query()`, display the output of `groupby()`, but not run `mean()` (Figure 6.3). TweakIt's AST parser skips control flow statements like `if/else` and `for` loops, so these statements do not

[1] The Office JavaScript API is available to the public and allows extensions to read data, write data, and draw shapes in an Excel sheet. While the online version of Excel used for the TWEAKIT prototype is not publicly available, future extensions like TWEAKIT could be implemented in pure JavaScript/Typescript, using the Office JS API in the publicly available Excel for the Web.

generate output previews in the spreadsheet. This might be addressed by incorporating additional program visualization techniques like Projection Boxes [181].

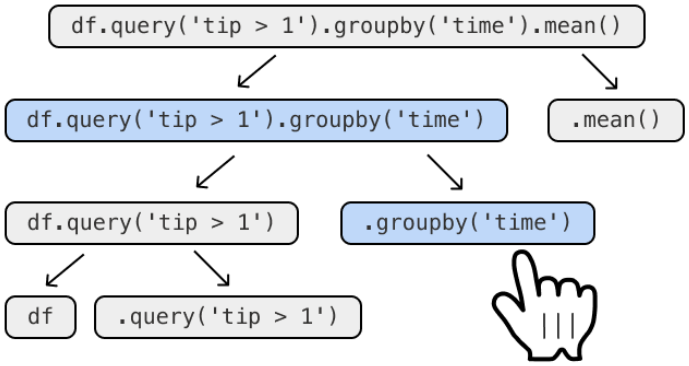


Figure 6.3. TWEAKIT uses the code's abstract syntax tree (AST) to decide which expression to execute. In this example, a user has clicked on the `groupby()` call. TWEAKIT finds the closest parent that is a complete expression in the AST, highlights the expression in the code editor, and displays the result of the expression in the spreadsheet.

When the user moves the cursor by clicking or typing, TWEAKIT finds the closest parent expression in the code AST that contains the new current cursor location, runs all the lines above the expression to reinitialize the program state, then runs the highlighted expression. TWEAKIT then displays the output of the Python interpreter in the Excel grid through its TypeScript API, with special cases to render Python collections and DataFrames as columns and tables. This implementation executes lines multiple times to generate intermediate values so code with side effects like disk I/O might be run multiple times unnecessarily. This limitation can be addressed by caching previous results to avoid recalculations.

To determine where to place the output in the grid, TWEAKIT applies another simple heuristic: it looks for the closest blank column to the right of the current Excel selection and places the output in that location. When the user highlights a different expression, TWEAKIT displays the new code output, then displays the old code output to the right. Finally, when the user selects cells in Excel, TWEAKIT extracts the data into a Python DataFrame named `df` so that the user's code can reference the `df` variable. TWEAKIT also initializes variables for each column

of the Excel selection; if the user selection spans columns B and C in the sheet, TWEAKIT initializes Python variables named `B` and `C` to contain the selected data in those columns. To execute code, TWEAKIT greedily runs the Python code on the entire column of data as a `pandas` Series object. If an error occurs, TWEAKIT will then to run the Python code once for every value in in the column. If the code still errors, TweakIt doesn't place any output into the sheet and instead displays the error message beneath the code. For simple cases, this heuristic allows the same code to work for both single cell and multiple cell selections.

6.5 In-lab Comparative First-Use Study

Our user study sought to understand how non-professional programmers used live output previews to edit and reuse existing code snippets.

Participants. We used purposive sampling to recruit data analysts through the UserTesting platform[2] and email. We looked for people who used Excel on a daily basis, were not professional programmers, yet still used programming as part of their work. While formative study participants were not required to have any programming background, in the user study we screened for participants that used programs in their data workflows. In pilot studies, we found that participants with no programming experience did not successfully edit code examples within the study's time limits.

Our final group of participants consisted of 14 data analysts across 7 industries. Of these, seven self-reported having little or no experience in programming with non-Python languages, and the remainder reported having a lot of experience. 13 reported having little or no Python experience, and one reported having a lot of Python experience.

Tasks. We designed six tasks that are representative of real-world code tweaking behaviour. All six tasks were based on actual tasks that participants in our formative interviews reported needing to do. Each task consisted of: a) a short textual description of the objective of the task, b) input data to be processed in some way, c) the exact desired output data, and

[2]https://www.usertesting.com/

d) a collection of Python code snippets taken verbatim from Stack Overflow searches with keywords from the task description. Participants selected, combined, and modified code snippets to produce a script that transformed the input data to the desired output. This was intended to be representative of a real-world scenario where a data analyst with a spreadsheet task looks to Stack Overflow for code that may help them with their task—not all snippets are relevant and the solution usually requires the combination and modification of multiple snippets.

Participants were allowed to search the web for documentation, reference material, and additional code snippets. Participants were allowed to author their own code from scratch—they were not required to use the snippets, but in every instance of a successful task in our study participants used the snippets they were given. Participants were not allowed to manually edit the input data or their output data to complete the transformation, nor were they allowed to use spreadsheet formulas: to be a valid solution, the script needed to perform the entire data transformation in a self-contained manner.

Protocol. We conducted a remote lab study in 3 phases: training, tasks, and survey. In the training phase (approximately 15 minutes), the experimenter guided the participant through a sample data manipulation task using both TWEAKIT and a baseline system that had a button see the output of the entire snippet rather than TWEAKIT's live previews. The task phase was divided into two blocks of 20 minutes each: in one block only the baseline system was available, and in the other only TWEAKIT was available. In each block participants were given 3 tasks and asked to complete as many as they could within 20 minutes. Although participants could work on tasks in any order, most chose to attempt the tasks in the order presented by the interface. Participants were asked to think aloud during tasks. The experimenter answered questions about the task goal and intervened to help participant recover from bugs in the system implementation, but did not intervene otherwise. The order of conditions and the assignment of tasks to conditions were both balanced: half of the participants used TWEAKIT first and the other half used the baseline first; each of the six tasks were assigned an equal number of times to both TWEAKIT and the baseline over the course of the study. After each 20-minute block, participants completed a survey about

how the tool helped them understand the code. After both blocks were completed, participants completed an additional closing survey, and the experimenter conducted a brief interview on their experience using the two systems. On average, the study took one hour and fifteen minutes to complete.

6.6 Quantitative Results

6.6.1 Code reuse tasks

In total, participants in the baseline condition completed 15/42 tasks. Participants achieved similar completion rates in the TWEAKIT condition: they completed 13/42 tasks. This difference was not identified to be statistically significant using Fisher's exact test. The low task completion rate can be explained by the difficulty of tasks—the only participant who had experience using the `pandas` package (P11) completed three out of the six tasks given. Four participants (P8, P9, P12, P14) did not complete any tasks at all because they repeatedly encountered bugs arising from unfamiliarity with the APIs used in the code snippet.

Participants completed tasks in similar amounts of time in both baseline and TWEAKIT conditions. Baseline participants completed tasks using a mean of 14.1 minutes ($\sigma = 5.24$ min) per task and TWEAKIT participants completed tasks using a mean of 13.7 minutes ($\sigma = 5.31$ min) per task. These differences were not identified to be statistically significant using a t-test ($t(13) = 0.23, p = .79$). However, we include this analysis of time taken only as additional description of the difficulty of our tasks; owing to the variable effects of a think-aloud protocol on timing, we do not draw any conclusions on the direct effect of condition on task time.

6.6.2 Usage of TWEAKIT's affordances

Participants used TWEAKIT to view a mean of 226 code outputs ($\sigma = 92.1$). In contrast, participants in the baseline condition viewed a mean of 27.6 code outputs ($\sigma = 15.0$). This difference was significant using a t-test ($t(13) = 7.87, p < .001$).

Participants found TWEAKIT's affordances useful—11/14 reported previewing output as "very useful" and 10/14 reported side-by-side comparison of outputs as "very useful". 9/14 participants responded that TWEAKIT was more useful than the baseline system for understanding code, which was statistically significant using a χ^2 test ($\chi^2(2, n = 14) = 6.14, p < .05$).

6.7 Qualitative Results

An overview of the themes in our study is given in Table 6.1. The following sections elaborate.

6.7.1 Guess-and-check as a desired workflow

All participants made heavy use of guesswork to complete their tasks. Analysts made edits to code by guessing at what kinds of edits might bring them closer to their goal, then checking their results by examining code output. They described their process as "try-and-test" (P11), "shooting from the hip" (P3), and "playing around" with the code (P6, P9). Participants appeared to prefer this workflow for pragmatic reasons. They felt that their workflow wasn't "proper coding" (P6) and that "there's definitely more efficient code [for this task]" (P10), but "if it works, who cares?" (P5). Some analysts mentioned their workflow for these tasks differed from a workflow they learned from programming courses, which stressed a top-down approach of breaking down a task into subgoals, writing pseudocode, then implementing the pseudocode (P10, P11). As one participant stated, "it doesn't matter what I want to do if I can't find the code to do it" (P12). The desire to guess their way to a solution was a defining trait of our participants.

6.7.2 Strategies for understanding unfamiliar code

However, participants' reliance on guess-and-check presented unique challenges for working with code examples. Using the metaphor of finding a path through a maze, participants encountered many forking paths and dead ends because of the large search space of possible code edits. Participants described code examples as "overwhelming" (P8) and they "didn't

Table 6.1. Summary of qualitative participant feedback, organized by themes.

Theme	Description	Representative Examples	Participants
Guess-and-check as a desired workflow	Participants were pragmatic—they sought working results via guess-and-check over clean code.	"I know it's not 'proper coding', but it works so I'm happy." "I'm sure there are better ways of doing this, but let's move on."	P1-P14
Strategies for understanding unfamiliar code	To narrow the search space of possible edits, participants wanted to read, execute, and edit small pieces of code.	"I'm reading this code to look for pieces I can use, but nothing is sticking out to me." "Is there a way to just see this specific thing?"	P1, P3, P4, P6, P8, P9, P10, P11, P13, P14
Using live previews and comparisons as explanations	Participants preferred live previews and comparisons over reading code, treating code outputs as an explanation for what the code did.	"Being able to click on single lines and see the result in real-time is a lot more helpful than just seeing the result [...] you need to know the process." "Being able to compare two outputs let me figure it out."	P1, P2, P3, P4, P5, P6, P7, P9, P10, P11, P12, P13, P14
Increasing confidence through exploration	Participants tied confidence in their ability to use code with their ability to introspect and explore code outputs.	"I wasn't even able to get close [with the baseline], but with TWEAKIT I got a better grasp of the code." "Without TWEAKIT, I didn't know where to even begin [...] I was making random guesses and things sometimes worked but I didn't know why."	P1, P2, P3, P4, P5, P6, P8, P9, P10, P11, P13, P14
Challenges in editing code	Despite forming useful plans, participants struggled to make correct code edits.	"I know what I want to do but I keep breaking the code." "I'm so close, I just don't know how to make the code do what I want."	P1-P14
Enthusiasm for using code in day-to-day work	Participants expressed enthusiasm to leverage code for their personal data tasks.	"With Excel, it's a lot of work to do this task but Python does it instantly." "I just know Python is really powerful. There's a lot more you can do with Python compared to Excel."	P1, P3, P5, P6, P9, P10, P11, P13, P14

know where to start" (P3, P14). To narrow down their options, analysts adopted a variety of strategies centered around understanding small pieces of code at a time. In the absence of live output previews (baseline condition), participants focused on visual attributes of the code—most commonly, they read the code and looked for function names that appeared relevant to their task (P1-P6, P8-P11, P14). One participant even used the syntax highlighting as suggestions for what parts of the code to edit: "the colors [from syntax highlighting] are speaking to me" (P8). Only a few participants used web search to search for function documentation; one explained that "I usually can't understand [the documentation] because I'm not used to [the jargon]" (P12). Instead, participants explained that they picked lines out of the example snippet using "a hunch" (P14), "intuition" (P7), and "a random guess, to be honest" (P5).

Without the live previews, all participants were shown how to run the example snippet yet still seemed to perceive code reading as more useful than running snippets. When probed further, participants explained that the output of an entire snippet did not seem to guide them towards what pieces of the code were most useful: "you need to know the process rather than the outcome" (P3). Some participants specifically manipulated the code to see the results of smaller code pieces. For example, one participant deleted the entire example snippet and incrementally added back lines one by one to see the output of each line (P9), which generated nearly identical outputs to TWEAKIT's live previews for each line. Another participant mentioned that "I know in python you can print variables but that seems like extra work" (P1). As a whole, participants wrestled with the perceived complexity of coding by adopting ad-hoc strategies to understand individual expressions and lines of code.

6.7.3 Using live previews and comparisons as explanations

When available, participants favored using live output and comparison over other strategies to understand code. They described the ability to quickly see outputs of individual expressions as "much easier" (P1, P13), "so convenient" (P4), and "super, super helpful" (P9). Participants also valued the "instantaneous" (P2) and "real-time" (P3) nature of the interactions,

explaining that the live previews reduced friction and encouraged them to examine more lines of code.

Participants treated live output previews and comparisons as explanations of what the code did. To make use of output previews, participants explicitly looked for visual similarities and differences between the output and the code (P1-P7, P9-P14). For example, P7 explained that "the code finally clicked when I saw the value `ABCD` in the code and then I saw that the table columns were also labeled A, B, C, and D." One participant described the previews as "a synopsis" (P4). Another mentioned that the previews allowed them to "analyze and digest" the code (P11). P3 looked up documentation for function calls in the baseline condition but not in TWEAKIT, explaining that "honestly using Google didn't cross my mind, because if there was something in line 6 I didn't know, I could just click on line 5 and compare the outputs." To analysts, live output previews and comparisons made code tweaking easier by making code concrete—instead of guessing at an expression's utility from reading function names, they could directly examine its output.

6.7.4 Increasing confidence through exploration

Participants felt that their understanding of code increased as they previewed and compared code outputs, even though most participants did not complete more tasks using TWEAKIT than using the baseline system. P9 completed one task in each condition yet stated that "I began to understand [the code] better throughout the 20 minutes with TWEAKIT whereas with [the baseline] I just felt more and more confused as time went on". P10 completed one fewer task using TWEAKIT but still felt that "without TWEAKIT, I was blind [...] I couldn't understand any of the [example] code even though I got something to work." Analysts also expressed that having the ability to compare previews would increase their confidence in future coding tasks. P2 explained that comparing previews assured him that he would be able to "figure out" code in the future and wished for more time to complete the tasks in the user study because "I was just starting to understand it". In general, participants felt that live output previews enabled them to

explore and understand more code, which increased their confidence to reuse code.

Analysts also used live previews to validate hypotheses they formed while making guess-and-check edits to the code. P2 and P12 mentioned that previews allowed them to check incremental changes and backtrack quickly if needed. P6 described previews as "a safety blanket" that encouraged him to try out edits without fear that he would introduce data errors. Although TWEAKIT was not intended to have learning outcomes, analysts described the live previews as "a great teaching tool" (P11) and that "it helped me learn Python" (P10). Overall, participants appeared to find preview comparison useful for refining their mental models of their programs.

6.7.5 Challenges in editing code

Although participants valued live previews for understanding code, all participants still encountered challenges with making correct code edits. Participants formed useful plans but struggled with implementation. Most commonly, analysts were unaware of package-specific syntax. For example, P12 tried to use `df['file', 'size']` to select two columns in `pandas` rather than the proper `df[['file', 'size']]`. In this regard, live previews were useful for catching bugs but not for helping analysts find valid edits. Participants also reported live previews as visually "distracting" or "disruptive" when they repeatedly encountered errors (P1, P14). In many instances this barrier completely halted participant progress.

6.7.6 Enthusiasm for using code in day-to-day work

Participants were enthusiastic about leveraging code for tasks they found tedious to complete in their spreadsheet applications. Consistent with our formative interviews, analysts shared that they edit data and formulas manually for bespoke tasks they found formulas ill-suited to address. For example, one analyst wanted to repeat a formula except for every fifth cell of a spreadsheet column (P13). Two other analysts mentioned that tasks in the user study were similar to tasks they performed manually in Excel but found easier using the code in the task (P10, P14). Other analysts explained that code worked better for larger datasets (P5), helped to avoid

common spreadsheet mistakes like skipping a cell (P9), offered versatility through packages (P14), made analyses easier to repeat (P7), and made getting help easier through websites like Stack Overflow (P11). Analysts "couldn't wait" for a future where they could reap the benefits of code directly in their spreadsheets without needing to invest time taking programming courses.

6.8 Discussion

6.8.1 Supporting the workflows of data analysts

For our data analysts, code is one of many tools in the toolbox to get the job done; they would rather complete a task than learn about packages to generate plots in Python. To this end, our analysts used their familiar spreadsheet applications as much as possible and only sought code when they reached tasks they felt were highly difficult to complete with their spreadsheets alone. Although analysts might generally see value in learning programming concepts more deeply, they encounter programs in the context of working on a specific task, and thus demonstrate the "paradox of the active user" [90]—they prefer actions that appear to make short-term progress on their immediate task even when developing conceptual knowledge might bring more long-term benefits. This suggests that tools to support data analysts in opportunistic code reuse should embed themselves within existing workflows and allow analysts to easily see the effects of code on their data.

One theme from our investigation is that analysts encounter bespoke tasks that the designers of their tools did not anticipate. For example, one of our analysts dealt with input data that would change the order of its columns every week, so he found and tweaked a script to extract the data he needed regardless of its position in the input sheet. The near-infinite variance in analysts' task requirements suggests that creating a one-size-fits-all graphical application for data processing is unlikely, as such an application would require the tool designer to correctly predict every possible task a user might need. For this reason, allowing analysts to leverage the versatility of code remains central to our tool design. The variety of tasks that analysts face

suggests that future tools to help analysts make use of code should reify code rather than hide it behind an interface.

6.8.2 Potential use cases in professional work

Although this paper focuses on data analysts who are not professional programmers, we postulate that TWEAKIT's affordances for live previewing and output comparison could benefit professionals as well since programming experts also engage in opportunistic code reuse [85]. To understand whether this hypothesis resonated with professionals, we conducted an informal focus group with product managers ($n = 2$) and software engineers ($n = 3$). They were enthusiastic about using live output comparisons for code reuse and thought these affordances would also help data scientists understand and maintain code written by colleagues. The engineers explained that they also pasted and tweaked code when they worked with unfamiliar software packages, and members of the group shared mockups they had independently created for implementing live output previews in other integrated development environments and computational notebooks. They also pointed out limitations in the TWEAKIT prototype for real-world use. For example, it was not easy to use TWEAKIT to compare the outputs of two large data tables since the user had to scroll up and down to spot differences. To address this, they suggested displaying data visualizations instead of data tables as a code preview. The discussion from the focus group and the existence of other preview-oriented debugging tools for experts like OzCode[3] support the idea that enabling live output comparisons can benefit both novice and expert programmers.

6.8.3 Limitations of TWEAKIT's affordances for code reuse

Our investigation surfaced characteristics of opportunistic code reuse that TWEAKIT did not address for data analysts. For example, when code produced an error, TWEAKIT displayed the error message from the Python interpreter verbatim. Our analysts often could not decipher these error messages since the messages assumed an understanding of programming concepts

[3] https://oz-code.com/blog/net-c-tips/the-complete-linq-debugging-guide

and vocabulary—for example, what the `Key` in `KeyError` means. Although TWEAKIT attempts to correct some errors that arise when code is pasted, future tools might run code on a best-effort basis, similar to languages like Perl and JavaScript, or provide a novice-friendly verbal explanation for errors like Elm.

Our investigation also revealed that analysts sit in a valley of struggle, sometimes manually editing data for days, weeks, and even months before deciding to find and modify a code example. How might we make coding a more desirable pathway for data analysts? One approach is to use program-by-example and program synthesis techniques to generate code examples as part of analysts' workflow [113, 157], then use TWEAKIT's affordances to help analysts tweak and apply these examples elsewhere in their work. Another approach is to bring live output previews directly into the browser when code examples arise from Web search, complementing previous tools [259].

6.8.4 Emergent findings during TWEAKIT design

As we designed TWEAKIT, we added and removed features throughout the prototyping process. One version of TWEAKIT contained a lightweight code versioning feature that enabled users to revert code to an older version and compare outputs between two versions of code. However, in our pilot studies this feature was rarely used and participants did not find this more useful than using the familiar undo functionality in their browser. Another version of TWEAKIT displayed an underline for the first code expression in the snippet that errored. Although we hoped that this feature could help users notice parts of code that needed editing, we found that highlighting errors could also mislead users when a mistake in an early line of code caused an error later on in the code. In our user study, participants P1 and P2 had access to both versioning and error highlighting features before we removed them for the remaining participants, and we did not include observations pertaining to these features in our qualitative analysis.

Analysts incidentally valued TWEAKIT as providing a safe environment for coding. Analysts perceived coding using TWEAKIT as low risk because they could preview data changes

before committing them to their spreadsheets, unlike running a script in VBA that immediately mutated their spreadsheet. Although we did not specifically design TWEAKIT to address this concern, this emergent finding supports the idea that programming tools for data analysts should allow them to edit and execute code without fear of making accidental and irreversible changes to their data.

As a whole, TweakIt makes a familiar live interaction for code introspection useful for data analysts who are not professional programmers. It accomplishes this by embedding itself within existing workflows, placing code outputs directly in the spreadsheet, and applying heuristics to execute code written for a single data value on multiple data values.

6.9 Conclusion

Our formative study uncovered challenges that data analysts face when attempting to opportunistically reuse code. To address this gap, we designed TWEAKIT, a prototype tool to enable analysts to reify the effects of code on their data through live output previews and comparisons. Our user study found that analysts valued TWEAKIT and felt that its affordances empowered them to explore and understand unfamiliar code. Overall, analysts were enthusiastic to use TWEAKIT to transmogrify code.

Chapter 6, in full, is a reprint of the material as it appears in the proceedings of the ACM Conference on Human Factors in Computing Systems (CHI) as TweakIt: Supporting End-User Programmers Who Transmogrify Code. Sam Lau, Sruti Srinivasa Ragavan, Ken Milne, Titus Barik, Advait Sarkar. 2021. The dissertation author was the primary investigator and author of this paper.

Chapter 7

Teaching Data Science by Visualizing Data Table Transformations: Pandas Tutor for Python, Tidy Data Tutor for R, and SQL Tutor

Data science instructors often find it hard to explain to students how a piece of code written in Python, R, or SQL executes in order to transform tabular data. They currently resort to hand-drawing diagrams or making presentation slides to illustrate the semantics of operations such as filtering, sorting, reshaping, pivoting, grouping, and joining. These diagrams are time-consuming to create and do not synchronize with real code or data that students are learning about. In this paper we show that a step-by-step visual representation of tabular data transforms can help instructors to explain these operations. To do so, we created a table visualization library that illustrates the row-, column-, and cell-wise relationships between an operation's input and output tables. On top of this library we built a trio of free web-based visualization tools – Pandas Tutor for Python, Tidy Data Tutor for R tidyverse, and SQL Tutor – that automatically run user code and produce diagrams of how Python/R/SQL transforms data tables step-by-step from input to output. Since launching in Dec 2021, over 61,000 people from over 160 countries have visited our website to try out these tools. For this project, I contributed the design and implementation of Pandas Tutor, while my research collaborators built the Tidy Data and SQL Tutors.

```
dogs
```

(a)

	breed	type	size	longevity
0	Golden Retriever	sporting	medium	12.04
1	Beagle	hound	small	12.30
2	German Shepherd	herding	large	9.73
3	Bulldog	non-sporting	medium	6.29
4	Boxer	working	medium	8.81
5	Great Dane	working	large	6.96
6	Boston Terrier	non-sporting	medium	10.92
7	Siberian Husky	working	medium	12.58
8	Pomeranian	toy	small	9.67
9	French Bulldog	non-sporting	medium	9.00

```
(dogs[dogs['size']=='medium'].sort_values('type')
   .groupby('type').median())
```

(b)

type	longevity
non-sporting	9.000
sporting	12.040
working	10.695

Figure 7.1. It can be hard for data science instructors to explain how code transforms data tables. Here (a) the user loads a table of dogs data and then (b) runs a line of Python pandas code to transform it into an aggregate output table. It is not clear how the output table (b) was derived from the input (a).

7.1 Introduction

The past two decades have seen a sea change in the data ecosystem, and there is more demand than ever for people who have the capabilities to transform and analyze data. As such, there is a growing diversity of data-oriented roles in the workforce – such as data scientists, engineers, analysts, and enthusiasts – as well as the number of people that identify with these roles [60]. In many universities, enrollments in introductory data classes have also ballooned due to interest from across disciplines [147]. For instance, at Columbia University, *W4111 Introduction to Databases* was the 6th largest class on campus in Fall 2021; at UC San Diego, *COGS108 Data Science in Practice* is one of the largest in our department.

Alongside this demand, there is a growing ecosystem of tools centered around dataframe programming APIs [209] in Python and R (e.g., pandas [63], tidyverse [64]), as well as the longstanding SQL ecosystem. At their core, these different languages and APIs all share the same underlying set of relational constructs (e.g., select, project, join, union, aggregation). Yet as data science instructors, we repeatedly encounter two major challenges when teaching data manipulation using one or combinations of these popular tools:

1) Individual code statements can be hard to understand. Dataframe APIs encourage programming idioms that perform multiple operations in a single dense statement [231]. For instance, say that Alice is teaching an introductory course and writes this Python pandas statement to analyze a dataset about dogs (Figure 7.1a):

```
dogs[dogs['size']=='medium'].sort_values('type')
  .groupby('type').median()
```

When she runs that code, it produces the output table in Figure 7.1b. Yet students may find it hard to understand *how* the input table was transformed into the output, because the statement actually performs four operations: filtering, sorting, grouping by a column, and aggregating within-group medians. Note that Alice could tediously break up her code into four separate lines and try to explain the intermediate outputs of each. But that still leaves the problem that grouping and aggregation are individually hard to explain. This problem is not limited to Python. The same concept might be presented in R using the tidyverse [64] API or as a SQL query:

```
dogs %>% filter(size == "medium") %>%  # R tidyverse pipeline
  arrange(type) %>% group_by(type) %>%
  summarize(longevity = median(longevity))


SELECT median(longevity) FROM dogs WHERE size = 'medium'
  ORDER BY size GROUP BY type  # SQL query
```

We note that SQL introduces additional challenges: its syntax is the *reverse* (but not quite) of the execution order, and an optimizer opaquely translates the SQL statement into a tree of physical operations. Understanding the conceptual and physical evaluation of a SQL query is a notorious stumbling block for many students.

Instructors like Alice must now resort to hand-drawing ad-hoc diagrams or making presentation slides to illustrate how these statements work, which can be tedious and time-consuming.

2) It is hard to understand how data science tools differ. Data science courses introduce multiple programming languages and tools (e.g., both pandas and SQL) as each is well-suited for different use cases. Students may expect that a pandas and SQL statement should be semantically equivalent, but be surprised that they are not. How can we allow them to see these subtle semantic differences?

In this study we show that a language-independent step-by-step visual representation of data transforms can address these pedagogical challenges. To do so, we created a JavaScript-based table visualization library that illustrates the row-, column-, and cell-wise relationships between an operation's input and output tables. On top of this library we built a trio of freely-available web tools – Pandas Tutor for Python [177], Tidy Data Tutor for R [172], and SQL Tutor [254] – that automatically produce diagrams of how Python/R/SQL code transforms data tables step-by-step from input to output.

As a concrete example, Figure 7.2 shows one of our tools, Pandas Tutor, running the code from Figure 7.1. The user visits pandastutor.com and writes their Python code. Then Pandas Tutor automatically produces a diagram for each of the four transformation steps: a) the filter shows the correspondence between input rows that were retained by the filter predicate, and draws a box around the attributes used in the predicate; b) sort renders arrows to map input rows to their new positions in the output and draws a box around the sorting attribute; c) groupby draws a box around the grouping attribute and color-encodes rows in each group; d) aggregation shows how rows in each group map to individual output statistics.

Compared to Figure 7.1, the step-by-step diagram created by Pandas Tutor in Figure 7.2 makes it much easier for an instructor to explain to students what is going on behind the scenes to transform the input table to the final output table. Our other two tools – Tidy Data Tutor for R and SQL Tutor – work the same way. *To our knowledge, these tools are the first to render step-by-step diagrams of data table transformations that appear across multiple languages used in teaching introductory data courses.*

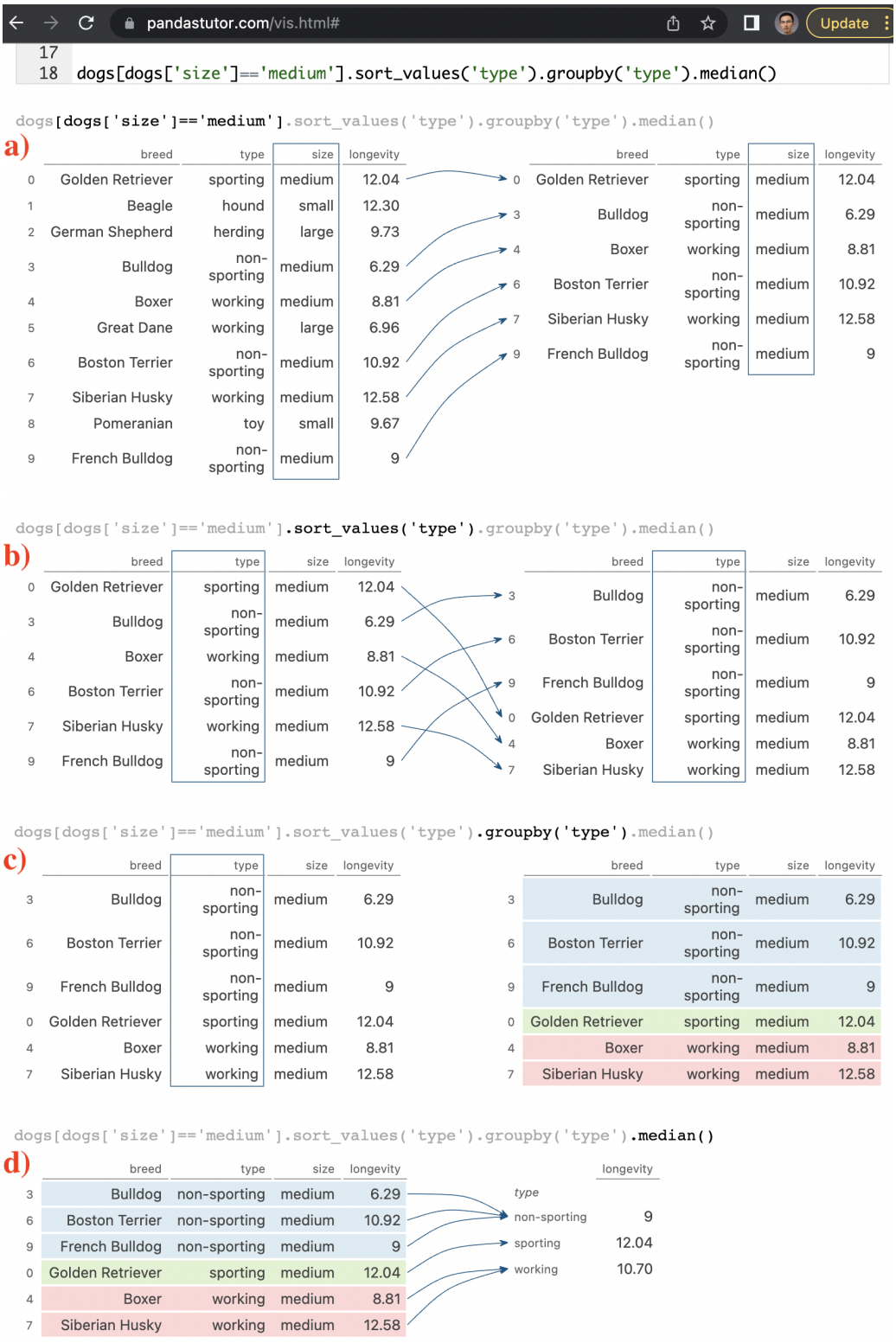


Figure 7.2. Pandas Tutor is a web application that automatically visualizes how Python pandas code transforms data tables step-by-step. This screenshot shows the example from Figure 7.1: a) filtering, b) sorting, c) grouping, d) aggregating. We also created analogous visualization tools for R and SQL.

7.2 System Design and Implementation

Pandas Tutor, Tidy Data Tutor, and SQL Tutor are web-based tools that take the user's code and data as inputs (data can be passed in via a .csv file), runs the code on that data, and produces a set of before-and-after table transformation diagrams (see Figure 7.2).

Each tool has a language-specific backend that takes Python, R, or SQL code and adds precise run-time provenance/lineage tracking to it. For instance, Pandas Tutor uses LibCST [145] to rewrite and instrument calls to Python pandas filtering functions like Figure 7.2a to track the column(s) being filtered on and the rows that were selected. This instrumentation approach means that we must manually[1] add support for each function to track. While doing so may be impractical for supporting, say, the entire pandas or tidyverse libraries with hundreds of API functions, in practice we found that supporting around a dozen functions for each language was sufficient for teaching basic concepts in data science courses.

SQL Tutor's backend uses a Python-based educational DBMS called Databass [253] that is instrumented to track record-level lineage on a per-operator basis based on techniques from Smoke [211]. We run Databass in a web browser using Pyodide [57] to translate to WASM (WebAssembly). It supports SPJA queries, including nested subqueries. In practice, any DBMS that can export its query plan and per-operator lineage info can be used as SQL Tutor's backend.

Each tool's backend runs the user's instrumented code and records the before and after states of the table that is being transformed in each step along with tracked provenance about affected rows/columns/cells. That is why a single line of code like Figure 7.2 can produce four diagrams, since it contains four transformation steps in a pipeline (filtering, sorting, grouping, aggregating). This information then gets sent to the web-based frontend, which uses our core visualization library (see next sections) to display it on-screen.

[1] We did consider automatically inferring provenance at the Python/R interpreter level. But false positives may produce inaccurate diagrams, which we did not want to risk.

7.2.1 Design of Core Table Visualization Library

All of our tools use a common core table visualization library that we wrote in JavaScript. Our main design principles were:

- **Show input-output correspondences** – Our library renders tables along with annotations such as bounding boxes, color highlights, and arrows between rows, columns, and cells (possibly across multiple tables). These annotations are critical for teaching how input and output data correspond, as shown by Figure 7.2. To avoid visual overload, users can mouse hover and click to selectively show/hide annotations.

- **Screenshot-friendly** – We wanted our visualizations to look polished enough so that users can take high-quality screenshots that they can put in presentation slides or lecture notes. We drew aesthetic inspiration from good hand-drawn diagrams that we saw online and from studying how expert instructors created their slides. Unlike related work such as Data Tweening [158] and Datamations [212] we choose not to use animations and instead render only static diagrams.

- **Compact** – What happens when users pass in tables that are too large to fit on the screen? In that case, our library uses heuristics to show the most relevant 12 rows by 8 columns – prioritizing those that are being operated on, truncating long strings, and sampling a few rows from each group (when grouping is used). Users can un-hide cells and long strings by dragging on hidden portions to reveal more data as needed.

- **Embeddable and shareable** – A JavaScript library makes it easy to embed visualizations on any webpage and to share as URLs. It also allowed us to embed Pandas Tutor into Jupyter Notebooks for Python and Tidy Data Tutor into RStudio for R since those data science IDEs are also built in JavaScript.

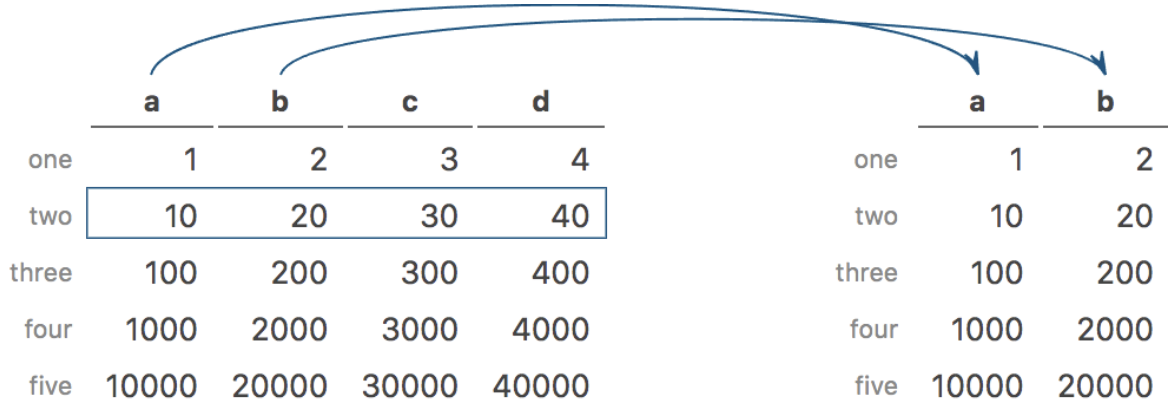
7.2.2 Supported Data Transformation Operators

On top of this core visualization library we implemented a set of interactive diagrams to teach common data transformations. In our experience, these have been sufficient to express the range of operators taught in introductory data science and databases courses. For each of these diagram types, we implemented API hooks into our Python/R/SQL backends so that when users run code in those languages, our tools call the visualization library to render the appropriate diagrams. Here are all of the supported diagram types:

Selecting and filtering: Our tools visualize how operators select individual rows/columns out of a table and optionally filter based on boolean conditions. In Python pandas, this is done via the bracket operator and other operators like `.get()`, `.loc[]`, and `.iloc[]`. In R tidyverse, it is done via the `select()`, `filter()`, and `mutate()` functions; and in SQL via the `SELECT` and `WHERE` clauses.

Figure 7.2a visualizes filtering rows based on a boolean condition (medium-sized dogs). Here is a more complex example of a line of pandas code that selects columns from a dataframe `df` filtered on the values of a specific row – `df.loc[:, df.loc['two'] <= 20]`

This kind of idiomatic pandas code with brackets, colons, `.loc[]`, and boolean conditions can be very hard for beginners to understand. Running it in Pandas Tutor clarifies what it does by showing how the `a` and `b` columns are selected because their values in the row labeled `two` (highlighted with a rectangle) are `<= 20`.



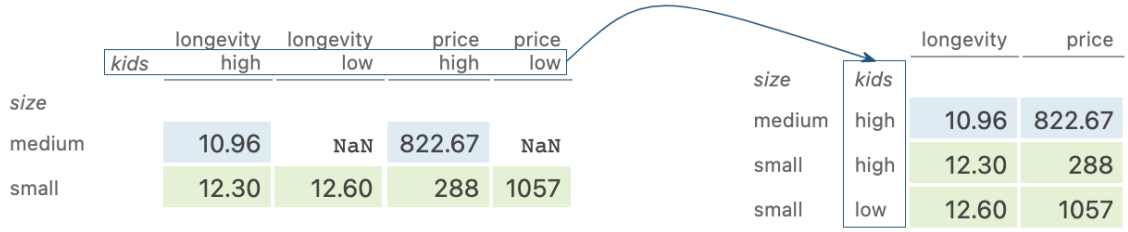
Sorting: Figure 7.2b shows how rows are sorted using the pandas `sort_values()` function. Similar diagrams are rendered for sorting using `arrange()` in R tidyverse and `ORDER BY` in SQL.

Grouping: Figure 7.2c shows the pandas `.groupby()` function creating three groups of rows, each with a unique color. Similar diagrams are rendered for `group_by()` in R tidyverse and `GROUP BY` in SQL. These diagrams use palettes from ColorBrewer [134] with 10 colors, which is usually enough for the small examples used in teaching. A warning appears if the user's code creates more than 10 groups.

Group-wise operations: After grouping, operations can be applied to all rows within each group. For instance, Figure 7.2 shows the pandas `.median()` function applied to numeric values within each group. R tidyverse has functions such as `summarize()`, `arrange()`, and `filter()` that are group-aware and thus run within each group. And SQL has different `SELECT` parameters to accompany `GROUP BY`.

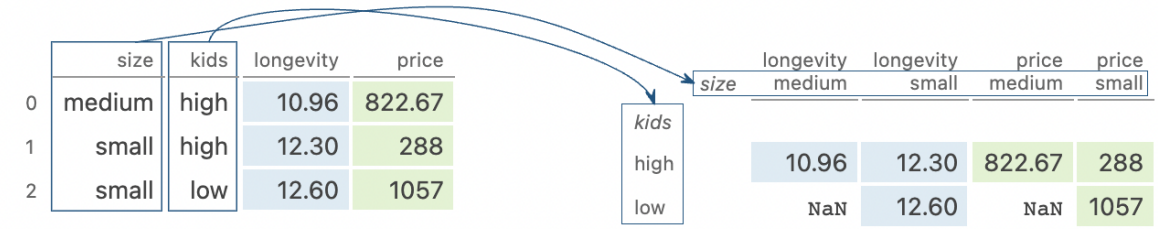
Reshaping: An important precursor to data analysis is *wrangling* [149] (or tidying [248]) raw datasets into a form that is amenable to analysis. This may involve reshaping tables to rearrange the orientations of their rows or columns. In our experience, table reshaping operations can be hard for students to understand because data suddenly moves around in non-intuitive ways.

Our tools use a combination of colors, highlights, and arrows to show how each reshaping operation works. For instance, here the pandas `.stack()` function 'rotates' cells around the `kids` label (called an 'index') and turns it from a column index to a row index:



Here is how the `.pivot()` function rotates a table from a 'long' to 'wide' format by turning `size` from a row to a column index:

```
dogs.pivot(index='kids', columns='size')
```



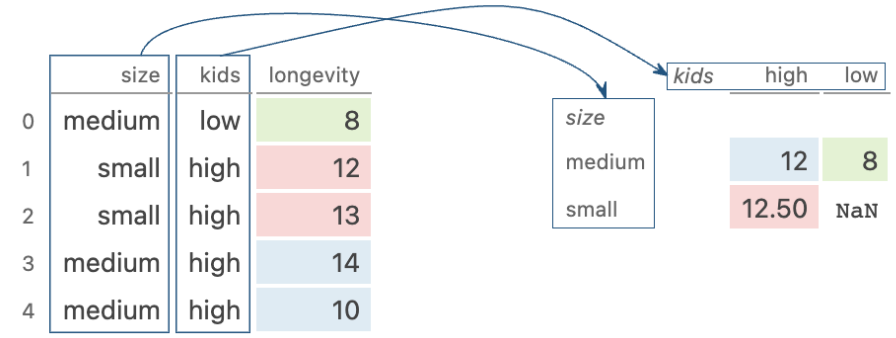
R tidyverse does reshaping via `pivot_longer()`, `pivot_wider()`.

What is especially challenging about teaching reshaping operators is that they can turn metadata in indices (e.g., row or column labels) into regular cell data and vice versa; also, multiple nested layers of indices (called hierarchical indexing [62] in pandas) may get created or destroyed in the process. Our tools visualize all of these intricate interactions between metadata and table cell data.

Pivot tables: The pandas `.pivot_table()` function aggregates data into a pivot table, an operation inspired by spreadsheets.

Here is how Pandas Tutor visualizes an example pivot table call where the `kids` row index pivots into being a column index and the values in the `longevity` column aggregate into a 2x2 cross-tab:

```
dogs.pivot_table(index='size', columns='kids', values='longevity')
```



Similar to reshapings, pivot tables can be hard to understand since passing in different values can result in very different outputs.

Joining two tables: All the above operators transform a single table, but our tools also show joins between two tables. In pandas this occurs via the `.join()` and `.merge()` functions, in

R tidyverse via the `inner_join()`, `left_join()`, `right_join()`, and `full_join()` functions, and in SQL via variants of the `JOIN` clause.

Here Pandas Tutor visualizes a call to `.merge()` to left-join two tables via the `likes` and `breed` columns. The two input tables appear side-by-side while the output table appears below them. The columns to join on are surrounded with rectangles, and each row's color matches the row in the other table that it is being joined with:

```
people.merge(dogs, left_on='likes', right_on='breed', how='left')
```

	name	likes
0	Sam	Samoyed
1	Sam	Dachshund
2	Tina	Beagle
3	Tina	Dachshund
4	Tina	Boxer

	breed	price
0	Beagle	288
1	Samoyed	1162
~~2~~	~~Golden Retriever~~	~~958~~
~~3~~	~~Yorkshire Terrier~~	~~1057~~
4	Dachshund	423

	name	likes	breed	price
0	Sam	Samoyed	Samoyed	1162
1	Sam	Dachshund	Dachshund	423
2	Tina	Beagle	Beagle	288
3	Tina	Dachshund	Dachshund	423
4	Tina	Boxer	NaN	NaN

Since this is a left join, all rows in the left table make it to the output (bottom), but the Golden Retriever and Yorkshire Terrier rows in the right table do not make it since they have no matches in the left table. The user can tweak the code to change the columns to join on or to switch to a right join, inner join, or full outer join; then the visualization will update to illustrate the differences.

7.2.3 Visualizing SQL Query Plans

Python pandas and R tidyverse code statements execute in a linear sequence (i.e., a pipeline), so the corresponding visualizations can also be linear (e.g., Figure 7.2). However, SQL

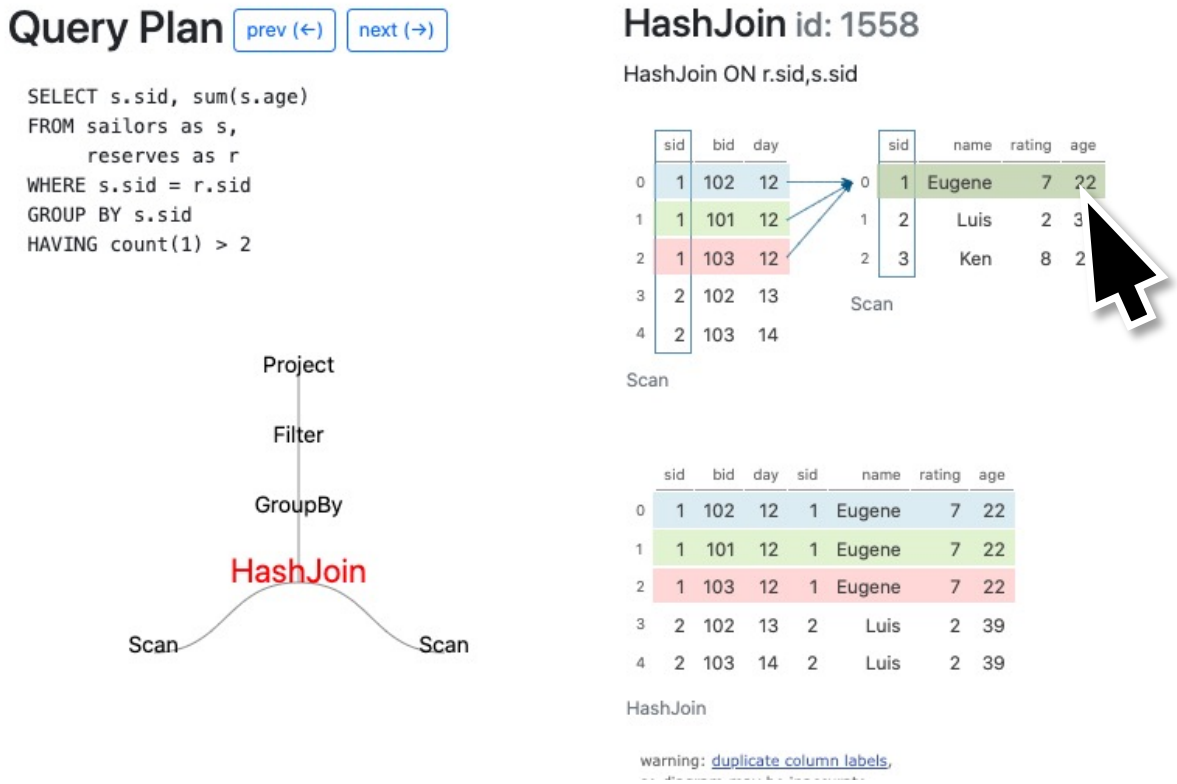


Figure 7.3. SQL Tutor visualizes a SQL statement's query plan (left) and lets the user step through its execution and interactively examine each operator's input and output tables along with their row, column, and cell-level dependencies (right).

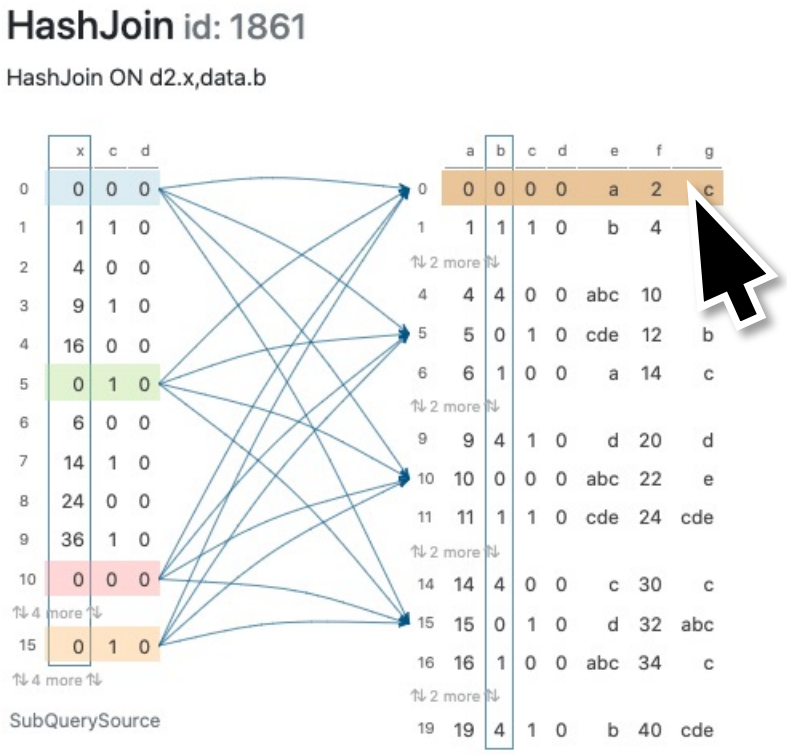


Figure 7.4. SQL Tutor visualizing a many-to-many join when the user hovers over a record in the right relation.

engines compile a single statement into a *tree* of physical operations, so this linear visualization is no longer sufficient. Thus, one challenge of learning SQL is that its declarative nature obscures the correspondence between the order of clauses in its syntax and the operators in the actual execution. In addition, the optimizer chooses from a large space of physical plans. To address this challenge, SQL Tutor [254] visualizes the step-by-step execution of a physical query plan.

Figure 7.3 shows its user interface. In contrast to Pandas or R statements, which form a linear sequence of operations, a physical query plan for a SQL statement forms a hierarchical structure. Thus, SQL Tutor explicitly shows the query plan tree on the left. To zoom into each step, users can click on an operator or step through via a depth-first tree traversal by clicking the previous/next buttons.

The right side of Figure 7.3 lists the operator name, internal id, and the operator's configuration (e.g., `HashJoin ON r.sid, s.sid`). The diagrams are nearly identical to those

for pandas and R since all tools use the same visualization library. For instance, this example joins the sailors (left) and reserves (right) relations on `sid`. The user is hovering over the first 'reserves' record, which filters the visualization annotations to the sailors that join with this record. The arrows show the sailor rows that join with `Eugene`, and use the same colors as their corresponding output records in the bottom table. Vertical bounding boxes denote the join keys in both relations.

For many-to-many joins, all matching records on both sides are highlighted. For instance, Figure 7.4 shows that the user has highlighted the first record on the right relation; the tool then colors all join candidates in the left relation *and* draw arrows from those candidates to their additional matches in the right relation.

7.3 Deployment and Preliminary Impact

We deployed `pandastutor.com` and `tidydatatutor.com` publicly in December 2021 and spread the word to fellow instructors via our professional and social networks. As a preliminary indicator of impact, Google Analytics shows that around 45,000 unique users visited `pandastutor.com` and 16,000 visited `tidydatatutor.com` between Dec 2021 and the end of Apr 2023 (17 months). These visitors came from over 160 countries spanning most of the world (see Figure 7.5). SQL Tutor is also available on the web [254], but we have not publicized it as widely since it is still under development.

Although these website visitor numbers from around the world are personally exciting to us, we acknowledge that they are not a substitute for conducting a formal evaluation to study what instructors do with these tools and whether it helps their students to learn better. As some early steps here, we collected the following anecdotes from our personal experiences teaching with these tools:

I (Sam Lau) used Pandas Tutor extensively while teaching DSC 10: Principles of Data Science at UC San Diego during Summer 2022. I observed that integrating Pandas Tutor

Figure 7.5. Google Analytics data showing the approximate number of people per country who visited the Pandas Tutor and Tidy Data Tutor websites from Dec 2021 to April 2023.

within Jupyter Notebooks was essential for student adoption since the course materials were all presented within Jupyter. I made custom stylistic adjustments such as font sizes and color contrast to make the visualizations more legible when presenting on a projector or via remote screen-share. Student feedback was positive, and I observed students using the tool during class to visualize more complex functions like groupby and merge. Other instructors of DSC 10 and also DSC 80: Practice and Application of Data Science at UC San Diego are now starting to use Pandas Tutor in class.

One limitation I encountered was the fact that Pandas Tutor cannot handle datasets larger than a few megabytes. While this is not a problem for demonstrating small in-class examples, homework assignments often involve larger datasets (e.g., tens to hundreds of MB) so students could not use Pandas Tutor to debug their homework code. Optimizing performance on larger datasets is one avenue for future work. (Tidy Data Tutor has the same limitation.)

More broadly, the Pandas Tutor and Tidy Data Tutor websites include a sign-up form to join a private instructor mailing list to receive updates on tool development. So far, over 90 data science instructors have joined the list and some wrote a note on their form about their interest in trying these tools in their classes. We hope to follow up with them to see whether they have used

it.

Eugene Wu, a research collaborator who designed SQL Tutor, released an early version of the tool in his section of Columbia University's Introduction to Databases course in Fall 2022. Students in the course used it to visualize over 300 queries, with spikes that coincided with exams. One limitation of the current implementation is its reliance on a custom and incomplete SQL parser. Our plan is to replace this with an instrumented version of DuckDB [214] that captures the lineage needed to visualize the query execution. DuckDB is increasingly used in industry and data courses due to its easy-to-install nature, ability to run in the browser thanks to WASM compilation, and fairly complete feature set. Once we make this transition, SQL Tutor will be released and disseminated more widely.

7.4 Conclusion

Data science instructors find it challenging to explain to beginners how exactly Python, R, and SQL code transforms tabular data. To help overcome this challenge, we created a set of freely-available web-based tools that visualize data table transformations step-by-step in three popular languages: Pandas Tutor for Python [177], Tidy Data Tutor for R [172], and SQL Tutor [254]. We are now working on integrating these tools more deeply into data science courses.

Chapter 7, in full, is a reprint of the material as it appears in the proceedings of the International Workshop on Data Systems Education (DataEd) as Teaching Data Science by Visualizing Data Table Transformations: Pandas Tutor for Python, Tidy Data Tutor for R, and SQL Tutor. Sam Lau, Sean Kross, Eugene Wu, Philip J. Guo 2023. The dissertation author was a primary investigator and author of this paper.

Chapter 8

Conclusion

This concluding chapter serves as a synthesis of the entire dissertation, starting by reflecting on the work presented in light of my thesis statement. Then, it presents a research agenda and vision for the future.

8.1 Summary of Findings

My thesis is that instructor-centered approaches enable the design of tools that directly support teaching at scale. To this end, the first portion of this dissertation begins by understanding instructors' goals and desires as they perform their teaching work, and by understanding the strengths and weaknesses of the tools they use to execute on their goals. One recurring challenge that emerged from this needfinding is that instructors often wish to present visual representations of code, yet must currently create these diagrams manually. To address this need, the second portion of this dissertation presents software tools that apply program analysis and visualization techniques to automatically create interactive, visual representations of programs as they execute.

In particular, this thesis makes two claims:

C1. Instructor-centered approaches can reveal previously unmet needs of instructors as they collaborate with their software tools.

C2. Using instructor needs as design goals enables tools that directly support teaching.

Next, I present the evidence from this dissertation that supports each of these claims.

Claim 1: Instructor-centered approaches can reveal previously unmet needs of instructors as they collaborate with their software tools.

The studies in this dissertation provide examples of instructor-centered approaches that subsequently highlight previously unmet needs. Chapter 3 highlights four challenges that instructors face when updating large technical courses: intricate dependencies, variants of materials, ad-hoc software infrastructure, and difficulty reusing third-party software. For example, one course's infrastructure grew from a single file of LaTeX macros to a sprawling collection of scripts that stitched together multiple programming languages (LaTeX, Python, and bash) and software tools. As a whole, this study finds that instructors not only face challenges in creating teaching plans but also in *executing on their plans*.

Instructors have the option to reuse existing course materials rather than make large-scale updates, which would result in less workload. However, our studies show that instructors rarely choose to do this. Chapter 4 provides one explanation: instructors often teach courses in collaboration with other instructors with different backgrounds and perspectives. Data science courses are particularly interdisciplinary, which motivates instructors to discuss and iterate on course materials in order to improve the quality of teaching. The insight of Chapter 4 is that instructors consistently seek to update their course materials, despite the work required to do so. This finding presents a design opportunity: if tools could better support instructors by minimizing the logistical burdens presented in Chapter 3, instructors could more easily iterate and improve their course materials.

To better understand the tools that instructors use, Chapter 5 surveys computational notebooks, a set of tools that have become particularly prominent in teaching programming and data science in recent years – when I started my PhD in 2018, notebook systems were generally considered a novelty. At the time of this writing five years later, notebook systems have become a standard tool for doing and teaching data science, machine learning, and programming. This study defines computational notebooks for the first time in the research literature, and presents a

design space to characterize 60 different notebook systems. In the context of instruction, this survey highlights design opportunities for notebook systems that can be better suited for teaching, for example by enforcing execution order, allowing for collaboration, and by supporting multiple presentation modalities (e.g. as a script, or as a slideshow).

Claim 2: Using instructor needs as design goals enables tools that directly support teaching.

Chapter 6 contributes the design and implementation of TWEAKIT, a tool that supports end-user programmers as they reuse code examples. In the context of teaching, learners are presented with many code examples across their course materials and need to understand and modify these examples for their work. One inspiration for TWEAKIT was observing that instructors often help students understand complicated code snippets by showing students the outputs of small pieces of these snippets at a time. However, instructors of large technical courses lack the ability to help every individual learner understand their code examples. From this perspective, the live previews that TWEAKIT provides its users can be seen as replicating part of what an expert instructor would seek to do while explaining code to a learner. TWEAKIT was shown to encourage exploration and increase confidence for Python novices, making it promising for classroom use.

Chapter 7 contributes Pandas Tutor, a tool that automatically draws diagrams to explain `pandas` code commonly used in data science courses. The tool was directly inspired by observing that instructors spend significant amounts of time creating and editing diagrams for their learning materials. However, the diagramming tools require instructors to create diagrams by hand since the tools lack the ability to link diagrams with code or data. Pandas Tutor enables instructors to rapidly create diagrams directly from their code snippets without any manual drawing, which supports teaching by enabling instructors to spend more time finding good examples to show in class rather than editing pixels in a diagramming tool.

8.2 Future Research Plans

By continuing to pursue my research interests, I can not only advance state-of-the-art tools for program visualization but also directly help students. The continued use of Pandas Tutor provides evidence that people find the tool provides usable and useful program visualizations. Future in-lab and in-classroom evaluations of Pandas Tutor could make these statements more precise – for example, do instructors who use Pandas Tutor experiment or present more example diagrams for teaching? Does Pandas Tutor help learners debug code more quickly and effectively? Answering these questions can not only improve the design of Pandas Tutor to make it more useful for instructors but also provide broad recommendations for using program visualization tools in teaching programming and data science.

Animated and interactive content hold promise for instruction not only because of potential benefits for learning [73, 79] but also because displaying high-resolution content with real-time interactivity is now feasible from an engineering standpoint. However, instructors rarely create animated or interactive content for teaching because their software tools present usability challenges even for professional programmers. Program visualization techniques, like the ones implemented in Pandas Tutor, provide a possible path forward. If static diagrams can be generated directly from code that instructors already write, it might also be feasible to layer interactions and animations on top of diagrams by leveraging usable programming techniques like programming-by-demonstration [92].

At a high level, tools like Pandas Tutor are capable of generating a subset of visualizations from a larger design space of all possible program visualizations. What are the primary dimensions of this design space? And is it possible to distill these dimensions into a succinct grammar of program visualizations? I see an analogy to the field of data visualization, where a grammar of data visualizations [251] enables software tools that provide more usable ways to design and implement interactive visualizations [224]. If such a grammar also existed for program visualizations, we could perform experiments to produce general guidelines for any

program visualization, not just the ones that Pandas Tutor is currently capable of producing. Having a grammar of program visualizations would enable the research community to work towards a science of program visualizations.

8.3 Concluding Remarks

I envision a future where instructors can fluently create, revise, and present their course materials in collaboration with their software tools. The work of this dissertation provides a path towards this goal, through instructor-centered needfinding studies that uncover instructor needs and novel tool designs that address these needs. I believe we are in the early stages of applying instructor-centered design towards the challenges of teaching programming and data science courses. I also believe that solving these challenges will provide great benefits for instructors, by enabling instructors to spend more time on pedagogy rather than course logistics and by providing instructors new ways of presenting ideas visually. By continuing the work that this dissertation starts, I seek to make this vision a reality for instructors and learners everywhere.

Bibliography

[1] Airtable: Part spreadsheet, part database, and entirely flexible, teams use Airtable to organize their work, their way. https://airtable.com/. Accessed: 2020-02-01.

[2] BeakerX. http://beakerx.com. Accessed: 2020-02-01.

[3] The Binder Project. https://mybinder.org/. Accessed: 2020-02-01.

[4] Carbide Alpha — Buggy But Live! https://alpha.trycarbide.com/. Accessed: 2020-02-01.

[5] CoCalc - Collaborative Calculation and Data Science. https://cocalc.com/. Accessed: 2020-02-01.

[6] Coda — A new doc for teams. https://coda.io/welcome. Accessed: 2020-02-01.

[7] Code Ocean — Professional tools for researchers. https://codeocean.com/. Accessed: 2020-02-01.

[8] Databricks Collaborative Notebooks. https://databricks.com/product/collaborative-notebooks. Accessed: 2020-02-01.

[9] Datalore. https://datalore.io/. Accessed: 2020-02-01.

[10] Deepnote - Data science notebook for teams. https://www.deepnote.com/. Accessed: 2020-02-01.

[11] Dynamicland: Our mission is to incubate a humane dynamic medium whose full power is accessible to all people. https://dynamicland.org/. Accessed: 2020-02-01.

[12] Eve. http://witheve.com/. Accessed: 2020-02-01.

[13] Explorable explanations, a hub for learning through play! https://explorabl.es/. Accessed: 2020-02-01.

[14] Gigantum - Build it. Move it. Share it. https://gigantum.com/. Accessed: 2020-02-01.

[15] Google Colaboratory. https://colab.research.google.com. Accessed: 2020-02-01.

[16] Google Realtime API Deprecation. https://developers.google.com/realtime/deprecation. Accessed: 2020-02-01.

[17] Hydrogen: run code interactively, inspect data, and plot. all the power of jupyter kernels, inside your favorite text editor. https://atom.io/packages/hydrogen. Accessed: 2020-02-01.

[18] IBM Watson Studio. https://www.ibm.com/uk-en/cloud/watson-studio. Accessed: 2020-02-01.

[19] Iodide. https://alpha.iodide.io/. Accessed: 2020-02-01.

[20] Jupyter Notebook 2015 UX Survey Results. https://github.com/jupyter/surveys/blob/master/surveys/2015-12-notebook-ux/analysis/report_dashboard.ipynb. Accessed: 2020-02-01.

[21] JupyterCon: The Official Jupyter Conference. https://conferences.oreilly.com/jupyter/jupny. Accessed: 2020-02-01.

[22] jupytext: Jupyter Notebooks as Markdown Documents, Julia, Python or R scripts. https://github.com/mwouts/jupytext. Accessed: 2020-02-01.

[23] Kaggle Kernels. https://www.kaggle.com/kernels. Accessed: 2020-02-01.

[24] Kogence. https://kogence.com/app/docs/JupyterNotebook. Accessed: 2020-02-01.

[25] Kyso — Data Analytics Knowledge Hub. https://kyso.io/. Accessed: 2020-02-01.

[26] Leisure. https://github.com/zot/Leisure. Accessed: 2020-02-01.

[27] Livebook. https://github.com/inkandswitch/livebook. Accessed: 2020-02-01.

[28] Machine learning research should be clear, dynamic and vivid. distill is here to help. https://distill.pub/about/. Accessed: 2020-02-01.

[29] Maple: The essential tool for mathematics. https://www.maplesoft.com/products/maple/. Accessed: 2020-02-01.

[30] MATLAB Live Editor: Create scripts that combine code, output, and formatted text in an executable notebook. https://www.mathworks.com/products/matlab/live-editor.html. Accessed: 2020-02-01.

[31] Microsoft Azure Notebooks. https://notebooks.azure.com/. Accessed: 2020-02-01.

[32] Mode Notebooks. https://mode.com/notebooks/. Accessed: 2020-02-01.

[33] Nbdev. http://nbdev.fast.ai/. Accessed: 2020-02-01.

[34] Nodebook from Stitch Fix. https://github.com/stitchfix/nodebook. Accessed: 2020-02-01.

[35] The notebook for reproducible research — Nextjournal. https://nextjournal.com/. Accessed: 2020-02-01.

[36] Notion: all-in-one workspace. https://www.notion.so/. Accessed: 2020-02-01.

[37] nteract: building the future of interactive computing. https://nteract.io/. Accessed: 2020-02-01.

[38] Observable. https://observablehq.com/. Accessed: 2020-02-01.

[39] Polynote — The polyglot Scala notebook. https://polynote.org/. Accessed: 2020-02-01.

[40] Quantopian: The Place For Learning Quant Finance. https://www.quantopian.com/. Accessed: 2020-02-01.

[41] RMarkdown. https://rmarkdown.rstudio.com/. Accessed: 2020-02-01.

[42] RStudio is an integrated development environment (IDE) for R. https://rstudio.com/products/rstudio/. Accessed: 2020-02-01.

[43] RunKit is a Node playground in your browser. https://runkit.com/. Accessed: 2020-05-01.

[44] Shiny is an R package that makes it easy to build interactive web apps straight from R. https://shiny.rstudio.com/. Accessed: 2020-02-01.

[45] Spark Notebook. https://github.com/spark-notebook/spark-notebook. Accessed: 2020-02-01.

[46] Spyder Website. https://www.spyder-ide.org/. Accessed: 2020-02-01.

[47] Streamlit — The fastest way to build custom ML tools. https://www.streamlit.io/. Accessed: 2020-02-01.

[48] The IPython notebook: a historical retrospective. http://blog.fperez.org/2012/01/ipython-notebook-historical.html. Accessed: 2020-02-01.

[49] ThebeLab: turning static html pages into live documents. https://github.com/minrk/thebelab. Accessed: 2020-02-01.

[50] Voila. https://github.com/voila-dashboards/voila. Accessed: 2020-02-01.

[51] Wikipedia: Notebook interface. https://en.wikipedia.org/wiki/Notebookinterface. Accessed: 2020-02-01.

[52] Wolfram Notebooks: Environment for Technical Workflows. http://www.wolfram.com/notebooks/. Accessed: 2020-02-01.

[53] Working with Jupyter Notebooks in Visual Studio Code. https://code.visualstudio.com/docs/python/jupyter-support. Accessed: 2020-02-01.

[54] Zeppelin. https://zeppelin.apache.org/. Accessed: 2020-02-01.

[55] Jupyter receives the ACM Software System Award. https://blog.jupyter.org/jupyter-receives-the-acm-software-system-award-d433b0dfe3a2, May 2018. Accessed: 2020-02-01.

[56] What's New In Python 3.7 — Python 3.9.6 documentation. https://docs.python.org/3/whatsnew/3.7.html#porting-to-python-37, 2018.

[57] Pyodide is a Python distribution for the browser and Node.js based on WebAssembly. https://pyodide.org/, 2019. Accessed: 2023-02-20.

[58] Dewey Defeats Truman. *Wikipedia*, July 2021.

[59] Project Gutenberg. https://www.gutenberg.org/, 2021.

[60] 11 Types of Data Science Jobs (With Responsibilities). https://www.indeed.com/career-advice/finding-a-job/types-of-data-science-jobs, 2022. Accessed: 2023-02-20.

[61] Otter-Grader Documentation — Otter-Grader documentation. https://otter-grader.readthedocs.io/en/latest/, 2022.

[62] pandas - multiindex / advanced indexing. https://pandas.pydata.org/docs/user_guide/advanced.html, 2023. Accessed: 2023-02-20.

[63] pandas - python data analysis library. https://pandas.pydata.org/, 2023. Accessed: 2023-02-20.

[64] Tidyverse: R packages for data science. https://www.tidyverse.org/, 2023. Accessed: 2023-02-20.

[65] Joel C. Adams. Creating a balanced data science program. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, pages 185–191, 2020.

[66] Genevera I. Allen. Experiential Learning in Data Science: Developing an Interdisciplinary, Client-Sponsored Capstone Program. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, pages 516–522, New York, NY, USA, March 2021. Association for Computing Machinery.

[67] Paul Anderson, James Bowring, Renée McCauley, George Pothering, and Christopher Starr. An undergraduate degree in data science: Curriculum and a decade of implementation experience. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, pages 145–150, New York, NY, USA, March 2014. Association for Computing Machinery.

[68] Computing Research Association. Generation cs: Computer science undergraduate enrollments surge since 2006. https://cra.org/data/generation-cs/, 2017. Accessed: 2021-01-15.

[69] Sriram Karthik Badam, Andreas Mathisen, Roman Rädle, Clemens Nylandsted Klokmose, and Niklas Elmqvist. Vistrates: A component model for ubiquitous analytics. *IEEE Trans. Vis. Comput. Graph.*, 25(1):586–596, 2019.

[70] Austin Cory Bart, Dennis Kafura, Clifford A. Shaffer, and Eli Tilevich. Reconciling the Promise and Pragmatics of Enhancing Computing Pedagogy with Data Science. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, SIGCSE '18, pages 1029–1034, New York, NY, USA, February 2018. Association for Computing Machinery.

[71] Soumya Basu, Albert Wu, Brian Hou, and John DeNero. Problems before solutions: Automated problem clarification at scale. In *Proceedings of the Second (2015) ACM Conference on Learning @ Scale*, L@S '15, page 205–213, New York, NY, USA, 2015. Association for Computing Machinery.

[72] Ben Baumer. A data science course for undergraduates: Thinking with data. *The American Statistician*, 69(4):334–342, 2015.

[73] Mireille Betrancourt. The animation and interactivity principles in multimedia learning. *The Cambridge handbook of multimedia learning*, pages 287–296, 2005.

[74] Eric A. Bier, Maureen C. Stone, Ken Pier, William Buxton, and Tony D. DeRose. Toolglass and magic lenses: The see-through interface. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '93, page 73–80, New York, NY, USA, 1993. Association for Computing Machinery.

[75] Ismail Bile Hassan, Thanaa Ghanem, David Jacobson, Simon Jin, Katherine Johnson, Dalia Sulieman, and Wei Wei. Data Science Curriculum Design: A Case Study. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, pages 529–534, New York, NY, USA, March 2021. Association for Computing Machinery.

[76] Andrew P Black, Oscar Nierstrasz, Stéphane Ducasse, and Damien Pollet. *Pharo by example*. Lulu. com, 2010.

[77] Avrim Blum, John Hopcroft, and Ravindran Kannan. *Foundations of Data Science*. Cambridge University Press, 2020.

[78] Joshua Blumenstock, Gabriel Cadamuro, and Robert On. Predicting poverty and wealth from mobile phone metadata. *Science*, 350(6264):1073–1076, November 2015.

[79] Eliza Bobek and Barbara Tversky. Creating visual explanations improves learning. *Cognitive Research: Principles and Implications*, 1(1), December 2016.

[80] Tracey Booth and Simone Stumpf. End-user experiences of visual and textual programming environments for arduino. In *International symposium on end user development*, pages 25–39. Springer, 2013.

[81] Marcel Borowski, Roman Rädle, and Clemens N. Klokmose. Codestrate packages: An alternative to "one-size-fits-all" software. In *Extended Abstracts of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI EA '18, New York, NY, USA, 2018. Association for Computing Machinery.

[82] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D3 data-driven documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, December 2011.

[83] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R Klemmer. Example-centric programming: integrating web search into the development environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 513–522, 2010.

[84] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R. Klemmer. Example-centric programming: Integrating web search into the development environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 513–522, New York, NY, USA, 2010. ACM.

[85] Joel Brandt, Philip J Guo, Joel Lewenstein, and Scott R Klemmer. Opportunistic programming: How rapid ideation and prototyping occur in practice. In *Proceedings of the 4th international workshop on End-user software engineering*, pages 1–5, 2008.

[86] Leo Breiman. Statistical modeling: The two cultures (with comments and a rejoinder by the author). *Statistical science*, 16(3):199–231, 2001.

[87] Thomas C. Bressoud and Gavin Thomas. A Novel Course in Data Systems with Minimal Prerequisites. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, SIGCSE '19, pages 15–21, New York, NY, USA, February 2019. Association for Computing Machinery.

[88] Julia Brich, Marcel Walch, Michael Rietzler, Michael Weber, and Florian Schaub. Exploring end user programming needs in home automation. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 24(2):1–35, 2017.

[89] Margaret Burnett, Simone Stumpf, Jamie Macbeth, Stephann Makri, Laura Beckwith, Irwin Kwan, Anicia Peters, and William Jernigan. Gendermag: A method for evaluating software's gender inclusiveness. *Interacting with Computers*, 28(6):760–787, 2016.

[90] John M Carroll and Mary Beth Rosson. Paradox of the active user. In *Interfacing thought: Cognitive aspects of human-computer interaction*, pages 80–111. 1987.

[91] Ned Chapin, Joanne E. Hale, Khaled Md. Kham, Juan F. Ramil, and Wui-Gee Tan. Types of software evolution and software maintenance. *Journal of Software Maintenance*, 13(1):3–30, January 2001.

[92] Sarah Elizabeth Chasins. *Democratizing Web Automation: Programming for Social Scientists and Other Domain Experts*. PhD thesis, UC Berkeley, 2019.

[93] Souti Chattopadhyay, Ishita Prasad, Austin Z. Henley, Anita Sarma, and Titus Barik. What's Wrong with Computational Notebooks? Pain Points, Needs, and Design Opportunities. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–12.

[94] Souti Chattopadhyay, Ishita Prasad, Austin Z. Henley, Anita Sarma, and Titus Barik. What's wrong with computational notebooks? pain points, needs, and design opportunities. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI '20, New York, NY, USA, 2020. ACM.

[95] Hao-Fei Cheng, Bowen Yu, Siwei Fu, Jian Zhao, Brent Hecht, Joseph Konstan, Loren Terveen, Svetlana Yarosh, and Haiyi Zhu. Teaching ui design at global scales: A case study of the design of collaborative capstone projects for moocs. In *Proceedings of the Sixth (2019) ACM Conference on Learning @ Scale*, L@S '19, New York, NY, USA, 2019. Association for Computing Machinery.

[96] Raj Chetty, Nathaniel Hendren, Patrick Kline, and Emmanuel Saez. Where is the land of opportunity? The geography of intergenerational mobility in the United States. *The Quarterly Journal of Economics*, 129(4):1553–1623, 2014.

[97] Anna T. Cianciolo and Robert J. Sternberg. *Practical Intelligence and Tacit Knowledge: An Ecological View of Expertise*, page 770–792. Cambridge Handbooks in Psychology. Cambridge University Press, 2 edition, 2018.

[98] William S. Cleveland. Data science: An action plan for expanding the technical areas of the field of statistics. *International statistical review*, 69(1):21–26, 2001.

[99] Committee on Envisioning the Data Science Discipline: The Undergraduate Perspective, Computer Science and Telecommunications Board, Board on Mathematical Sciences and Analytics, Committee on Applied and Theoretical Statistics, Division on Engineering and Physical Sciences, Board on Science Education, Division of Behavioral and Social Sciences and Education, and National Academies of Sciences, Engineering, and Medicine. *Envisioning the Data Science Discipline: The Undergraduate Perspective: Interim Report*. National Academies Press, Washington, D.C., March 2018.

[100] Matthew Conlen and Jeffrey Heer. Idyll: A markup language for authoring and publishing interactive articles on the web. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, UIST '18, page 977–989, New York, NY, USA, 2018. Association for Computing Machinery.

[101] Juliet M. Corbin and Anselm L. Strauss. *Basics of qualitative research: techniques and procedures for developing grounded theory*. SAGE Publications, Inc., Thousand Oaks, Calif, 2008.

[102] National Research Council. *Nonresponse in Social Science Surveys: A Research Agenda*. National Academies Press, 2013.

[103] Russ Cox. Surviving software dependencies. *Commun. ACM*, 62(9):36–43, August 2019.

[104] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. Social coding in github: Transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*, CSCW '12, page 1277–1286, New York, NY, USA, 2012. Association for Computing Machinery.

[105] Sarah Dahlby Albright, Titus H. Klinge, and Samuel A. Rebelsky. A Functional Approach to Data Science in CS1. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, SIGCSE '18, pages 1035–1040, New York, NY, USA, February 2018. Association for Computing Machinery.

[106] Tuan Nhon Dang, Leland Wilkinson, and Anushka Anand. Stacking graphic elements to avoid over-plotting. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):1044–1052, 2010.

[107] Thomas H. Davenport and D. J. Patil. Data Scientist: The Sexiest Job of the 21st Century. *Harvard Business Review*, October 2012.

[108] Richard D. De Veaux, Mahesh Agarwal, Maia Averett, Benjamin S. Baumer, Andrew Bray, Thomas C. Bressoud, Lance Bryant, Lei Z. Cheng, Amanda Francis, and Robert Gould. Curriculum guidelines for undergraduate programs in data science. *Annual Review of Statistics and Its Application*, 4:15–30, 2017.

[109] Robert DeLine, Danyel Fisher, Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, James Terwilliger, and John Wernsing. Tempe: Live scripting for live data. VL/HCC '15, 10 2015.

[110] John DeNero. Data 8. http://data8.org/.

[111] John DeNero. Data 8. http://data8.org/, 2021.

[112] Ian Drosos, Titus Barik, Philip J. Guo, Robert DeLine, and Sumit Gulwani. Wrex: A unifed programming-by-example interaction for synthesizing readable code for data scientists. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI '20, New York, NY, USA, 2020. ACM.

[113] Ian Drosos, Titus Barik, Philip J Guo, Robert DeLine, and Sumit Gulwani. Wrex: A unified programming-by-example interaction for synthesizing readable code for data scientists. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–12, 2020.

[114] James R Eagan and John T Stasko. The buzz: supporting user tailorability in awareness applications. In *Proceedings of the sigchi conference on human factors in computing systems*, pages 1729–1738, 2008.

[115] Nadia Eghbal. *Working in Public: The Making and Maintenance of Open Source Software*. Stripe Press, 2020.

[116] Alan Fekete, Judy Kay, and Uwe Röhm. A Data-centric Computing Curriculum for a Data Science Major. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, pages 865–871, New York, NY, USA, March 2021. Association for Computing Machinery.

[117] Stuart I. Feldman. Make–a program for maintaining computer programs. *Software: Practice and experience*, 9(4):255–265, 1979.

[118] Joseph Feliciano, Margaret-Anne Storey, and Alexey Zagalsky. Student experiences using github in software engineering courses: A case study. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 422–431, 2016.

[119] Danyel Fisher, Badrish Chandramouli, Robert DeLine, Jonathan Goldstein, Andrei Aron, Mike Barnett, John Platt, James Terwilliger, and John Wernsing. Tempe: An interactive data science environment for exploration of temporal and streaming data. Technical report, November 2014.

[120] George W. Fitzmaurice, Hiroshi Ishii, and William A. S. Buxton. Bricks: Laying the foundations for graspable user interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '95, page 442–449, USA, 1995. ACM Press/Addison-Wesley Publishing Co.

[121] Bjarke Fog and Clemens Nylandsted Klokmose. Mapping the landscape of literate computing. In *Proceedings of the 30th Annual Workshop of the Psychology of Programming Interest Group*, PPIG, 2019.

[122] Dan Frank. Reproducible research: Stripe's approach to data science. https://stripe.com/blog/reproducible-research. Accessed: 2020-02-01.

[123] Wai-Tat Fu and Wayne D Gray. Resolving the paradox of the active user: Stable suboptimal performance in interactive tasks. *Cognitive science*, 28(6):901–935, 2004.

[124] R. Stuart Geiger, Nelle Varoquaux, Charlotte Mazel-Cabasse, and Chris Holdgraf. The types, roles, and practices of documentation in data analytics open source software libraries. *Computer Supported Cooperative Work (CSCW)*, 27(3):767–802, 2018.

[125] Michael W. Godfrey and Daniel M. German. The past, present, and future of software evolution. In *2008 Frontiers of Software Maintenance*, pages 129–138, 2008.

[126] T. R. G. Green. Cognitive dimensions of notations. In *Proceedings of the Fifth Conference of the British Computer Society, Human-Computer Interaction Specialist Group on People and Computers V*, page 443–460, USA, 1990. Cambridge University Press.

[127] Joel Grus. I Don't Like Notebooks - Joel Grus - #JupyterCon 2018. https://docs.google.com/presentation/d/1n2RlMdmv1p25Xy5thJ UhkKGvjtV-dkAIsUXP-AL4ffI/edit?urp=gmail_link &usp=embed_facebook, 2018. Accessed: 2020-02-01.

[128] Philip Guo. Ten million users and ten years later: Python tutor's design guidelines for building scalable and sustainable research software in academia. In *The 34th Annual ACM Symposium on User Interface Software and Technology*, UIST '21, page 1235–1251, New York, NY, USA, 2021. Association for Computing Machinery.

[129] Philip J. Guo, Sean Kandel, Joseph M. Hellerstein, and Jeffrey Heer. Proactive wrangling: Mixed-initiative end-user programming of data transformation scripts. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, pages 65–74.

[130] Philip J. Guo, Sean Kandel, Joseph M. Hellerstein, and Jeffrey Heer. Proactive wrangling: Mixed-initiative end-user programming of data transformation scripts. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, UIST '11, page 65–74, New York, NY, USA, 2011. Association for Computing Machinery.

[131] Philip J. Guo and Margo Seltzer. Burrito: Wrapping your lab notebook in computational infrastructure. In *Proceedings of the 4th USENIX Workshop on the Theory and Practice of Provenance*, TaPP'12, Berkeley, CA, USA, 2012. USENIX Association.

[132] Aaron Halfaker, R. Stuart Geiger, Jonathan T. Morgan, and John Riedl. The rise and decline of an open collaboration system: How wikipedia's reaction to popularity is causing its decline. *American Behavioral Scientist*, 57(5):664–688, 2013.

[133] J. Hardin, R. Hoerl, Nicholas J. Horton, D. Nolan, B. Baumer, O. Hall-Holt, P. Murrell, R. Peng, P. Roback, D. Temple Lang, and M. D. Ward. Data Science in Statistics Curricula: Preparing Students to "Think with Data". *The American Statistician*, 69(4):343–353, October 2015.

[134] Mark Harrower and Cynthia A Brewer. Colorbrewer.org: an online tool for selecting colour schemes for maps. *The Cartographic Journal*, 40(1):27–37, 2003.

[135] Björn Hartmann, Mark Dhillon, and Matthew K Chan. Hypersource: bridging the gap between source and code-related web sites. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2207–2210, 2011.

[136] Björn Hartmann, Loren Yu, Abel Allison, Yeonsoo Yang, and Scott R. Klemmer. Design as exploration: Creating interface alternatives through parallel authoring and runtime tuning. In *Proceedings of the 21st Annual ACM Symposium on User Interface Software and Technology*, UIST '08, page 91–100, New York, NY, USA, 2008. Association for Computing Machinery.

[137] Jessen Havill. Embracing the Liberal Arts in an Interdisciplinary Data Analytics Program. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, SIGCSE '19, pages 9–14, New York, NY, USA, February 2019. Association for Computing Machinery.

[138] Brian Hayes. *Thoughts on Mathematica*. Pixel, Jan/Feb 1990.

[139] Andrew Head, Fred Hohman, Titus Barik, Steven M. Drucker, and Robert DeLine. Managing messes in computational notebooks. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, pages 1–12.

[140] Andrew Head, Fred Hohman, Titus Barik, Steven M. Drucker, and Robert DeLine. Managing messes in computational notebooks. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, CHI '19, New York, NY, USA, 2019. Association for Computing Machinery.

[141] Andrew Head, Jason Jiang, James Smith, Marti A. Hearst, and Björn Hartmann. Composing flexibly-organized step-by-step tutorials from linked source code, snippets, and outputs. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI '20, New York, NY, USA, 2020. ACM.

[142] Tony Hey, Stewart Tansley, and Kristin Tolle. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. October 2009.

[143] Raphael Hoffmann, James Fogarty, and Daniel S Weld. Assieme: finding and leveraging implicit references in a web search interface for programmers. In *Proceedings of the 20th annual ACM symposium on User interface software and technology*, pages 13–22, 2007.

[144] Tom Horak, Andreas Mathisen, Clemens N. Klokmose, Raimund Dachselt, and Niklas Elmqvist. Vistribute: Distributing interactive visualizations in dynamic multi-device setups. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, CHI '19, New York, NY, USA, 2019. Association for Computing Machinery.

[145] Meta Platforms Inc. LibCST - A Concrete Syntax Tree (CST) parser and serializer library for Python. https://github.com/Instagram/LibCST, 2019. Accessed: 2023-02-20.

[146] David A. Joyner and Charles Isbell. Master's at scale: Five years in a scalable online graduate degree. In *Proceedings of the Sixth (2019) ACM Conference on Learning @ Scale*, L@S '19, New York, NY, USA, 2019. Association for Computing Machinery.

[147] Alexander C. Kafka. With student interest soaring, berkeley creates new data-sciences division. *The Chronicle of Higher Education*, Nov 2018.

[148] Helge Kahler. *Supporting collaborative tailoring*. PhD thesis, Roskilde Universitetscenter, Department of Communication, Journalism and . . . , 2001.

[149] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. Wrangler: Interactive visual specification of data transformation scripts. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, page 3363–3372, New York, NY, USA, 2011. Association for Computing Machinery.

[150] Mary Kery and Brad Myers. Interactions for untangling messy history in a computational notebook. VL/HCC '18, pages 147–155, 10 2018.

[151] Mary Beth Kery, Amber Horvath, and Brad A Myers. Variolite: Supporting exploratory programming by data scientists. In *CHI*, volume 10, pages 3025453–3025626, 2017.

[152] Mary Beth Kery, Bonnie E. John, Patrick O'Flaherty, Amber Horvath, and Brad A. Myers. Towards effective foraging by data scientists to find past analysis choices. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, CHI '19, New York, NY, USA, 2019. Association for Computing Machinery.

[153] Mary Beth Kery and Brad A. Myers. Exploring exploratory programming. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 25–29, Oct 2017.

[154] Mary Beth Kery and Brad A Myers. Exploring exploratory programming. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 25–29. IEEE, 2017.

[155] Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E. John, and Brad A. Myers. The story in the notebook: Exploratory data science using a literate programming tool. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, pages 1–11.

[156] Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E. John, and Brad A. Myers. The story in the notebook: Exploratory data science using a literate programming tool. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, pages 174:1–174:11, New York, NY, USA, 2018. ACM.

[157] Mary Beth Kery, Donghao Ren, Fred Hohman, Dominik Moritz, Kanit Wongsuphasawat, and Kayur Patel. mage: Fluid moves between code and graphical work in computational notebooks. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, pages 140–151, 2020.

[158] Meraj Khan, Larry Xu, Arnab Nandi, and Joseph M. Hellerstein. Data tweening: Incremental visualization of data transforms. *Proceedings of the VLDB Endowment*, 10(6):661–672, 2017.

[159] Kimia Kiani, George Cui, Andrea Bunt, Joanna McGrenere, and Parmit K. Chilana. Beyond" One-Size-Fits-All" Understanding the Diversity in How Software Newcomers Discover and Make Use of Help Resources. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, pages 1–14.

[160] Aniket Kittur and Robert E. Kraut. Beyond wikipedia: Coordination and conflict in online production groups. In *Proceedings of the 2010 ACM Conference on Computer Supported Cooperative Work*, CSCW '10, page 215–224, New York, NY, USA, 2010. Association for Computing Machinery.

[161] Clemens N. Klokmose, James R. Eagan, Siemen Baader, Wendy Mackay, and Michel Beaudouin-Lafon. Webstrates: Shareable dynamic media. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, UIST '15, pages 280–290, New York, NY, USA, 2015. ACM.

[162] Carlos Delgado Kloos, Ma Blanca Ibáñez, Carlos Alario-Hoyos, Pedro J Muñoz-Merino, Iria Estévez Ayres, Carmen Fernández Panadero, and Julio Villena. From software engineering to courseware engineering. In *2016 IEEE Global Engineering Education Conference (EDUCON)*, pages 1122–1128. IEEE, 2016.

[163] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, and Carol Willing. Jupyter notebooks – a publishing format for reproducible computational workflows. In F. Loizides and B. Schmidt, editors, *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, pages 87 – 90. IOS Press, 2016.

[164] Donald E. Knuth. Literate programming. *Comput. J.*, 27(2):97–111, May 1984.

[165] Amy J Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, et al. The state of the art in end-user software engineering. *ACM Computing Surveys (CSUR)*, 43(3):1–44, 2011.

[166] Amy J. Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, Mary Beth Rosson, Gregg Rothermel, Mary Shaw, and Susan Wiedenbeck. The state of the art in end-user software engineering. *ACM Comput. Surv.*, 43(3):21:1–21:44, April 2011.

[167] Amy J Ko and Brad A Myers. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 151–158, 2004.

[168] Amy J Ko, Brad A Myers, and Htet Htet Aung. Six learning barriers in end-user programming systems. In *2004 IEEE Symposium on Visual Languages-Human Centric Computing*, pages 199–206. IEEE, 2004.

[169] Joseph A. Konstan, J. D. Walker, D. Christopher Brooks, Keith Brown, and Michael D. Ekstrand. Teaching recommender systems at large scale: Evaluation and lessons learned from a hybrid mooc. *ACM Trans. Comput.-Hum. Interact.*, 22(2), apr 2015.

[170] David Koop and Jay Patel. Dataflow notebooks: Encoding and tracking dependencies of cells. In *Proceedings of the 9th USENIX Conference on Theory and Practice of Provenance*, TaPP'17, page 17, USA, 2017. USENIX Association.

[171] Sean Kross and Philip J. Guo. Practitioners teaching data science in industry and academia: Expectations, workflows, and challenges. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, CHI '19, pages 263:1–263:14, New York, NY, USA, 2019. ACM.

[172] Sean Kross and Philip J. Guo. Tidy Data Tutor - visualize R tidyverse data pipelines. https://tidydatatutor.com/, 2021. Accessed: 2023-02-20.

[173] Philippe Kruchten, Robert L. Nord, and Ipek Ozkaya. Technical debt: From metaphor to theory and practice. *IEEE Software*, 29(6):18–21, 2012.

[174] Chinmay Kulkarni, Julia Cambre, Yasmine Kotturi, Michael S. Bernstein, and Scott R. Klemmer. Talkabout: Making distance matter with small groups in massive classes. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing*, CSCW '15, page 1116–1128, New York, NY, USA, 2015. Association for Computing Machinery.

[175] Chinmay E. Kulkarni, Michael S. Bernstein, and Scott R. Klemmer. Peerstudio: Rapid peer feedback emphasizes revision and improves performance. In *Proceedings of the Second (2015) ACM Conference on Learning @ Scale*, L@S '15, page 75–84, New York, NY, USA, 2015. Association for Computing Machinery.

[176] Sam Lau, Ian Drosos, Julia M. Markel, and Philip J. Guo. The design space of computational notebooks: An analysis of 60 systems in academia and industry. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, VL/HCC '20, Aug 2020.

[177] Sam Lau and Philip J. Guo. Pandas Tutor - visualize Python pandas code. https://pandastutor.com/, 2021. Accessed: 2023-02-20.

[178] Sam Lau, Deborah Nolan, and Joseph Gonzalez. *Learning Data Science*. O'Reilly Media, Inc., 2023.

[179] Samuel Lau and Joshua Hug. *Nbinteract: Generate Interactive Web Pages from Jupyter Notebooks*. Master's Thesis, Master's thesis, EECS Department, University of California, Berkeley, 2018.

[180] David Lazer, Ryan Kennedy, Gary King, and Alessandro Vespignani. The parable of Google Flu: Traps in big data analysis. *Science*, 343(6176):1203–1205, 2014.

[181] Sorin Lerner. Projection boxes: On-the-fly reconfigurable visualization for live programming. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–7, 2020.

[182] Aristotelis Leventidis, Jiahui Zhang, Cody Dunne, Wolfgang Gatterbauer, H. V. Jagadish, and Mirek Riedewald. QueryVis: Logic-based diagrams help users understand complicated SQL queries faster. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 2303–2318, 2020.

[183] Tom Lieber, Joel R. Brandt, and Rob C. Miller. Addressing misconceptions about code with always-on programming visualizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, page 2481–2490, New York, NY, USA, 2014. Association for Computing Machinery.

[184] Dong C. Liu and Jorge Nocedal. On the limited memory BFGS method for large scale optimization. *Mathematical programming*, 45(1):503–528, 1989.

[185] Zhicheng Liu and Jeffrey Heer. The effects of interactive latency on exploratory visual analysis. *IEEE transactions on visualization and computer graphics*, 20(12):2122–2131, 2014.

[186] Adam Loy, Shonda Kuiper, and Laura Chihara. Supporting data science in the statistics curriculum. *Journal of Statistics Education*, 27(1):2–11, 2019.

[187] David Malan. Cs50 docs: All the docs! https://cs50.readthedocs.io/, 2021. Accessed: 2021-01-15.

[188] David J. Malan. Cs50 sandbox: Secure execution of untrusted code. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, page 141–146, New York, NY, USA, 2013. Association for Computing Machinery.

[189] Wes McKinney. Pandas: A foundational Python library for data analysis and statistics. *Python for high performance and scientific computing*, 14(9), 2011.

[190] Mauricio Verano Merino, Jurgen Vinju, and Tijs van der Storm. Bacatá: Notebooks for DSLs, Almost for Free. In *Proceedings of the Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming*, Programming '20, New York, NY, USA, 2020. Association for Computing Machinery.

[191] Daphne Miedema and George Fletcher. Sqlvis: Visual query representations for supporting sql learners. In *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 1–9. IEEE, 2021.

[192] Kate Milner and Jonathan Rougier. How to weigh a donkey in the Kenyan countryside. *Significance*, 11(4):40–43, 2014.

[193] Diba Mirza, Phillip T. Conrad, Christian Lloyd, Ziad Matni, and Arthur Gatin. Undergraduate teaching assistants in computer science: A systematic literature review. In *Proceedings of the 2019 ACM Conference on International Computing Education Research*, ICER '19, page 31–40, New York, NY, USA, 2019. Association for Computing Machinery.

[194] Andrés Moreno, Niko Myller, Erkki Sutinen, and Mordechai Ben-Ari. Visualizing programs with jeliot 3. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, AVI '04, page 373–376, New York, NY, USA, 2004. Association for Computing Machinery.

[195] Michael Muller, Ingrid Lange, Dakuo Wang, David Piorkowski, Jason Tsay, Q. Vera Liao, Casey Dugan, and Thomas Erickson. How data science workers work with data: Discovery, capture, curation, design, creation. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, CHI '19, New York, NY, USA, 2019. Association for Computing Machinery.

[196] Brad A Myers, Amy J Ko, and Margaret M Burnett. Invited research overview: end-user programming. In *CHI'06 extended abstracts on Human factors in computing systems*, pages 75–80, 2006.

[197] Alok Mysore and Philip J. Guo. Torta: Generating mixed-media gui and command-line app tutorials using operating-system-wide activity tracing. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, UIST '17, pages 703–714, New York, NY, USA, 2017. ACM.

[198] MySQL. EXPLAIN Statement. https://dev.mysql.com/doc/refman/8.0/en/explain.html, 2023. Accessed: 2023-02-20.

[199] Bonnie A Nardi. *A small matter of programming: perspectives on end user computing*. MIT press, 1993.

[200] Fabio Niephaus, Eva Krebs, Christian Flach, Jens Lincke, and Robert Hirschfeld. PolyJuS: A Squeak/Smalltalk-Based Polyglot Notebook System for the GraalVM. In *Proceedings of the Conference Companion of the 3rd International Conference on Art, Science, and Engineering of Programming*, Programming '19, New York, NY, USA, 2019. Association for Computing Machinery.

[201] Deborah Nolan and Duncan Temple Lang. Computing in the statistics curricula. *The American Statistician*, 64(2):97–107, 2010.

[202] Midas Nouwens, Marcel Borowski, Bjarke Fog, and Clemens Nylandsted Klokmose. Between scripts and applications: Computational media for the frontier of nanoscience. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI '20, New York, NY, USA, 2020. ACM.

[203] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, and Vincent Dubourg. Scikit-learn: Machine learning in Python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

[204] Bo Peng, Gao Wang, Jun Ma, Man Chong Leong, Chris Wakefield, James Melott, Yulun Chiu, Di Du, and John N Weinstein. SoS Notebook: an interactive multi-language data analysis environment. *Bioinformatics*, 34(21):3768–3770, 05 2018.

[205] Jeffrey Perkel. Why jupyter is data scientists' computational notebook of choice. *Nature*, 563:145–146, 11 2018.

[206] Jeffrey M. Perkel. Why Jupyter is data scientists' computational notebook of choice. *Nature*, 563(7732):145–147, 2018.

[207] Alan J. Perlis. Special feature: Epigrams on programming. *SIGPLAN Not.*, 17(9):7–13, September 1982.

[208] Devin Petersohn, Stephen Macke, Doris Xin, William Ma, Doris Lee, Xiangxi Mo, Joseph E. Gonzalez, Joseph M. Hellerstein, Anthony D. Joseph, and Aditya Parameswaran. Towards scalable dataframe systems. *arXiv preprint arXiv:2001.00888*, 2020.

[209] Devin Petersohn, Stephen Macke, Doris Xin, William Ma, Doris Lee, Xiangxi Mo, Joseph E. Gonzalez, Joseph M. Hellerstein, Anthony D. Joseph, and Aditya Parameswaran. Towards scalable dataframe systems. *Proc. VLDB Endow.*, 13(12):2033–2046, jul 2020.

[210] Chris Piech, Lisa Yan, Lisa Einstein, Ana Saavedra, Baris Bozkurt, Eliska Sestakova, Ondrej Guth, and Nick McKeown. Co-teaching computer science across borders: Human-centric learning at scale. In *Proceedings of the Seventh ACM Conference on Learning @ Scale*, L@S '20, page 103–113, New York, NY, USA, 2020. Association for Computing Machinery.

[211] Fotis Psallidas and Eugene Wu. Smoke: Fine-grained lineage at interactive speed. *Proc. VLDB Endow.*, 11(6):719–732, feb 2018.

[212] Xiaoying Pu, Sean Kross, Jake M. Hofman, and Daniel G. Goldstein. Datamations: Animated explanations of data analysis pipelines. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, pages 1–14, 2021.

[213] Xuedi Qin, Yuyu Luo, Nan Tang, and Guoliang Li. Making data visualization more efficient and effective: A survey. *The VLDB Journal*, 29(1):93–117, 2020.

[214] Mark Raasveldt and Hannes Mühleisen. Duckdb: An embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1981–1984, New York, NY, USA, 2019. Association for Computing Machinery.

[215] Roman Rädle, Midas Nouwens, Kristian Antonsen, James R. Eagan, and Clemens N. Klokmose. Codestrates: Literate computing with webstrates. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, UIST '17, pages 715–725, New York, NY, USA, 2017. ACM.

[216] Bina Ramamurthy. A Practical and Sustainable Model for Learning and Teaching Data Science. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, SIGCSE '16, pages 169–174, New York, NY, USA, February 2016. Association for Computing Machinery.

[217] Suraj Rampure, Allen Shen, and Josh Hug. Experiences Teaching a Large Upper-Division Data Science Course Remotely. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, pages 523–528, New York, NY, USA, March 2021. Association for Computing Machinery.

[218] Stephanie Rosenthal and Tingting Chung. A data science major: Building skills and confidence. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, pages 178–184, 2020.

[219] Mary Beth Rosson and John M Carroll. The reuse of uses in smalltalk programming. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 3(3):219–253, 1996.

[220] Adam Rule, Amanda Birmingham, Cristal Zuniga, Ilkay Altintas, Shih-Cheng Huang, Rob Knight, Niema Moshiri, Mai H. Nguyen, Sara Brin Rosenthal, Fernando Pérez, and Peter W. Rose. Ten simple rules for writing and sharing computational analyses in jupyter notebooks. *PLOS Computational Biology*, 15(7):1–8, 07 2019.

[221] Adam Rule, Ian Drosos, Aurélien Tabard, and James D. Hollan. Aiding collaborative reuse of computational notebooks with annotated cell folding. *Proc. ACM Hum.-Comput. Interact.*, 2(CSCW), November 2018.

[222] Adam Rule, Aurélien Tabard, and James D. Hollan. Exploration and explanation in computational notebooks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, pages 32:1–32:12, New York, NY, USA, 2018. ACM.

[223] Mariam Salloum, Daniel Jeske, Wenxiu Ma, Vagelis Papalexakis, Christian Shelton, Vassilis Tsotras, and Shuheng Zhou. Developing an interdisciplinary data science program. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, pages 509–515, 2021.

[224] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. Vega-lite: A grammar of interactive graphics. *IEEE transactions on visualization and computer graphics*, 23(1):341–350, 2016.

[225] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. Vega-Lite: A grammar of interactive graphics. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):341–350, January 2017.

[226] Harvard Business School. What is the case study method? https://www.exed.hbs.edu/hbs-experience/learning-experience/case-study-method. Accessed: 2021-08-13.

[227] Matthew Seal, Kyle Kelley, and Michelle Ufford. Scheduling Notebooks at Netflix. Accessed: 2020-02-01.

[228] E. Segel and J. Heer. Narrative visualization: Telling stories with data. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):1139–1148, Nov 2010.

[229] Chad Sharp, Jelle van Assema, Brian Yu, Kareem Zidane, and David J. Malan. An open-source, api-based framework for assessing the correctness of code in cs50. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '20, page 487–492, New York, NY, USA, 2020. Association for Computing Machinery.

[230] Houshmand Shirani-Mehr, David Rothschild, Sharad Goel, and Andrew Gelman. Disentangling bias and variance in election polls. *Journal of the American Statistical Association*, 113(522):607–614, 2018.

[231] Nischal Shrestha, Titus Barik, and Chris Parnin. Unravel: A fluent code explorer for data wrangling. In *The 34th Annual ACM Symposium on User Interface Software and Technology*, UIST '21, page 198–207, New York, NY, USA, 2021. Association for Computing Machinery.

[232] Arjun Singh, Sergey Karayev, Kevin Gutowski, and Pieter Abbeel. Gradescope: A fast, flexible, and fair system for scalable assessment of handwritten work. In *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale*, L@S '17, page 81–88, New York, NY, USA, 2017. Association for Computing Machinery.

[233] Juha Sorva, Ville Karavirta, and Lauri Malmi. A review of generic program visualization systems for introductory programming education. *ACM Trans. Comput. Educ.*, 13(4), nov 2013.

[234] Juha Sorva and Teemu Sirkiä. Uuhistle: A software tool for visual program simulation. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli Calling '10, page 49–54, New York, NY, USA, 2010. Association for Computing Machinery.

[235] Sumukh Sridhara, Brian Hou, Jeffrey Lu, and John DeNero. Fuzz testing projects in massive courses. In *Proceedings of the Third (2016) ACM Conference on Learning @ Scale*, L@S '16, page 361–367, New York, NY, USA, 2016. Association for Computing Machinery.

[236] Sruti Srinivasa Ragavan, Sandeep Kaur Kuttal, Charles Hill, Anita Sarma, David Piorkowski, and Margaret Burnett. Foraging among an overabundance of similar variants. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, pages 3509–3521, 2016.

[237] Beckie Supiano. It matters a lot who teaches introductory courses. here's why. *The Chronicle of Higher Education*, Apr 2018.

[238] Steven L Tanimoto. Viva: A visual language for image processing. *Journal of Visual Languages & Computing*, 1(2):127–139, 1990.

[239] Steven L. Tanimoto. A perspective on the evolution of live programming. In *2013 1st International Workshop on Live Programming (LIVE)*, pages 31–34, May 2013.

[240] Randall H Trigg and Susanne Bødker. From implementation to design: tailoring and the emergence of systematization in cscw. In *Proceedings of the 1994 ACM conference on Computer supported cooperative work*, pages 45–54, 1994.

[241] Edward R. Tufte, Nora Hillman Goeler, and Richard Benson. *Envisioning Information*, volume 2. Graphics press Cheshire, CT, 1990.

[242] Michelle Ufford, M Pacer, Matthew Seal, and Kyle Kelley. Beyond Interactive: Notebook Innovation at Netflix. https://netflixtechblog.com/notebook-innovation-591ee3221233. Accessed: 2020-02-01.

[243] April Y. Wang, Ryan Mitts, Philip J. Guo, and Parmit K. Chilana. Mismatch of Expectations: How Modern Learning Resources Fail Conversational Programmers. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, pages 1–13. Association for Computing Machinery.

[244] April Yi Wang, Anant Mittal, Christopher Brooks, and Steve Oney. How data scientists use computational notebooks for real-time collaboration. *Proc. ACM Hum.-Comput. Interact.*, 3(CSCW), November 2019.

[245] April Yi Wang, Zihan Wu, Christopher Brooks, and Steve Oney. Callisto: Capturing the "why" by connecting conversations with computational narratives. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI '20, New York, NY, USA, 2020. ACM.

[246] Wei Wang, David Rothschild, Sharad Goel, and Andrew Gelman. Forecasting elections with non-representative polls. *International Journal of Forecasting*, 31(3):980–991, 2015.

[247] Mark Weiser. The computer for the 21st century. *Scientific American*, 265(3):66–75, January 1991.

[248] Hadley Wickham. Tidy data. *Journal of Statistical Software*, 59(10):1–23, 2014.

[249] Wikipedia. Package manager. https://en.wikipedia.org/wiki/Package_manager, 2021. Accessed: 2021-01-15.

[250] Wikipedia. Pivot table. https://en.wikipedia.org/wiki/Pivot_table, 2021. Accessed: 2021-01-15.

[251] Leland Wilkinson. The grammar of graphics. In *Handbook of Computational Statistics*, pages 375–414. Springer, 2012.

[252] Jeffrey Wong and Jason I Hong. Making mashups with marmite: towards end-user programming for the web. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 1435–1444, 2007.

[253] Eugene Wu. databass is a query compilation engine built for columbia's database courses. https://github.com/w6113/databass-public, 2020. Accessed: 2023-02-20.

[254] Eugene Wu. SQLTutor Visualizes Query Execution. https://cudbg.github.io/sqltutor/, 2022. Accessed: 2023-02-20.

[255] Annie TT Ying and Martin P Robillard. Code fragment summarization. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 655–658, 2013.

[256] YoungSeok Yoon, Brad A Myers, and Sebon Koo. Visualization of fine-grained code change history. In *2013 IEEE Symposium on Visual Languages and Human Centric Computing*, pages 119–126. IEEE, 2013.

[257] Alexey Zagalsky, Joseph Feliciano, Margaret-Anne Storey, Yiyun Zhao, and Weiliang Wang. The emergence of github as a collaborative platform for education. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing*, CSCW '15, page 1906–1917, New York, NY, USA, 2015. Association for Computing Machinery.

[258] Amy X. Zhang, Michael Muller, and Dakuo Wang. How do data science workers collaborate? roles, workflows, and tools. volume 1, New York, NY, USA, January 2020. Association for Computing Machinery.

[259] Xiong Zhang and Philip J. Guo. Ds.js: Turn any webpage into an example-centric live programming environment for learning data science. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, UIST '17, pages 691–702, New York, NY, USA, 2017. ACM.