The Story in the Notebook: Exploratory Data Science using a Literate Programming Tool

Mary Beth Kery[1] **Marissa Radensky**[2] **Mahima Arya**[1] **Bonnie E. John**[3] **Brad A. Myers**[1]

[1]Human-Computer Interaction Institute
Carnegie Mellon University
Pittsburgh, PA
mkery, mahimaa, bam @cs.cmu.edu

[2]Amherst College
Amherst, MA
mradensky19@amherst.edu

[3]Bloomberg L. P.
New York City, NY
bjohn11@bloomberg.net

ABSTRACT
Literate programming tools are used by millions of programmers today, and are intended to facilitate presenting data analyses in the form of a narrative. We interviewed 21 data scientists to study coding behaviors in a literate programming environment and how data scientists kept track of variants they explored. For participants who tried to keep a detailed history of their experimentation, both informal and formal versioning attempts led to problems, such as reduced notebook readability. During iteration, participants actively curated their notebooks into narratives, although primarily through cell structure rather than markdown explanations. Next, we surveyed 45 data scientists and asked them to envision how they might use their past history in a future version control system. Based on these results, we give design guidance for future literate programming tools, such as providing history search based on how programmers recall their explorations, through contextual details including images and parameters.

Author Keywords
Literate Programming; Exploratory Programming; Data Science; End-User Programmers (EUP), End-User Software Engineering (EUSE)

ACM Classification Keywords
H.1.2 User/Machine Systems: Software psychology; D.2.6 Programming Environments.

INTRODUCTION
As more and more people want to make use of the abundance of available data, there is a fast growing population of end-user programmers who are using the power of programming to work with data. Analyzing data through code has long been the domain of end-user programmers, including computational scientists, mathematicians, researchers, and engineers, many of whom never receive formal training in software engineering [30].

ACM ISBN 978-1-4503-5620-6/18/04…$15.00
https://doi.org/10.1145/3173574.3173748

As even more technical novices engage with code and data manipulations, it is key to have end-user programming tools that address barriers to doing effective data science. For instance, programming with data often requires heavy exploration with different ways to manipulate the data [14,23]. Currently even experts struggle to keep track of the experimentation they do, leading to lost work, confusion over how a result was achieved, and difficulties effectively ideating [11]. Literate programming has recently arisen as a promising direction to address some of these problems [17]. It originates from a 1984 paper by Donald Knuth:

> "*Time is ripe for significantly better documentation of programs, and that we can achieve this best by considering programs to be works of literature*" [17:1]

Knuth's philosophy held that humans should write code foremost as a natural language prose expression of their reasoning, which a *literate programming tool* facilitates by allowing formatted text annotations to be rendered inline with plain code. The actual computer code would be a secondary translation of these essay-like annotations [17]. Knuth's intended audience, mainstream software engineering, largely did not adopt the idea. Arguably, literate programming is a poor fit to software engineering, as documentation quickly gets out of date and is costly to maintain [18,21]. Heavy annotations also conflict with the idea in software engineering that well-structured code should "speak for itself" and be understandable with minimal documentation [19].

Yet in a different community, programming for math and sciences, literate programming has thrived since 1988 in tools such as Mathematica [13]. For a biologist, physicist or financial analyst who codes an analysis, there may be theories and equations embodied in their source code which could benefit from additional explanation as formatted equations or images [31]. By allowing the mix of any media needed to understand domain-rich code, literate programming has gained an important role in the sharing and reproducibility of computational analyses [31][16]. Today, open source literate programming tools like Jupyter notebooks [24] and knitr [31] have become hugely popular, with millions of users across a wide range of expertise and subject domains.

Although literate programming currently aids a programmer to communicate an analysis [26] data science tasks are known to be highly iterative and exploratory [9]. For every useful model feature or insightful visualization data scientists create, there may be many less-successful features, plots, or analyses they have tried. In order for the original programmer or another person to improve and build upon their work, knowledge of this exploration is important. Data scientists need to keep track of what they have attempted, including failed approaches, to justify *why* they choose certain approaches over others and to be more effective in their ideation [11]. Current approaches, including traditional version control, have been found to be ineffective or require high amounts of tedious manual note taking on part of the data scientist [14]. With the increased ability to keep code, data, input, and output together in one document, we wondered if literate programming helps data scientists retain the story of their in-progress exploration. Some in the scientific computing literature say yes. An article in Nature says that a literate programming tool like Jupyter notebooks "*helps researchers to keep a detailed lab notebook for their computational work*" [26]. However, although literate programming is an essential and distinct kind of programming in data computing today, there have been no studies on how literate programming affects data scientists. In this paper we present two studies that investigate how data scientists explore ideas as they develop code, and how and why they develop a narrative structure in a literate programming tool.

In our first study, we interviewed 21 professional data scientists who use Jupyter notebooks, a popular literate programming tool in data science with over 2 million users as of 2015 [25]. We found that although notebooks are limited in how they can be used to keep a detailed record of all explorations, several patterns of curatorial behavior emerged during iteration to build narrative.

In a second study, we probed how future tools may improve data scientists' interactions with exploration history. We surveyed 45 data scientists and asked how they might use historical records of their analyses if they had a magical oracle to deliver any prior analyses content to them. These results revealed highly varied ways of interacting with history. Finally, we discuss implications for future research.

BACKGROUND & RELATED WORK

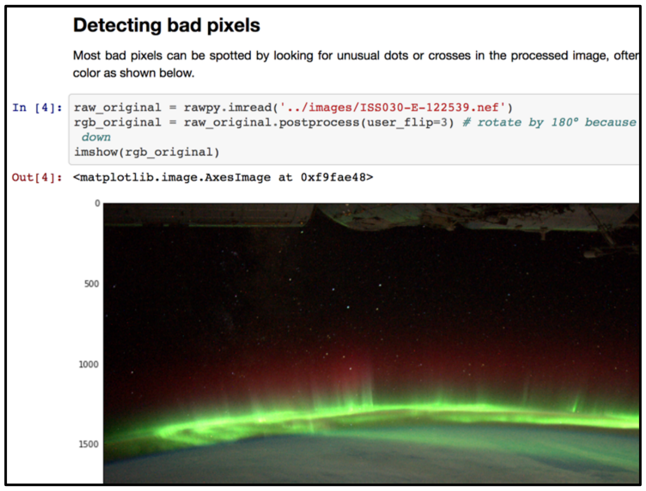
Data Science & Exploratory Programming

How do data scientists get to the final computational analyses? Guo [6] described the process for obtaining insights from data as having four phases: *preparation*, where data is acquired and cleaned, an iterative cycle of *analysis*, where code is written, inspected, and debugged, and *reflection* to interpret the outputs and suggest alternatives, and *dissemination* of the results in the form of written reports, raw results such as plots, or shared scripts [6]. We consider the iterations in the first three of these phases to be "exploration" [12], in which the programmer's goals for their program evolve by writing and testing code instantiations of different ideas. Some recent work has looked at exploratory programming for data science work [10,14,22], however not in the context of literate programming. Prior exploratory programming work has found that programmers conceptualize a narrative as they try to understand s*omeone else's* code [27], but that programmers create rather haphazard adhoc code during their own exploration [3,14].

Notebook programming environments

A "notebook" environment is one particular genre of literate programming tools that is supported by many data-centric literate programming tools such as Jupyter Notebooks, Mathematica [13], Databricks [34], Apache Zeppelin [7], and Sage Notebooks [5]. Although here we specifically study Jupyter Notebooks, we report on the usage of core features like cell structure, cell layout, and cell types, which are common features to this entire genre of tools and should allow our findings to generalize more broadly.

A notebook environment supports chunks of content, called "cells." A cell can contain code, output, a table, a plot, formatted "Markdown" text, or other kinds of media. For example, the first cell in Figure 1 contains formatted Markdown text (indicated by no background color and no number in the left margin), the second cell contains python code (labeled `In [4]:`), and the third contains the graphical output of that code (labeled `Out [4]:`), which is updated each time the code in that cell is run.



Detecting bad pixels

Most bad pixels can be spotted by looking for unusual dots or crosses in the processed image, ofte
color as shown below.

```
In [4]:  raw_original = rawpy.imread('../images/ISS030-E-122539.nef')
         rgb_original = raw_original.postprocess(user_flip=3) # rotate by 180° because
          down
         imshow(rgb_original)

Out[4]:  <matplotlib.image.AxesImage at 0xf9fae48>
```

Figure 1. Excerpt from a Jupyter notebook [8]

With a notebook, a programmer can produce a literate program that fits Knuth's ideal definition: a chronological progression of Markdown cells, code cells, and output cells, explaining everything from top to bottom. However, the notebook environment does not enforce this structure. A programmer is not forced to add Markdown explanations or any code comments. A notebook also allows each code cell to be edited and run individually at any time in any order, so rather than running the entire file from top to bottom, or only editing at the end of the notebook, programmers can pick and choose which code cells they would like to edit and run. As notebooks do not actually enforce a literate style, the programmers' situational needs are likely what motivates them to create a coherent literary-style document or not.

Literate programming tool developers ask their users for feedback. For example, in 2015, the Jupyter Project surveyed over 1000 users, on general usage, pain points, and feature requests [33]. They reported a quantitative analysis of all data, including keyword frequency counts on free-text responses. Version control was the most highly requested feature, although what the respondents meant by "version control" and what pain points it was intended to resolve was not probed. In addition to these data, there are many public examples of Jupyter notebooks online (for instance, a simple search on GitHub yields over 89,000 notebooks). However the artifacts themselves do not provide enough detail about *small-grained iterations and intention* [32] to answer our research questions, requiring us to use different methods to understand how data scientists explore ideas.

Lab notebooks and scientific documentation

Lab notebooks are a log of scientific activity that contain enough detail for a scientist to later reproduce their experiments [15]. Lab notebooks are ideally immutable logs once written, in order to serve as legal evidence of discovery in patent cases or questions of scientific validity [15]. Lab notebooks take the form of paper, digital note-taking tools, and hybrids between paper and digital notes [29]. Prior studies have explored the design needs that scientists have for their lab notebooks and their struggles with searching and maintaining scientific records [15,20,29]. Notebook programming, like any digital tool that can record text, can be used as a lab notebook [26] but can also serve many other purposes. Prior studies have largely focused on wet-lab scientists, whereas a chief difference in our current behavioral study is our sample of computational analysts who primarily work through code. Thus our investigation is not just on the record keeping, but on the iteration of their primary work through code.

STUDY 1

Method

To get as unbiased a view as possible of data scientists working with a literate programming environment, we followed the Grounded Theory Method (GTM) described by Corbin and Strauss [4] for data collection and analysis. We had the opportunity to collect data at the inaugural conference for Jupyter Notebook users (JupyterCon 2017). This provided a concentrated sample of people who have experience using Jupyter notebooks for real-world professional data analysis programming. Participants were recruited through conference speakers announcing the activity and organizers tweeting about it.

We interviewed 25 participants, but 4 were removed from our analysis because these interviewees turned out to be managers or otherwise did not personally do data analysis in notebooks. The final 21 interviewees held job titles shown in Table 1 and reported gender identity of 19 men, and 2 women. Interviews lasted 10 to 30 minutes, based on the participant's availability.

Role	Participant
College teacher & researcher	IP01, IP02, IP10, IP15, IP17
Financial analyst	IP05, IP06
Computational Biologist	IP08
Software Developer	IP19
Data visualization designer	IP03, IP04, IP21
Data Scientist	IP07, IP09, IP11, IP12, IP13, IP14, IP16, IP18, IP20

Table 1. Interview participants' primary job roles

The interviews were planned around a few open-ended questions, beginning with an overview: "Please tell me briefly what you use Jupyter notebooks for?". The next probe asked for details of what the interviewees did in notebooks to develop their ideas from inception to final result (over 60% of the interviewer's utterances were on this topic). If participants had their laptop with them, they were invited to show the interviewer their actual notebook documents, and four of the participants did. Our other planned questions involved sharing with other people, the use of markdown cells and code comments, the size of code cells and notebooks, and version control. Specific questions were generated on the fly in response to interviewees' answers e.g. "You mentioned that you've used version control, and you've also used a sort of numbering scheme. Is there any reason why you do one or the other?" Due to time constraints and which topics a participant expressed the most details on, not all interviews touched on all topics.

Analysis

After transcribing the 21 interviews, the first author did a line-by-line open coding on a random sample of six transcripts and produced a coding guide. The open coding evolved through this process. For example, codes for *prototyping*, *experimenting*, *testing*, and *iterating* were abstracted into *Testing Ideas*. Following the advice in [4] about avoiding bias through allowing scrutiny of the analysis by others, we used inter-rater reliability as an indication that these codes were meaningfully defined. The fourth author used the guide to code one of the transcripts, disagreements were discussed and the definitions improved. Both authors coded another transcript with the new guide and attained a Cohen's Kappa of 0.82. The fourth author then coded all transcripts using the new guide. Codes continued to evolve, primarily in sub-topics, e.g., *Annotating* became *Annotating with Markdown* and *Annotating with Code Comments*.

Limitations of the Data Collection and Analysis Method

Interviewing attendees at JupyterCon2017 is an example of "convenience sampling", as target participants were congregating at a single place, for a short period of time, and we could obtain permission and physical space from the conference organizers. Interviewing was a condensed and intense activity, with no time in between interviews in which to interweave analysis as is normal in GTM. Therefore, this study should be considered the first step in a research program and we provide recommendations for theoretical sampling at the end of the paper. Although four participants showed us examples of their work, subsequent data collection should consider doing field-based contextual inquiry [1] to better ground the data in observable behavior.

Results and Discussion

Interview participants will be referred to as IP01 to IP21 (see Table 1). First we discuss participants' use cases in the notebook environment, and then their iteration behaviors.

Use Cases

Our data revealed three use cases for notebooks: (1) preliminary "scratch pad" work, (2) work that ends up extracted out of the notebook for use in a production pipeline, and (3) work intended to be shared in different ways. These use cases are not disjoint. Pieces of code from a scratch pad can be extracted out into a script for use in a production pipeline. Code extracted for production is sometimes also shared with others via a detailed literate notebook complete with graphics and Markdown explanations.

Scratch Pad Use Case: Almost half our participants (10) used notebooks as *scratch pads*; 6 explicitly used that term. By this they meant they wrote code cells that they *expected to be preliminary and short-lived.* Scratch pads were used to answer a specific question, such as how to debug a piece of code, test out example code from the internet, or test if an analysis idea was worth pursuing further.

> *"I was just testing to make sure I had the syntax right on these tuples."* - IP13

> *"OK so can we do k-means on this dataset and like does it make sense"* - IP11

Scratchpad use appeared on two levels. "Scratch cells" were used within a notebook during exploration.

> *"only the last [cell] you did is actually useful so you get rid of some of this sort of trial and error-y things"* - IP18

At a higher level, some participants had whole notebooks that they used only for scratch work.

> *"In fact, I have a whole scratch directory where I just run a Jupyter server there and make a notebook, do something real quick and that's easier than just about anything else.*" - IP17

Since scratchpad work was intended to be short-lived, participants did not spend time annotating it. However, when the results of scratchwork were successful, that code was extracted out to a permanent place or transitioned from a scratchpad to a substantial literate document.

Production Pipeline Use Case: Seven participants reported that finished code was incorporated into a bigger code base. Finished code had to be extracted out of the notebook and placed in a plain-text code file when it was needed in a larger production pipeline because the extra metadata in a notebook file made it unusable by automated processes.

Sharing Use Case: Almost all our participants (20 of 21) shared the results of the analysis in a notebook, or the notebook itself, with someone else. These notebooks were typically significant explorations, included developing a model, conducting computational research, or creating a comprehensive analysis. Participants iterated on these analyses over the span of days to months, and generally took care in adding structure to these notebooks, as we will discuss later. Five participants were teachers who gave notebooks to their students for structured assignments:

> *"We do all of our teaching through notebooks. This includes lectures... in notebooks... then we give them projects to work on... in the form of notebooks."* - IP17

Two other participants shared notebooks with clients:

> *"not only for the actual data exploration myself but then to communicate the results of that effectively back to the person that asked me to do the work."* - IP08

Twelve shared with collaborators or teammates:

> *"I think the reason I use Jupyter is because it actually allows me to share the process by which I arrive at results with people who I want to convince of something, both so that they can spot any errors I may have made and also that they can use similar techniques in their own work."* - IP18

Two others referred to sharing with their future selves:

> *"And the idea is to comment them enough… [if you] came back next year would you understand exactly what the notebook was supposed to be doing."*- IP13

If any sharing was anticipated, including with a future self, participants reported putting extra care into making sure the notebook was clear to read.

Although notebooks themselves were shared, especially by teachers, many other formats for sharing were mentioned. For example, both financial analysts said they shared results in Excel. Other participants mentioned JPGs, HTML, PDFs, and slides. Today, this is often a necessity:

> *"most of my clients don't have Jupyter installed on their machines so I can't just give them a notebook file."*- IP08

On the other hand, five participants were enthusiastic about sharing interactive widgets which can be added as extensions to a cell to allow a reader to tune variables:

"Being able to have them… play with things will be a huge step forward… that would definitely have to be a centrally hosted kind of thing, because we're not going to expect anyone to download Jupyter"- IP08

Iteration Behaviors

Organizing the notebook while iterating: Participants reported different strategies for organizing their notebooks while iterating on the code. For example, the bottom of the notebook was used in idiosyncratic ways: two participants reported coding top-to-bottom so the most recent code was always at the bottom; one did all debugging at the bottom and often left it there; one put previously-written functions in a section at the bottom labeled with a "big markdown title" (IP11). Two participants put function definitions at the top whereas another used the top to import data. One participant took care to put all cells that loaded external packages in the same place, whereas another participant loaded external packages throughout the notebook.

Another repeated theme about notebook organization was adding new cells directly to where in the notebook the original analysis took place (mentioned by 4 participants):

"if I have to iterate a part of it then obviously I tried to do it close to the place where I inserted it previously, so either in the cell above or below" - IP09

This created implicit thematic regions of the notebook where an idea and alternatives to that idea were clustered.

Notebook constraints encourage "expand then reduce" behavior: 8 of the 21 participants explicitly mentioned that they tried to organize their notebooks so each code cell represented a logical unit in the analyses. However, this structure usually came about after cleanup. Participants reported a range of 1-70 lines of code in a single cell, but during active exploration, programmers instead favored creating many small code cells, often only 1-2 lines of code at a time, to incrementally test and build up functionality. This "expand then reduce" pattern was reported by six participants.

"So at the beginning it's usually a lot of little code cells that are one at a time... just making things work... I end up with this huge mess where there are several threads in sort of the same series. So I usually go back and start deleting things or combining cells" - IP17

After expanding on an idea, the reduce step is where participants talked about actively "cleaning up" code cells (6 participants) by deleting those they did not need anymore and consolidating working cells into one code cell that represented a logical unit.

Why was the expand step necessary? First, expanding an idea into many small code cells enabled a programmer to pick and choose which cells to run, and thus quickly test different approaches to the same problem. Second, having small code cells allowed a programmer to view cells of intermediary output after each code cell, making it easier to view and reason about their iteration. Third, some participants, although experts in their own domains, were not expert programmers. One participant (IP16) referred to the notebooks as a "crutch", because as a programming novice he felt the notebook had much more support for debugging one line at a time than a standard code editor.

Although generating small code cells was common, it became impractical to leave them all there for the long term. Participants complained that many loose code cells made the notebook a "mess" (a term used by five participants) and more difficult to understand:

"I'll clean up as I go because otherwise it would be very difficult to be remembering all that stuff" – IP5.

The expand-reduce behavior was often talked about in context of fairly low-level exploration, such as building up a working function, or figuring out appropriate library calls. Participants also talked about cleaning up after more significant explorations. For example 11 of 21 participants actively "reduced" their experimentation history by deleting alternatives of an idea from the notebook, or even deleting entire analyses that ultimately proved less fruitful. Although it would be less effort by the programmer to leave prior work in the notebook untouched and only add new work below, instead, programmers took active effort to continually delete scratch work from the notebooks. The attention to cleanup stands in contrast to prior work in non-notebook environments that has reported that programmers have low-investment in tidying code during exploratory data science programming [3,14].

Notebook constraints encourage managing the length of notebooks: Although a Jupyter notebook does not stop a programmer from adding unlimited content, for pragmatic reasons participants reported that the notebook interface does not work well with long documents. Four participants said that a long notebook was difficult to manage with scrolling up and down. Two others said that when code cells were distantly separated, the code was hard to comprehend. Because programmers kept different alternative analysis code in the notebook at the same time, they did not want to press the "run all" button to execute all code cells. Instead, participants ran analyses by picking and choosing individual code cells to run. This sometimes required going to the top of the notebook to rerun their standard code cells that import libraries and read in the data, and then scrolling back down to their current work. Scrolling to the top and down repeatedly over a long notebook became a burden.

The practical limit of a notebook, one participant (IP16) said, was about 60 code cells. After the notebook got too long or too cluttered, participants would either stop and curate the notebook by deleting alternatives no longer needed or start a new "fresh" notebook, copying in the most successful parts of the old notebook to the new one.

> *"when I open a notebook and I have to scroll for a long time… I just move on to a new notebook"* - IP03

It is unclear if this is a flaw or a benefit of the notebook design, because the *de facto* length limit encouraged data scientists to curate which ideas to retain moving forward.

Reuse, reduce, recycle (code): Almost all participants talked about reusing code (19). Of those, 11 simply used copy-paste. Four reported copying code cells within a notebook to keep code dependencies next to new code. Eight participants copied code into a different notebook:

> *"I'll be like, I remember I did this for this project but I can't remember exactly how to do it. So I'll go find the project and look at my code and copy paste into the other one."* - IP05

In addition to copy-paste to reuse functionality, five participants defined functions and six extracted code into an external script that could be imported into any notebook. For instance, IP11 created a new utils.py file for each notebook he worked with, in order to put reusable functions in a special place and reduce the size of the notebook. This practice has been encouraged in some science literature:

> "*As the code gets longer and more stable, it should be split out into Python modules to keep the notebook short and readable.*" [28]

However, this routine practice of extracting notebook code out into a plain Python (or Ruby, Scala, etc.) file for reuse is akin to "throwing the baby out with the bathwater" in that by discarding the notebook's metadata, the data scientists are also discarding their annotations, graphical output, and richness of exploration that shows how they derived that chunk of analysis code.

Narrative Structure of Notebooks

As in literature, the narrative structure of a notebook that tells the story of the analysis can be linear or non-linear. A pure linear structure would be akin to paper laboratory notebooks that keep a complete record of every thought, mistake, dead-end, and conclusion, in chronological order, often to preserve dates for patent purposes [29]. A non-linear structure could present the story of the analysis as a straightforward progression, recording only important decisions and rationale rather than the circuitous path that actually occurred. This would produce a curated document optimized for comprehensibility over completeness and chronology. A minority of our participants (4) attempted to keep a linear structure, e.g.,

> *"I have a sort of history of the development upstairs in the notebook."* - IP01

On the other hand, most of our participants produced a curated document, e.g.,

> *"I put the right code where it's supposed to be and delete the other cells, get rid of it to clean up my code."* - IP09

It is important to note that these two goals were contrasting situational goals, and not only individual preferences. Two participants who attempted to create complete records for the purpose of their research *also* created curated story notebooks. They created curated stories when the goal of that work was to present a specific analysis to an audience, and created detailed research records when research, not communication, was their primary goal.

We now turn our attention to how narrative structures appeared in the notebooks.

Explanation Annotations are Rarely Used in the Exploration Phase of Work: Only six participants spoke about annotating their code during the exploration phase of their work. Of these, three used markdown cells primarily as headers to separate sections of the code. Using markdown only for structural organization, rather than explanation woven throughout the program, is inconsistent with the definition of literate programming. However, three participants did use markdown during exploration to record their thoughts as they went along.

> *"I just put the [markdown] on some key changing points of the thought"* - IP06

> *"I'll use markdown cells to put any notes I notice like. 'OK. Here's a common way that you make a mistake'"* - IP17

One person used code comments (not markdown cells) as

> *"a way for me to track my process of going along and to keep thinking through the problem… comments help me think."* - IP16

In contrast, after the process of exploratory programming was done, if a participant had a long-term purpose for their notebook such as sharing it or keeping it for a record, they would then add more explanatory documentation to the notebook that is more consistent with literate programming. Nine participants reported this behavior:

> *"If it's a notebook that... has to be rerun by me or by somebody else, I'll try to explain the data sources, where it comes from... And just the different major steps in the analysis,"* - IP05

> *"[When] I'm doing research, it's almost like a source code. And then when I really want to clean it up and show it to someone else, then I put in annotation."* - IP10

Mechanisms used to provide narrative structure: The interweaving of input code and output is a primary mechanism of narrative structure of any notebook.

> *"it's nice because all of the images are right there and all the code is right there."* - IP02

In addition, participants talked about how they used the code itself to provide narrative structure, through which code cells they chose to keep and delete.

"if you read my notebook from top to bottom you see the evolution of my thought. You see that I first do some... small part of the function, then… the universal functions… And finally... the conclusions that are made using the functions." - IP06

During the dissemination phase, participants used markdown to create a narrative structure:

"If I'm putting together a script notebook for someone else to use [I'm] making it nicer and adding markdown and everything." - IP03

Sometimes markdown was used to tell a linear story:

"Not only do I just say [in markdown] what I… removed, but sometimes I show those intermediate steps so that they can see the progression from raw uncleaned data to the final product." - IP08

Other times, markdown was used to curate the story in concert with deleting less important code to make the key points of the exploration more apparent:

"I end up with a really messy notebook and I might end up… opening another one and just doing the clean version… The stuff that worked. And just with more comments and just you know nicely formatted." - IP05

One participant felt that the narrative structure emerged as he cleaned up his code:

"I usually go back and start deleting things or combining cells or shifting things around… so the eventual form with the notebook only gradually emerges" - IP17

In contrast, participants who used a linear narrative structure made earlier cells in their notebook historical and immutable by avoiding overwriting code cells that had already produced output, and added new code cells only to the bottom of the notebook. This meant a series of code cells that perform a data transformation might be duplicated at different locations in the notebook, enabling the author to keep different output for each variation and retain a chronological record. This completeness came at the cost of a hard to read narrative:

"*I can't get an overview of what's going on in my notebook… it's just a lot of stuff and stuff... with all these random outputs that never get cleaned up.*" - IP01

Although these participants achieved a more detailed record by avoiding curation, it should be noted that they necessarily curated each time they decided whether to overwrite and re-run a cell or create a new cell.

Version Control

The vast majority of participants spoke about iterating extensively on their code (only 2 of 21 said their code development progressed in a straightforward fashion). However, all this exploration was generally thrown away. Participants identified why current means of versioning with literate programming notebooks is fairly dysfunctional. Recall that improved version control was the most requested feature in the Jupyter Project 2015 UX Survey [33].

While 11 of our 21 interview participants did use a version control tool like Git for their notebooks, the metadata included in the file format of notebooks currently makes Git utilities such as diff (viewing the differences between two source code versions) unusable because utilities were not designed to treat metadata differently from source code.

"diffing is so hard...I develop until I'm happy and then I'm going to put it in a file and then I'm going to version control the file not the notebook." - IP15

Although a technical annoyance, two participants' workplaces had scripts to extract the code out of a notebook and just version that, enabling a normal Git workflow.

Some participants appreciated the conditions under which formal version control like Git or SVN is important:

"If it's an application, usually there's all sorts of dependencies. And that's when version control becomes important. Also if… I have to release this in concert with something else… then you have to do some sort of version control" - IP12

Because of the effort involved in using formal versioning tools, participants often used informal versioning. Consistent with our prior observations of informal versioning practices [14], 4 of 21 participants relied on different file names for version control:

"The stupidest possible version control… you rename the notebook to something like V0 or V3." - IP18

This informal method has its own problems:

"...we have like 500 different files all variations of the same thing and they're all numbered in a way that's completely useless because I don't remember whether it was two weeks ago or two months ago I was at this stage of the iteration." - IP12

Also consistent with [14], participants used local versioning inside their notebook. For instance, two participants said they had code cells containing alternative approaches simultaneously in view to be able to compare them. However placing alternative code and output cells directly above or below the original was a problem due to screen space. With large code or output cells, authors could not see everything they needed at once in a single notebook window. Two participants reported workarounds in order to see alternatives side by side, for example opening two different windows of the same notebook to place the windows side by side on their screen.

STUDY 2

These data and prior work [14] suggest that data scientists need better versioning tools, but no previous studies have probed more specific pain points and functional needs for future tools to address. We conducted a second study to explore how data scientists think they might want to use a detailed record of their explorations.

Method

We drew inspiration from the "grounded brainstorming" procedure described in [1] to design a short computer survey to elicit data scientists' versioning needs. The survey first grounded them in their real experience by asking them to describe a recent exploratory data analysis they had performed (Q1). The survey then primed them to think of an imagined future with a "*magical perfect record of every analysis run you did in this project. You also have a magic search engine that can retrieve for you any code version, parameters used or output from the past.*" After this preparation, we asked people to brainstorm by typing *"as many queries as you can think of that could be helpful to you to retrieve a past experiment. Don't worry about feasibility."* We asked participants to "*phrase* [your query] *in natural human language like you're talking to a colleague*" both to discourage the participant from assessing feasibility and to provide phrases we were likely to understand (Q2). Finally, we probed for the types of real world problems such future magic technology might be able to solve: "*Has **not** being able to find a past experiment ever caused you problems? If yes, what happened?*" (Q3).

The survey was conducted at JupyterCon 2017 on a laptop (27 participants) and online (18 participants), advertised through posts on the first author's social media inviting data scientists to participate.

Analysis

Treating the participants' answers to Q2 as brainstorming ideas, we used affinity diagramming to cluster the imagined queries into different categories (Table 2). We performed a separate affinity diagramming to cluster participants reported problems (Q3) into categories.

Results

All survey participants will be referred to as SP01 to SP45. In Q2, 45 participants generated a total of 125 queries for the "magic search engine". Participants' queries referred to many kinds of contextual details, including libraries used, output, plots, data sources, parameters used, running time of an analysis, time periods, version numbers, and specific dates (Table 2). Participants did not limit themselves to imagining only prose queries, e.g., SP13 submitted *"Here's a visualization I produced, let me right click on it to give me the script to produce it"*.

In addition, some queries required semantic or conceptual understanding of the programmer's task, for instance "*Show me all the different ways I oversampled the minority class*" (SP21), or "*What was the state of my notebook the last time that my plot had a gaussian-ish peak?*" (SP17). Some participants also asked for properties of an analysis relating to process or rationale, for example: "*Find me how I cleaned the data from start to finish*" (SP08) or "*What questions did I ask that didn't pan out?*" (SP12).

Referenced analysis attribute	# Queries
Analysis (e.g. "convolutional model")	46
Output (e.g. "training accuracy")	25
Time period (e.g. "go back 5 hours")	17
Dataset (e.g. "previous test result for this particular dataset")	15
Plot (e.g. "how did I generate plot 5")	11
Specific variable	10
Parameters	10
Library	4
Running time of the program (e.g. "How long did it take to process country X")	3

Table 2. Affinity diagramming groups for 125 queries. A query can appear in multiple groups.

For Q3, 31 of the 41 participants who answered experienced problems from being unable to find prior analyses versus 10 who had not. The most-mentioned problem was the need to rewrite code (20 participants). This need had several sources, including losing the code because that part of the work had not been saved or losing the rationale behind the code because it had not been recorded. Without the code that produced a result, 7 participants no longer trusted that result. The second most frequently reported problem was time delays (12 participants) caused by excessive time searching for code, having to re-run code, or having to rewrite code. Two participants reported having to consult with a colleague to solve the problem.

The answers to Q3 validate prior findings [15,29] that past analyses can be hard and sometimes impossible for data scientists to find. In notebooks, version control is currently poor enough that records of prior iterations often do not exist. Yet even with improved version control, it should be noted that some 'magic' queries from Q2 cannot easily be translated into traditional text search-engine queries, e.g. "*the last time that my plot had a gaussian-ish peak*".

IMPLICATIONS FOR DESIGN

Solutions for some of the problems uncovered in this research may already exist in newer UI extensions to the notebook. For instance, the existing Table of Contents plugin for notebooks may help with participants' reported struggles navigating long notebooks. The recently released tool called JupyterLab makes collapsing cell sections easier and allows side-by-side viewing of notebooks, which may obviate participants' workarounds for comparing two notebooks.

Our results contain inspiration for other features like the option to mark a cell so its output is not displayed or to not be run at all. Below we discuss design implications for broader thematic changes to notebook tools.

Automated version control

For interview participants who attempted to keep a full record of their exploration, this often meant that their notebooks lacked clarity and were full of "stuff" and "random outputs" (IP01). An automated form of version control may be a more systematic way of keeping a clear history of an analysis, while freeing the data scientist to keep a more concise and clear notebook without needing to keep old cells on view at all times. Additionally, the diversity of features and details that participants wanted to retrieve about their analyses in Q2 of Study 2 suggests that automated forms of version control, paired with much richer forms of search, will be needed to match a data scientist's conceptual recollection of their work with the artifacts they are looking for. For instance, in order to answer questions about plots or visual output in a notebook, the user must be able to search based on a visual artifact. In order to answer Q2 questions about parameters and specific variables, a version and search system must keep some knowledge about the notebook's abstract-syntax-tree to track the different values of a variable from one notebook version to the next. In addition, to answer Q2 questions particular to a dataset or library such as "which of my analyses used dataset X?", some code dependency information will need to be stored. A tool could collect variable environment information in an active notebook and track which lines of code use which resources. Current version control systems like Git keep plain-text "blobs" of code and do not store the required structured data about the code. Inferring and then storing more program-rich metadata would allow a variety of context-specific version searches as expressed in Q2, although at the cost of requiring more metadata storage per notebook and more time-expensive forms of program analysis. However, such automated versioning would require no effort for the programmer using the notebook.

Adding multiple lenses to cell representation

Jupyter Notebooks' prominent cell structure may visually encourage logical chunking of code and results in a way that an unbroken stream of text in a source code file does not. Indeed, participants described having many small, loose code cells as "messy". Although programmers' inattention to code structure during exploration is a prevailing theme in related work [3,14], our participants routinely curated their exploration, suggesting cell messiness was disruptive enough to cue some cleanup. This suggests that cell structure is a valuable UI feature that might be leveraged to address some of the problems our participants reported.

Logically chunking cells was a mechanism of delivering narrative in notebooks, yet participants also used cell structure to support versioning, comparison, and debugging. We propose a *lens* [2] interaction approach to notebooks that would enable addressing the same cell content from a number of different perspectives. A lens is a UI approach which provides transparent overlays and transformed ways of viewing existing content while adding as little as possible to the existing screen space to avoiding overcomplicating the existing interface [2]. For instance, before reducing cells to logical units, participants used small 1 or 2 line code cells for active iteration as these were far easier to debug. Instead of this manual expand-reduce behavior, a debugging lens [12] could explode a cell into individual lines for debugging and then collapsed back into a cell story unit when the programmer is satisfied with the code.

For participants who wanted a detailed record, small and repeated code cells provided versioning detail at the cost of a messy notebook where the main points were difficult to pick out. Instead of having to keep all those small iteration cells around, a notebook could support a historical lens for cells [2,12]. Similar to how a debugging lens might "explode" the cell into one-liners, a history lens could explode the cell into a historical view of how the current version of the cell was achieved. Instead of copy-pasting cells to compare and create slight versions of them, this historical view could provide a local versioning mechanism within cells, similar to Variolite [14].

When it comes to a historical record, a recurring theme in literature on scientific lab notebooks is that users need to interact with bits of content from many different perspectives [20,29]. For instance, in Oleksik et al.'s study of physicists' lab notebooks, participants wanted the "ability to pivot on specific entities or attributes" to generate summaries on-the-fly that were particularly relevant to their needs at different points in time [20]. Study 2's Q2 results support this need for pivot views. Thus, in a history lens on a notebook cell, a user should be able to select the item they are curious about, such as a plot in an output cell. Rather than displaying all versions of the cell, a history lens could then display only those versions relevant to the particular pivot selection. Given the difficulties reported by participants in Q3 of study 2 with trusting prior results, the origins of a result and related dependencies must be displayed with enough detail to support data scientists deciding whether to trust a result. Tools could support this decision by showing the date, data, author, and code from which the results came, and alerting users if that exact data or code was later edited, which may make these results outdated.

IMPLICATIONS FOR FUTURE STUDIES

Our data analysis has produced accounts of three types of use cases, a variety of mechanisms for narrative structure and version control, and several design directions. As mentioned, these results can be viewed as hypotheses in a longer research program and future studies should consider using theoretical sampling in the following three ways.

First, our convenience sampling did not screen for profession or domain of study, and our data suggest that different professions do different things. For example, the financial analysts used Excel for dissemination; the computational biologist said clients do not have Jupyter; and teachers shared notebooks, but tended not to create production code. Future research should sample from different professions, especially if designers want to produce tools for specific user groups.

Second, we studied only Jupyter Notebooks and the iteration behaviors we observed may have been influenced by specific UI details. For example, managing the size of notebooks may not be necessary with an environment with an easy cell hiding capability, as is included in the JuptyerLab tool. Future studies may want to sample other tools to find which behaviors generalize.

Finally, perhaps most importantly, the data about participants' behaviors was self-reported, not observed, so future studies should seek to verify these hypotheses through direct observation e.g., [1] or fine-grained logging that could confirm behaviors like expand/reduce.

CONCLUSION

Data scientists from a broad range of domains and skill levels are doing impactful work through code. In this study of literate programming we found that programmers do create narrative structure during their exploration, although often by manipulating cell structure rather than using much explanatory markdown. Creating a narrative also intersects and conflicts with other objectives, such as participants who prototyped and debugged code by expanding-reducing cell structure, or participants who kept a clutter of old iterations in their notebooks to retain a history of their work. We hope our results will inspire future designs for ways to interact with notebook cells for browsing history, debugging, and other tasks which may improve the effectiveness of literate programming for supporting data science.

ACKNOWLEDGEMENTS

We thank our participants and the JupyterCon organizers for giving us permission and space to run our study. This research was supported in part by a grant from Bloomberg L. P. and in part by NSF grant IIS-1314356. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the sponsors.

REFERENCES

1. Hugh Beyer and Karen Holtzblatt. 1997. *Contextual design: defining customer-centered systems*. Elsevier.
2. Eric A. Bier, Maureen C. Stone, Ken Pier, William Buxton, and Tony D. DeRose. 1993. Toolglass and magic lenses: the see-through interface. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, 73–80. DOI: https://doi.org/10.1145/166117.166126
3. Joel Brandt, Philip J. Guo, Joel Lewenstein, and Scott R. Klemmer. 2008. Opportunistic programming: How rapid ideation and prototyping occur in practice. In *Proceedings of the 4th international workshop on End-user software engineering*, 1–5. DOI: https://doi.org/10.1145/1370847.1370848
4. Juliet Corbin and Anselm Strauss. 1990. Grounded Theory Research: Procedures, Canons and Evaluative Criteria. *Zeitschrift für Soziologie* 19, 6: 515. DOI: https://doi.org/10.1007/BF00988593
5. The Sage Developers. *SageMath, the Sage Mathematics Software System (Version x.y.z)*.
6. Danyel Fisher, Badrish Chandramouli, Robert DeLine, Jonathan Goldstein, Andrei Aron, Mike Barnett, John C. Platt, James F. Terwilliger, and John Wernsing. 2014. Tempe: an interactive data science environment for exploration of temporal and streaming data. *Tech. Rep. MSR-TR-2014--148*.
7. Apache Software Foundation. 2017. Apache Zeppelin 0.7.0. Retrieved from https://zeppelin.apache.org/
8. Maik Riechert. 2016. Repairing Bad Pixels. Retrieved January 6, 2018 from https://github.com/letmaik/rawpy-notebooks/blob/master/bad-pixel-repair/bad-pixel-repair.ipynb
9. Philip Jia Guo. 2012. Software tools to facilitate research programming. Ph.D. Dissertation. Stanford University.
10. Philip J. Guo and Margo I. Seltzer. 2012. Burrito: Wrapping your lab notebook in computational infrastructure. In *Proceedings of the 4th USENIX Workshop on the Theory and Practice of Provenance*. DOI:
11. Charles Hill, Rachel Bellamy, Thomas Erickson, and Margaret Burnett. 2016. Trials and tribulations of developers of intelligent systems: A field study. In *Visual Languages and Human-Centric Computing (VL/HCC), 2016 IEEE Symposium on*, 162–170. DOI: https://doi.org/10.1109/VLHCC.2016.7739680
12. Scott E. Hudson, Roy Rodenstein, and Ian Smith. 1997. Debugging Lenses: A New Class of Transparent Tools for User Interface Debugging. In *Proceedings of the 10th Annual ACM Symposium on User Interface Software and Technology* (UIST '97), 179–187. DOI: https://doi.org/10.1145/263407.263542
13. Wolfram Research Inc. Mathematica, Version 11.2.
14. Mary Beth Kery, Amber Horvath, and Brad A. Myers. 2017. Variolite: Supporting Exploratory Programming by Data Scientists. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems* (CHI '17), 1265–1276. DOI: https://doi.org/10.1145/3025453.3025626
15. Clemens Nylandsted Klokmose and Pär-Ola Zander. 2010. Rethinking Laboratory Notebooks. In *Proceedings of COOP 2010*. Springer, London, 119–139. DOI: https://doi.org/10.1007/978-1-84996-211-7_8
16. Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E. Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B. Hamrick,

Jason Grout, Sylvain Corlay, and Others. 2016. Jupyter Notebooks-a publishing format for reproducible computational workflows. In *ELPUB*, 87–90.

17. Donald Ervin Knuth. 1984. Literate programming. *Computer Journal* 27, 2: 97–111. DOI: http://dx.doi.org/10.1093/comjnl/27.2.97
18. Timothy C. Lethbridge, Janice Singer, and Andrew Forward. 2003. How software engineers use documentation: The state of the practice. *IEEE Software* 20, 6: 35–39. DOI: https://doi.org/10.1109/MS.2003.1241364
19. Robert C. Martin. 2009. *Clean code: a handbook of agile software craftsmanship*. Pearson Education.
20. Gerard Oleksik, Natasa Milic-Frayling, and Rachel Jones. 2014. Study of Electronic Lab Notebook Design and Practices That Emerged in a Collaborative Scientific Environment. In *Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work & Social Computing* (CSCW '14), 120–133. DOI: https://doi.org/10.1145/2531602.2531709
21. David Lorge Parnas. 1994. Software aging. In *Proceedings of the 16th international conference on Software engineering*, 279–287.
22. Kayur Patel. 2010. Lowering the barrier to applying machine learning. In *Adjunct proceedings of the 23nd annual ACM symposium on User interface software and technology*, 355–358. DOI: https://doi.org/10.1145/1866218.1866222
23. Kayur Patel, James Fogarty, James A. Landay, and Beverly Harrison. 2008. Investigating statistical machine learning as a tool for software development. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 667–676. DOI: https://doi.org/10.1145/1357054.1357160
24. Fernando Pérez and Brian E. Granger. 2007. IPython: a System for Interactive Scientific Computing. *Computing in Science and Engineering* 9, 3: 21–29. DOI: https://doi.org/10.1109/MCSE.2007.53
25. Fernando Perez and Brian E. Granger. 2015. Project Jupyter: Computational Narratives as the Engine of Collaborative Data Science. *Project Jupyter Blog*. Retrieved from http://blog.jupyter.org/2015/07/07/project-jupyter-computational-narratives-as-the-engine-of-collaborative-data-science/
26. Helen Shen. 2014. Interactive notebooks: Sharing the code. *Nature* 515, 7525: 151. DOI: https://doi.org/10.1038/515151a
27. Sruti Srinivasa Ragavan, Sandeep Kaur Kuttal, Charles Hill, Anita Sarma, David Piorkowski, and Margaret Burnett. 2016. Foraging Among an Overabundance of Similar Variants. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems* (CHI '16), 3509–3521. DOI: https://doi.org/10.1145/2858036.2858469
28. Jean-Luc R. Stevens, Marco Elver, and James A. Bednar. 2013. An automated and reproducible workflow for running and analyzing neural simulations using Lancet and IPython Notebook. *Frontiers in neuroinformatics* 7. DOI: https://dx.doi.org/10.3389%2Ffninf.2013.00044
29. Aurélien Tabard, Wendy E. Mackay, and Evelyn Eastmond. 2008. From Individual to Collaborative: The Evolution of Prism, a Hybrid Laboratory Notebook. In *Proceedings of the 2008 ACM Conference on Computer Supported Cooperative Work* (CSCW '08), 569–578. DOI: https://doi.org/10.1145/1460563.1460653
30. Greg Wilson. 2006. Software carpentry: getting scientists to write better code by making them more productive. *Computing in science & engineering* 8, 6: 66–69. DOI: https://doi.org/10.1109/MCSE.2006.122
31. Yihui Xie. 2014. knitr: a comprehensive tool for reproducible research in R. *Implement Reprod Res* 1: 20.
32. Youngseok Yoon, Brad A. Myers, and Sebon Koo. 2013. Visualization of fine-grained code change history. In *Visual Languages and Human-Centric Computing (VL/HCC), 2013 IEEE Symposium on*, 119–126. DOI: https://doi.org/10.1109/VLHCC.2013.6645254
33. 12/2015. Jupyter Notebook 2015 UX Survey Results. *Jupyter Project Github Repository*. Retrieved from https://github.com/jupyter/surveys/blob/master/surveys/2015-12-notebook-ux/analysis/report_dashboard.ipynb
34. 2013. Databricks. Retrieved from https://databricks.com/