

Error Identification Strategies for Python Jupyter Notebooks

Derek Robinson

Department of Computer Science
University of Victoria
Victoria, Canada
drobinson@uvic.ca

Neil A. Ernst

Department of Computer Science
University of Victoria
Victoria, Canada
nernst@uvic.ca

Enrique Larios Vargas

Department of Computer Science
University of Victoria
Victoria, Canada
elariosvargas@uvic.ca

Margaret-Anne D. Storey

Department of Computer Science
University of Victoria
Victoria, Canada
mstorey@uvic.ca

ABSTRACT

Computational notebooks—such as Jupyter or Colab—combine text and data analysis code. They have become ubiquitous in the world of data science and exploratory data analysis. Since these notebooks present a different programming paradigm than conventional IDE-driven programming, it is plausible that debugging in computational notebooks might also be different. More specifically, since creating notebooks blends domain knowledge, statistical analysis, and programming, the ways in which notebook users find and fix errors in these different forms might be different. In this paper, we present an exploratory, observational study on how Python Jupyter notebook users find and understand potential errors in notebooks. Through a conceptual replication of study design investigating the error identification strategies of R notebook users, we presented users with Python Jupyter notebooks pre-populated with common notebook errors—errors rooted in either the statistical data analysis, the knowledge of domain concepts, or in the programming. We then analyzed the strategies our study participants used to find these errors and determined how successful each strategy was at identifying errors. Our findings indicate that while the notebook programming environment is different from the environments used for traditional programming, debugging strategies remain quite similar. It is our hope that the insights presented in this paper will help both notebook tool designers and educators make changes to improve how data scientists discover errors more easily in the notebooks they write.

1 INTRODUCTION

Jupyter Notebook¹ is an open-source, browser-based programming environment that allows users to weave rich text, code, equations, and visualizations into a single human-readable document. Jupyter Notebooks and other computational notebooks, such as Google Colab², RMarkdown Notebooks³, and Azure Notebooks⁴, have become immensely popular for anyone who wishes to perform data analysis or exploration tasks. The Jupyter platform specifically has grown exponentially since 2015, with over 6 million publicly available Jupyter Notebooks currently residing on GitHub alone [15].

Despite its popularity, Jupyter Notebook users have mixed opinions about the process of debugging [6]. Some praise its ability to help them find errors quickly, but others complain about a poor debugging experience [6]. This has given us some insight into how users *feel* about debugging Jupyter Notebooks, but not much about their actual debugging processes.

Debugging involves two phases. The first phase involves identifying what and where the error is, and the second phase involves determining how best to fix the error [29]. We have many insights about the process of debugging software systems and programs in general [2, 14, 26], but we lack similar insights for computational notebooks. To the best of our knowledge, there are a few studies which examine how computational notebooks are debugged. For instance, Yang *et al.* [28], focused on a tool for improving code understanding with program synthesis but did not synthesize strategies used in debugging. Without knowing *how* computational notebooks are debugged, it is difficult to support—whether with tools or processes—the debugging of computational notebooks.

Motivated by the sheer popularity [15] of the Jupyter Notebook platform across many disciplines, we aimed to identify strategies adopted by users of Jupyter Notebooks in the first phase of debugging, i.e., strategies for *identifying* errors in data analysis notebooks. Understanding debugging strategies may provide educators and students with valuable knowledge about how errors are found in computational notebooks, improving how they teach or learn error-finding methods. Although our findings are not aimed at any specific community of Jupyter Notebook users, as the platform is used across many different domains [5, 20, 25], we hope that our results may serve as recommendations for users of Jupyter Notebooks who may not have a strong background in the process of debugging.

Our work focused on this research question:

What strategies do data scientists use to find statistical, data, and programming errors in Python Jupyter Notebooks?

Our study is a conceptual replication of a work-in-progress study of how experts and novices debug RMarkdown documents [13]. All source materials [3] were provided by the authors of the R study. We replicated the R study design and performed an observational study with 14 participants tasked with finding and understanding errors in one of four Jupyter Notebooks. We translated the RMarkdown study materials [3] into Python, retaining the same

¹<https://jupyter.org/>

²<https://colab.research.google.com/>

³<https://rmarkdown.rstudio.com/>

⁴<https://notebooks.azure.com/>

pre-populated errors in the statistics, data, and programming code and the same supporting analysis text as designed in the original study materials [13]. The participants could use any error-finding method they chose and were not time-constrained. We observed that participants followed seven different strategies. We characterized the strategies according to the frequency of use and their success (which are not necessarily the same!). Consulting external resources via a search engine was the most *common* strategy. However, the most *successful* strategy was Expectation Confirmation, where there was a mismatch between what explanatory markdown cells claimed and what the code actually did. On average, our participants found approximately 40% of the errors in the notebook they analyzed.

2 BACKGROUND

Jupyter Notebooks are the “de-facto standard” for data scientists [16], and much has been learned about how reproducible they are, the quality of the code written in them, and the narratives that describe the analyses within them [17, 19, 22, 23]. These studies agree that notebook code is frequently low-quality and error-prone. Closely related to our work is that from Yang *et al.* [28]. They report on a tool to support bug detection in Kaggle notebooks, which they characterized as ‘data wrangling code’. Like us, they show this style of development is quite different than pure source code approaches and prone to errors, yet not well supported by tools. They introduce WrangleDoc, a program synthesis technique to summarize code in order to facilitate debugging. We did not examine such summarization approaches in our study, and our Jupyter instance contained no plugins.

While the work of Yang *et al.* highlights the potential problems with data science code, there is still much we do not understand. In particular, their study looked at specific tool support using a documentation approach. We focused instead on the strategies leading to the discovery of notebook errors, the starting point of the debugging process.

Challenges of using notebooks. While the literature on notebook error detection is limited, other research has revealed the challenges of using notebooks. Chattopadhyay *et al.* identified the main pain points of using computational notebooks, including Jupyter [6]. Of interest to this study is the *Manage Code* pain point which mentions that without sufficient software engineering support, debugging, writing code, managing dependencies, and testing relies on *ad-hoc* workarounds. Specifically, they found that writing code in notebooks efficiently requires knowledge of all the function names and classes, plus the use of a second window to search for online resources such as documentation [6]. They also observed a divide in how their participants felt about debugging in notebooks. Some participants were able to find errors in a notebook quickly, but others found that debugging was a horrible experience when they had to rely on `print()` statements. In addition, they found that testing in computational notebooks was difficult as there is no standard method to test a notebook. Some study participants wrote test cases in the same notebook, while others created a new notebook for testing.

Debugging traditional programs. There is plenty of related research on debugging conventional programs. We mention some

closely related studies here. Murphy *et al.* present a qualitative study of the debugging strategies employed by computer science students [14]. They observed three distinct categories of strategy: the good, the bad, and the quirky. The good strategies (or effective strategies) included *gaining domain knowledge, tracing, testing, understanding the code, using resources, using tools, isolating the problem, pattern matching, and considering alternatives*. They also identified that many students employed strategies that were less effective (the bad). These “bad” strategies were the same as the effective strategies, but employed less effectively. Finally, the quirky strategies were ones which surprised Murphy *et al.*

Hypothesis-driven debugging. Alaboudi and Latoza authored two papers that relate to our study. The first paper, titled “Using Hypothesis as a Debugging Aid” [1], describes two studies. In the first study, they observed live-stream videos of developers’ programming activities. Their second study was a controlled study of 25 participants tasked with debugging three API misuse problems. Overall, they observed that developers found it challenging to formulate a reasonable hypothesis about a potential error. In the second paper, “An Exploratory Study of Debugging Episodes”, Alaboudi and Latoza observed 15 live-streamed programming sessions (in C, C#, JavaScript) [2]. They found that developers spent 48% of their programming sessions debugging. They also found that no single activity dominated a debugging session, with developers spending varied amounts of time on different activities. Additionally, they observed significant differences between long and short debugging episodes. Short debugging episodes focused on editing and testing code, while long debugging episodes involved various activities, such as consulting external resources, and inspecting program state in addition to testing and editing.

Student approaches to debugging. Like our study, Whalley *et al.* examined students’ thoughts about their debugging process (in non-notebook code). They examined whether reflecting on the debugging process helps students perceive a need for change in their approach, and if they perceive value in a structured, formal debugging process [26]. Whalley *et al.* used semi-structured interviews to answer their research questions. Their analysis uncovered themes about code comprehension, bug location, information gathering strategies, challenges locating bugs, emotions felt during the debugging process, and the value students give to a formal debugging process. When students were asked to reflect on their debugging process, their comments referred to both high- and low-level activities. High-level activities included activities such as reading code, search space reduction, and hypothesis forming. Most of the reflections shared about debugging were about low-level activities, such as where to place print statements, code tracing, and examining function parameters and return values. Many students perceived debugging as inefficient, likely due to the lack of a formal process to follow. One-third of the participants described their debugging process as flawed, and they universally described their hypothesis-forming method as imprecise, opting to guess and check instead.

Data science debugging. Debugging traditional programs is well studied and the works discussed above are a very small subset of the work available on debugging. In contrast to the debugging of traditional programs, data science work can be quite different [27]. For example, it is a common part of the workflow for scientists

to re-run analyses (e.g., as part of exploratory data analysis). This might happen when, for example, removing outliers or experimenting with different hyper-parameters. Thus, research has looked at supporting such experimental workflows in order to manage versions of notebooks [10, 24], and support cleanup and refactoring [9]. Related to that is work that uses tools to debug data flows in large (non-notebook) data analytics pipelines [18], or support statistical transparency with multiverse analysis [7]. None of this work focused on *how* errors were detected. However, Brown *et al.* introduces four error types present in data analysis [4].

3 MATERIALS AND METHODS

To answer our research question, we observed the behaviour of 14 participants as they browsed and debugged existing Jupyter Notebooks that contained errors. The observations took place over Zoom, and participants shared their screens. We recorded video and audio of the meetings for a later qualitative analysis of the strategies our participants used. The data collection was conducted from November 2020 to January 2021.

3.1 Participants

Table 1 summarizes participant demographics. A majority (8) of the participants were from the domain of computer science or computer science combined with either music, biochemistry, or life sciences. The other (5) participants were each students in one of chemistry, physics, civil engineering, software engineering, and electrical/computer engineering. The remaining participant was a professional who worked in the education domain. Nine participants had used computational notebooks for less than one year, and the remaining five used them for between one and three years. All participants stated that Jupyter Notebooks was their computational notebook of choice, with two also using Google Colab.

We recruited participants by contacting instructors of three 400/500-level courses with data science themes, by posting on online communities, and through personal contacts. Participation in our study was voluntary, but we encouraged participation by providing a \$25 Amazon gift card to the first 12 respondents. Our study was approved by our institutional review board.

3.2 Study Materials

The notebooks used for our study were translated from R notebooks created as part of an in-progress study to investigate data scientists' debugging behavior [13]. The R notebooks were written by statistics and data science education researchers [3] and covered two different topics (NBA Player of the Week and the 2011 Spain Election). Each notebook had three versions: A, B, and C. Errors were introduced into versions A and B of the original R notebooks by two members of the R study [13] team, the C notebook had no errors. Table 2 lists the number of errors per notebook.

We translated the R notebooks (including errors) to Python, and the translations were verified by a third party, experienced in Python, statistics, and data science. Additionally, the first author of this paper verified that the Python translations returned the same data, visualizations, and values. The Python and R Notebooks are available at [3], along with a complete list of the errors.

```

1 nba = nba.assign(
2     Height = pd.to_numeric(nba['Height'].str
3     ↪ .replace('cm', '').str.replace('-[0-9]*', '')),
4     Weight = pd.to_numeric(nba['Weight'].str
5     ↪ .replace('kg', ''))

```

Listing 1: A data error: assumes all data is in cm/kg.

```

1 my_test = ttest_ind(
2     x1 = nba[(nba['Position'] == 'PG')]['Height'],
3     x2 = nba[(nba['Position'] == 'SG')]['Height'],
4     alternative = 'smaller')

```

Listing 2: A statistical error: using a 1-sided t-test when a 2-sided t-test is the proper choice.

We retained the error classification system from the R Study, which identified embedded errors as: **data**, **statistical**, and **programming** [3]. In the notebooks, error types were not mutually exclusive and a given error could be a **programming** error in addition to a **data** or **statistical** error. These three different types of errors align with the different categories of errors defined by Brown *et al.* [4]. We describe the three categories of errors below and provide examples using the NBA Player of the Week notebook(s).

Data errors occur when a notebook does not fully explore the dataset, or when the format of the data is misunderstood. Listing 1 shows an example of a **data** error: the Height column is of type string and is assumed to either contain the centimeter unit or a string value representing a measurement. Similarly, it is assumed that the measurements in the Weight column are all in kilograms, with some measurements containing the kilogram units. In fact, the Height column has units of either centimeters, such as 203cm, or feet-inches, such as 6-11. The Weight column has measurements which are in kilograms and contain the kilogram units, or are measurements in pounds that contain no units. The above code cell removes the centimeter unit by calling `str.replace('cm', '')`. The inches measurement is also removed by calling `str.replace('-[0-9]*', '')`. The same is done with the Weight column, removing the kilogram units via `str.replace('kg', '')`. Thus, the column is incorrectly cleaned as the above code cell performs no unit conversions. This leaves the Height and Weight columns in mismatched units without any unit identifier.

Statistical errors occur either when an incorrectly chosen statistical test or visualization is used, or when a correctly chosen test or visualization is wrongly interpreted by the user. In Listing 2, the goal is to determine if a statistical difference between the average height of point guards and shooting guards exists using a t-test. The error is that the alternative parameter is set to `smaller`, indicating a one-sided t-test. This parameter should be set to `two-sided` as the goal was to determine whether or not a statistical difference exists, rather than which average was smaller.

Lastly, **programming** errors occur when a code cell does not achieve the goal stated in the preceding markdown cell. The goal of Listing 3 is to filter the nba dataframe so that it only contains unique players. While this code cell does output a set of unique

Table 1: Participant Demographics

Participant	Role	Domain	Notebook Experience (yrs)
P1	Master's	Electrical & Computer Engineering	< 1
P2	Master's	Chemistry	1-3
P3	Undergraduate	Computer Science & Biochemistry	< 1
P4	Undergraduate	Computer Science & Life Sciences	1-3
P5	Master's	Computer Science	< 1
P6	Undergraduate	Physics	1-3
P7	Master's	Civil Engineering	< 1
P8	Undergraduate	Software Engineering	< 1
P9	Undergraduate	Computer Science	1-3
P10	Educational Specialist	Education	< 1
P11	Undergraduate	Computer Science & Music	< 1
P12	Undergraduate	Computer Science & Music	< 1
P13	Doctoral	Computer Science	1-3
P14	Undergraduate	Computer Science & Music	< 1

```

1 nba.groupby('Player').agg(
2     Height=('Height', 'median'),
3     Weight=('Weight', 'median'),
4     Position=('Position', 'first'))

```

Listing 3: A programming error: using the original, not the filtered data-frame.

players, a copy is returned, which is not saved to the nba dataframe. Through the remainder of this notebook, the original nba dataframe is used, and thus the code has an error and does not achieve its goal.

3.3 Jupyter Notebook Study Design

Each participant was tasked with finding potential errors in one of the four notebooks which contained errors (Versions A or B). Version C was shown to them after their analysis if they wanted to see an error-free version. We aimed to balance the number of participants analyzing each notebook (see Table 2). This task was open-ended in that the participants were allowed to use any method they liked to find potential errors. The only specific instructions given were for them to think aloud whenever possible and to notify the researcher when they thought they had found an error.

We performed two rounds of pilots (with members of our research group) to improve the study task and to confirm our study would provide sufficient observations on error finding strategies. The feedback from the pilots helped us improve the study materials. The supplementary materials contain the task description, interview questions, and Jupyter Notebooks [3].

At the start of each study session, we described the task and emphasized that our aim was not to test their skills. Participants were then presented with a Jupyter Notebook and informed that any of the notebook components might contain errors that they should try to identify. We mentioned they could modify the notebook, search the documentation, or use the internet for help.

Each study session consisted of two phases: an observational phase and an interview phase. During the observational phase, participants analyzed the notebook for errors while one researcher observed their behaviours and took notes. Once the participant was satisfied with their analysis of the notebook, we held the interview phase of the study.

The interview began with unstructured questions, using notes from our observations to guide our follow-up questions. Asking these questions immediately after the participant had performed their task was important as their strategies were still fresh in their mind. These unstructured questions were asked to gain insights into a specific approach and why it was used. Following the unstructured part of the interview, additional questions were asked about the participant's domain of study, how long they had been using Jupyter Notebooks, and the computational notebook they used most often. The complete list of these additional questions is available in a replication package [3].

3.4 Data Collection and Analysis

The Zoom video recordings of the studies were uploaded and we analyzed the recordings directly using ATLAS.ti 8⁵. We used an open coding process to code all activities performed by our participants. The first author of this paper performed the initial coding. After the initial coding cycle, discussion sessions were held with the second and fourth authors, where the codes were further analyzed and compared with the findings from previous participants, and refined in an iterative manner.

Throughout our discussion sessions, we identified emergent higher-level groups for the codes and merged some codes:

- **Action:** An action that a participant performed.
- **Docs:** A specific documentation website that a participant visited.
- **Online Resource:** An online resource other than documentation that a participant visited.
- **Reasoning:** A reason for performing an action or a reason for why something was an error.
- **Participant Attribute:** To describe a participant.

⁵<https://atlasti.com/product/what-is-atlas-ti/>.

Table 2: Number of Participants and Errors per Notebook (D: Data Error S: Statistical Error P: Programming Error)

Notebook	# of Participants	Participants	# of Errors	Distribution of Errors	Size (# of Code Cells)
nba_analysis_A	4	P1, P8, P10, P13	6	D:3 S:3 P:2	10
nba_analysis_B	3	P2, P9, P12	4	D:0 S:2 P:3	12
elections_analysis_A	4	P3, P5, P6, P14	10	D:2 S:7 P:5	12
election_analysis_B	3	P4, P7, P11	10	D:2 S:9 P:3	12

Throughout our discussion sessions, we noticed many *actions* were performed together. We called these connected sets of actions *strategies*. The first author analyzed the raw data again to identify and code strategies from each group of *actions*.

Once we identified strategies, we analyzed videos again to determine the success rate of each strategy. Whenever we observed a participant analyzing an erroneous cell, we entered their chosen strategy into a spreadsheet, along with the type of error they were working on and whether that strategy was successful or not.

4 FINDINGS

Our research question asked what strategies data scientists use to find statistics, data/domain, and programming errors. Our analysis reveals (a) *actions* and (b) *strategies* that our participants employ to find errors in Python Jupyter notebooks. Additionally, we present (c) the *relationship between strategies and error-finding success*. Tables 3 and 5 show the entire list of actions and strategies identified in our exploratory observational study. Finally, in Table 6, we present how strategies relate to the different error types. We describe each action, strategy, and their respective relationship with an error type below.

4.1 Actions Taken

Our participants performed various actions while analyzing the notebooks to find errors. In this context, an action is an (atomic) activity such as reading a markdown/code cell or examining a CSV file. Table 3 lists the number of participants who performed each action and the average amount of time all participants spent per action. There were some actions that participants always used, such as reading code and markdown cells, writing or editing code, and using the search engine. Other actions often used included looking at the documentation, checking code output, and inspecting dataframes. Finally, a few actions were only occasionally used, such as inspecting a CSV file or adding a comment. We describe these actions in more detail below.

A1: Reading a code cell. This action refers to when participants (P1-P14) *read through a code cell to understand what it was doing*.

A2: Reading markdown cells. This action refers to when a participant (P1-P14) *read through a markdown cell to gain context into what the preceding code cell tried to accomplish*.

When analyzing a notebook to find errors, participants performed actions A1 and A2 successively. In this scenario, P8 emphasized, “[I read] the documentation first then [I read] the code”. Additionally, P12 highlighted the value of reading the markdown aloud to better understand what was going on.

A3: Writing/Editing code. This action occurred when participants (P1-P14) *wrote new code in a code cell (either one they added or one present in the notebook) or edited a code cell that was initially in the notebook*. We observed that participants edited code for several different reasons. For example, P14 stated that they edited code cells to make them more readable. Other participants, such as P10, edited the parameters of functions to view more of the data returned by that function: for example, P10 edited calls to Pandas Series.nlargest() function. Some participants also wrote new code into the notebooks, which served various purposes. For instance, P13 wrote code during their analysis of the nba_analysis_A notebook to verify if two sets of rows in the nba dataframe were the same. Both P6 and P11 wrote code to perform type checking through the use of Python’s type() method.

A4: Using a search engine. All participants used a search engine to *access some online resource or documentation page*. Typically participants transitioned from the notebook to the search engine and then to either an online resource or a documentation page. Depending on whether or not the initial search result was helpful, they would return to the notebook or select another result from the search engine. The **Search Engine** action is highly associated with both **A5: Looking at documentation** and **A8: Looking at an online resource**. We define online resources as any website other than a documentation page. Table 4 shows the most commonly accessed documentation websites and online resources.

A6: Checking code output. Commonly, participants (P1-P12, P14) *inspected the output of a code cell visually, either one initially present in the notebook or one which the participant added*.

A7: Inspecting dataframe. We observed that participants (P1-P4, P6-P12, P14) used the DataFrame.head() method to *visually inspect the dataframe, either to gain a preliminary understanding of the data or to check if anything seemed out of place*.

A9: Inspecting a graph. Participants (P3, P5-P14) performed this action to *visually inspect any graph present in the notebook*.

A10: Inspecting CSV File. In a similar situation to A7, participants (P5, P6, P9, P10, P13) inspected the data in its raw state. For instance, P13 pointed out that when using Jupyter Notebooks, they do not use the CSV viewer native to Jupyter; instead, they use an alternative application. Likewise, P9 indicated they use Notepad++ to view their CSV files. Finally, P10 highlighted that they inspect the CSV file when they are unsure how to perform a task programmatically. In this matter, P10 stated, “I’m just learning Python, so I can’t...list these things, I actually refer to the CSV quite a bit”.

Table 3: Actions Taken in Error Identification.

Action	Action ID	# Participants	Average Time Spent (mm:ss)
Reading code cell	A1	14	06:46
Reading markdown	A2	14	05:51
Writing/Editing code	A3	14	03:21
Using a search engine	A4	14	01:55
Looking at documentation	A5	13	03:00
Checks code output	A6	13	02:24
Inspecting DataFrame	A7	12	04:07
Looking at an online resource	A8	12	03:06
Inspecting graph	A9	11	01:53
Inspecting CSV file	A10	5	02:40
Reading an error message	A11	3	01:51
Adding a comment	A12	2	08:42

Table 4: Number of Visits. † indicates a Documentation page. The remainder are Online Resources. Fourteen other Online Resources were each visited between one and three times.

Resource	Number of Visits
Pandas †	58
Plotnine †	28
Statsmodels †	21
stackoverflow.com	14
geeksforgeeks.org	6
investopedia.com	6
Numpy †	4
Scipy.stats †	4
tutorialspoint.com	4
w3schools.com	4

A11: Reading an error message. This action occurred when participants (P1, P3, P5, P8-P11) changed the notebook as initially the notebooks did not return any error messages. In this scenario, P10 pointed out that when they see an error message, they “*don’t have a clue*”.

A12: Adding a comment. This action occurred when participants (P5, P11, P14) *added a comment to a code cell either in the form of a note or to comment out code*.

While these actions capture the more atomic tasks our participants performed, we also observed that several actions were used together to form strategies that helped participants find or understand the cause of errors. In the remainder of this section, we describe these strategies in more detail.

4.2 Error-Finding Strategies

Participants performed many of the preceding actions together to serve a particular purpose. We call a collection of related actions a *strategy*. We describe the strategies we found in detail. Table 5 gives a brief description along with the number of participants who used each strategy.

Search Engine-Driven Approach. The most common strategy we observed was the *search engine-driven approach*, which every participant used. All participants made several transitions from the

notebook to the search engine, then to an external resource, until they found a helpful online resource or documentation page.

Participants outlined three different reasons for using the search engine and external resources. First, they used the search engine as a first step to gather a solution from an online reference. For instance, P12 highlighted that they use Google quite often when using a Jupyter Notebook, and without it, they would not know what to do. Not knowing what to do without the search engine hints at being dependent on it; it is unknown whether this is caused by a lack of general programming knowledge or knowledge of a specific API, such as Pandas.

Second, the search engine was also used as a confirmatory aid; this happened when participants had prior knowledge. However, they sought supplementary expertise to confirm or refresh their intuition. For instance, P7 stated they often remember general concepts but use the search engine to gather information about what some specific terms mean to interpret them correctly, such as when P7 gathered information about interpreting the results of an ordinary least squares (OLS) regression.

Finally, participants also used the search engine to gather code snippets as potential solutions. P8 emphasized that their particular use of the search engine was to find code snippets that could help them fix the errors they identified.

Assume and (Sometimes) Check. Participants would only cursorily inspect a code cell, see what the code is doing, and return to it only when they identified a potential problem in their theory of the notebook’s execution. They then made an assumption about where in the preceding cells that problem happened, and then examined that code in more detail than they did on their first pass over it. However, participants “sometimes” left some assumptions unchecked. This may be due to the contrived nature of the study (fixing the bug was not part of the task). When participants did check assumptions, they wrote new code in the notebook or examined the dataframe/CSV file.

Consider the error and thought processes of P8 while they use the *assume and (sometimes) check* strategy to determine the error described in Listing 1 (code cell 3 of the notebook `NBA_Analysis_A`). P8 began analyzing the notebook using a *once-over* (see the next strategy) and noticed in a later code cell that the given mean of the height column was roughly 12. They then remarked that a “*mean*

Table 5: Strategy Descriptions

Strategy	Description	# Participants	Associated Actions
Search Engine-Driven Approach	Using the search engine and external resources to gather useful information.	14	A4, A5, A8
Assume and (Sometimes) Check	Making an assumption related to the notebook or to an API call and sometimes checking it.	14	A3, A7, A12
Expectation Confirmation	The participant's expectation, set up by an explanatory markdown cell, of what a code cell does cannot be confirmed upon seeing its output.	7	A1, A2
Once-Over	Briefly browsing through the notebook in order to gain a preliminary understanding of what it contains.	4	A1, A2, A6
Re-implement to Check	Re-implementing a code cell using a different syntax in order to check its validity.	3	A3, A6
Key Information	Extracting need-to-know information from a markdown cell and placing it in a comment inside the related code cell.	1	A2, A10
Start With What You Know	Starting at a point in the notebook which is most familiar.	1	A1, A2

height of 12 doesn't seem to make a lot of sense" (since height in cm should be (broadly) greater than 100cm and less than 225cm). They then transitioned to read code cell 3 (Listing 1 line 2), which cleaned and adjusted the height column. Rereading the code cell led them to *assume* something must have gone wrong in that notebook cell. They then inspected the original dataframe and made another assumption: *"Here the measurements are presumably in feet-inches and over here we have them in cm"*. This second assumption is an example of assuming the purpose of a series of method calls. A closer inspection of code cell 3 allowed them to identify the error as replacing inches with the empty string and not accurately converting feet-inches to cm.

Expectation Confirmation. Seven participants (P1, P3, P5, P7, P10, P11, P13) indicated that a discrepancy between explanatory text in a markdown cell and the subsequent code cell helped them identify an error. P7 described the explanatory markdown as a *"guidance for what I should be looking for"*, and that when a difference occurred between the markdown and the code cell, they knew something was incorrect. Additionally, P5 used an analogy to describe the discrepancy between the markdown and code, stating, *"It's basically like 'Hey, we did this' and then [I] look at the code and it's like 'No, you didn't.'"* Finally, P11 emphasized, *"what I was expecting is that we want a percentage and this is obviously not a percentage"*, outlining how the markdown sets their expectations. When the code does not fulfill these expectations, they know something is wrong.

Once-Over. Four participants (P2, P6, P8, P14) used this strategy, which involves looking through the notebook to gain a preliminary understanding. This strategy consists of reading markdown and code cells, running code cells and briefly checking their output, and generally inspecting the notebook's initial state. A once-over gives a basic understanding of what the notebook is doing without too much detail. All four of the participants, when using the *once-over* strategy, employed different language to describe it. For example, P2 stated they were getting *"a lay of the land"*.

Re-implement to Check. The *re-implement to check* strategy was used by three participants (P1, P6, P11) and implies rewriting a code cell using a different syntax and then comparing the results of

```
1 nba[(nba['Position'] == 'PG') | (nba['Position'] ==
↪ 'SG')].groupby('Position').agg(Height=('Height',
↪ 'mean'))
```

Listing 4: Code snippet P1 wrongly thought was incorrect.

```
1 nba[(nba['Position'] == 'PG') | (nba['Position'] ==
↪ 'SG')].groupby('Position').agg('Height').mean()
```

Listing 5: P1's re-implementation of Listing 4.

both to see if there are any differences. For example, P1 wrongly believed that Listing 4 was incorrect due to the `.agg()` syntax. They continued to add a new cell and rewrite the code (Listing 5), only to find that they produced the same result.

P6 stated that they would have shown a correlation by plotting rather than using an OLS regression, but they did not re-implement this code cell as they were unfamiliar with the Plotnine package used to generate the plots. While not precisely re-implementation, P11 wrote pseudocode before looking at a code cell and after reading its markdown explanation. They then compared this pseudocode to the actual code, and if similar, P11 believed this code cell was correct and continued to a new cell. Additionally, participants combined this pseudocode strategy with a re-implementation to further validate a given code snippet.

Key Information. The *Key Information* strategy was used four times by P5 and describes extracting only the information you need from the markdown description of a code cell; P5 then placed this information inside the code cell as a comment. Extraction of the key information allowed P5 to get the information closer to the code, and reduced the number of times they re-read a markdown cell to remind themselves of what a code cell was doing. In addition, they highlighted how extracting the key information allowed for easier comparison of the code and markdown, and eliminated any extraneous information they did not need to know. Using the *Key Information* strategy allowed P5 to more easily employ the *Expectation Confirmation* strategy.

Table 6: Relating Strategies (from Table 5) and Error Type. A dash (-) indicates no use. The once-over strategy was not used for any of the error types. There are a maximum of 21 programming errors, 12 statistical errors, and 7 data errors.

Strategy	Error Type	Times Used	Errors Found	Percentage
Search Engine-Driven Approach	Programming	26	11	52.4%
	Statistical	16	7	53.9%
	Data	10	2	28.6%
Assume and Check	Programming	13	7	33.3%
	Statistical	8	4	30.8%
	Data	5	4	57.1%
Expectation Confirmation	Programming	20	17	81.0%
	Statistical	6	0	0%
	Data	7	7	100%
Re-implement to Check	Programming	1	0	0%
	Statistical	-	-	-
	Data	1	0	0%
Start With What You Know	Programming	1	0	0%
	Statistical	-	-	-%
	Data	1	0	0%
Key Information	Programming	3	2	9.52%
	Statistical	2	1	7.69%
	Data	-	-	-%

Start With What You Know. P5 employed another strategy named *Start With What You Know*, which involved analyzing parts of the notebook they were familiar with first. They mentioned that doing so made them “*feel more confident*”, and that starting with the topics they were more familiar with gave them a better chance to find errors. This confidence then allowed them to find errors in the other sections of the notebook as they were better able to understand the nature of the errors.

4.3 Strategy Success

We now describe how the strategies outlined in Section 4.2 were used to find the various types of errors present in each notebook. As our study was exploratory, we do not make any claim that these are the best strategies for finding a particular type of error (such a claim would require future work). Recall that our study included the analysis of three types of errors from [13]: programming errors, statistical errors, and data/context errors (see Section 3.2). The error types are not mutually exclusive and a given error can belong to more than one error type. While some strategies were less successful, they are still worth examining. First, we cannot claim that unsuccessful strategies might not be successful in different contexts. Second, these strategies, if repeatedly used, might become anti-patterns for debugging that are important to know about and to avoid. Finally, strategy success can be user-dependent. Murphy *et al.* [14] also found that the same strategy can be effective or ineffective, depending on the way it is used.

Table 6 outlines the number of times our participants used each strategy per error type, the number of errors found per strategy, and the percentage of total errors found by each strategy. We report on all seven strategies. The most successful strategies are **Expectation Confirmation** and **Search Engine-Driven Approach**. The **Expectation Confirmation** strategy success is influenced by

the markdown present in our notebooks. The markdown description set expectations for our participants. When the participants read the code following the descriptive text, they contrasted their expectations of what the code was supposed to do with what the code actually did. We note that in practice, Pimentel *et al.* found that notebooks contain very little markdown [17].

Additionally, we note that the efficacy of the **Search Engine-Driven Approach** is associated with the popularity of using online resources to guide users of Jupyter Notebooks [11], as pointed out by participants P7, P8, and P12. Koenzen *et al.* similarly determined that code reuse in Jupyter Notebooks most commonly comes from searches on the web, most often from websites that provide a tutorial, followed by API documentation [11].

5 DISCUSSION

We discuss the implications of our work to Jupyter notebook users, notebook tool designers, and educators. We also provide insights about the differences in debugging notebooks and non-notebook code, and the threats to the validity of our work.

5.1 Implications

In general, the error-finding strategies we identified point to the need for more tool support when developing Jupyter Notebooks to bring them to the same level as support in more mature non-notebook code tools. For example, the release of Jupyter Lab 3.0 introduced a visual debugger that can be used to step through code or to check the value of a variable [21]. This need for more tool support is suggested by other studies as well [6, 28].

We also uncovered two strategies that are not common in other approaches and may be specific to notebooks: *Re-implement to Check*

and *Start With What You Know*. We discuss the implications of these two strategies for notebook stakeholders below.

Re-implement to Check:

Tool designers could implement a tool which supplies the user with code snippets that use a different implementation so they could compare if the results are the same.

Users, if unsure what a particular code cell does, could be advised to re-implement the code to increase their understanding of the code in question and make it easier to identify an error.

Educators, when teaching students how to perform a task, could help students be aware that there may be more than one correct implementation. This would mitigate the false assumption that unfamiliar implementations (e.g., Pythonic list comprehensions) are incorrect.

Start With What You Know:

Tool designers could provide complexity measures for code cells so that the user can compare their own previous experience with the complexity of the cell to gauge where to start.

Users could be advised to start by self-reflecting on their skills in code understanding (e.g., data cleaning vs. statistical analysis) and start the process of error identification in cells by leveraging that skill. This may make the process of error finding easier as the user is more familiar with this approach and can build confidence in error finding.

Educators should understand what students are most familiar with (statistical, code, data domains) and then help them build knowledge in other areas. They could include a component on Jupyter or other notebook-specific debugging skills, such as the shift tab shortcut to access documentation.

5.2 Comparing Debugging Notebooks and Debugging Non-Notebook Code

The development of non-notebook code differs from the development of computational notebooks. The type of problem managed in a notebook involves more data wrangling, experimentation, and analysis code. Following the study which inspired our research [13], our study separated these into potential problems with statistics, programming, and data / domain knowledge. The notebook environment also has a literate programming component that goes beyond code comments, with markdown cells that can be used to describe the purpose of the code.

Furthermore, non-notebook IDEs have robust tool support for debugging, for example, setting breakpoints in IntelliJ. However, in the traditional computational notebook interface (say Jupyter Notebook), debugging is not specifically supported by the tool. Data science tools are actively working to fix this, for example, Jupyter Lab's debugger [21] and RStudio's debugging interface. IDEs are also now able to integrate notebook code into the IDE directly, such as with Visual Studio Code.

Given these differences, we ask whether error identification approaches for Jupyter Notebooks are also different. This study identified several strategies participants used to identify errors in Jupyter Notebooks. Other researchers have identified strategies for debugging non-notebook code. Table 7 outlines strategies identified by

Murphy *et al.* and Whalley *et al.*, that are similar to those we have identified [14, 26].

Table 7: Strategies Similar to Those We Identified

Strategies We Identified	Similar Strategies
Search Engine-Driven Approach	Using Resources [14]
Assume and (Sometimes) Check	Information Gathering [26], Bug Location [26]
Expectation Confirmation	Pattern Matching [14]
Once-over	Gain Domain Knowledge [14], Understanding the Code [14], Static Code Comprehension [26]
Re-implement to Check	N/A
Key Information	Understanding Code [14], Static Code Comprehension [26]
Start With What You Know	N/A

The **Search Engine-Driven Approach** strategy is closely related to the **Using Resources** strategy identified by Murphy *et al.* in [14]. Both strategies involve the use of documentation and tutorials. The difference between these two strategies is that we observed our participants using the search engine as their gateway to many resources. Murphy *et al.* make no mention of the search engine.

Both the **Information Gathering** and **Bug Location** strategies identified by Whalley *et al.* mention the use of speculation and guessing about the locations and causes of bugs. Alaboudi and LaToza also report on using hypotheses as a debugging aid [1]. Our **Assume and (Sometimes) Check** strategy is similar, based on making an assumption and optionally checking that assumption.

In their description of the **Pattern Matching** strategy, Murphy *et al.* state that their participants found bugs due to things not "looking right". In our notebook study, this was made more explicit than the heuristics Murphy *et al.* describe. Our participants were able to identify errors when a code cell seemed like it was not correct based on a description given in a markdown cell (the **Expectation Confirmation** strategy). The breakdown of an expectation could be thought of as pattern matching as our participants were attempting to match the pattern of what they were told the code was trying to accomplish to what they could observe the code doing. However, the presence of explicit documentation makes this strategy quite successful (at least for our example notebooks).

The **Once-over** and **Key Information** strategies identified by us are both similar to the **Understanding Code** and **Static Code Comprehension** strategies identified by Murphy *et al.* and Whalley *et al.*, respectively. In addition, the **Once-over** strategy is similar to the **Gain Domain Knowledge** strategy identified by Murphy *et al.* Both of our strategies were used in order to gain understanding about the contents of the notebook, and involve comprehending both the code and the markdown, much like the **Understanding Code** and **Static Code Comprehension** strategies are about comprehending code. The **Once-over** strategy was used to gain domain knowledge in the sense that the four participants who used this strategy did so to gain a brief understanding of the domain the notebook covered.

One key difference between notebooks and non-notebook code is that code cells are capable of independent output, closer to a Read-Eval-Print Loop session than debugging a complete source file. This difference may be to blame for why the **Re-implement to Check** and **Start With What You Know** strategies were not observed in other literature related to debugging non-notebook code. As the code in notebooks is often more granular and independent, these strategies are more viable when used in a notebook debugging context. This independence and high granularity allows for easier isolation of changes and re-implementations as a single unit of code in notebooks than in non-notebook code. The value of these strategies, however, may be dependent on the user's level of experience. For example, the **Re-implement to Check** strategy would more likely be adopted by users who know more than one way to implement a given task. On the other hand, the **Start With What You Know** strategy is more likely to be used by novice users that may want to stay within their comfort zone for as long as possible.

We found that debugging non-notebook code differs from debugging computational notebooks in a few ways. One, the type of development is different: there are more data science-related tasks such as data wrangling. Two, the development tools are at different levels of maturity when it comes to debugging support. Three, while five out of seven of the strategies we observed are related to non-notebook code debugging strategies identified in the literature, we found that two strategies were not found in non-notebook code studies. We also saw differences in how **Expectation Confirmation** and **Assume and (Sometimes) Check** are conducted in practice, given the way a notebook isolates individual code cells.

6 THREATS TO VALIDITY

In the following, we address the validity of this study in the context of qualitative research [8, 12].

Internal validity. We did not impose time-constraints on our participants, and they were assured our study was not a test of their skill. However, given the nature of the task, it is possible our participants felt pressure to perform well. Due to this pressure, participants may have overlooked errors in the Jupyter Notebooks. However, during the interviews we conducted immediately after the tasks, we did not detect that our participants felt any undue stress due to the study.

Construct validity. Our study prompt and task description may have influenced participants to perform actions which were not part of their typical error identification process in Jupyter Notebooks. For instance, modifying the notebook, searching documentation, and using the internet for help may not have been naturalistic behaviours. To mitigate this threat, we adopted multiple strategies, such as two rounds of pilots, to ensure comprehensibility and raise the realism of the tasks. In addition, task descriptions and scripts were reviewed and validated by a domain expert, and the task was confirmed to be within the recruited participants' skill level.

External validity. The primary threat to external validity is how we recruited and selected participants. We used convenience sampling methods to recruit participants from upper-level undergraduate and graduate-level courses at the university. Therefore, most

of our participants were students who used Jupyter Notebooks for school assignments and not professionally. However, we designed the tasks according to our participants' skill levels, and the context of tasks was fairly approachable (elections and sports) by any participant independently of their academic background. Our participants did not express unfamiliarity with the domain. However, some participants expressed unfamiliarity with specific packages imported into the notebook, namely Pandas and Plotnine. Another threat to external validity is the inclusion of markdown cells in the notebooks which may not reflect real-world notebooks [17].

Reliability. The open coding process was performed by one researcher, the first author of this paper. To reduce potential researcher bias and subjectivity, we conducted several discussion sessions to iteratively build a codebook. We confirmed with the feedback of an expert reviewer, the fourth author of this paper, to raise the reliability and maturity of our findings.

7 CONCLUSION

We conducted an observational study with fourteen participants, mostly university students from varying technical backgrounds, and observed the strategies these Jupyter Notebook users employed to identify errors seeded in four sample notebooks. The most commonly used strategy we observed was using the search engine to find external help such as API documentation or websites that provide a tutorial. However, the most successful strategy was Expectation Confirmation, when they discovered a mismatch between the description and the code itself. We identified some implications for practice, including the need for better debugging support in notebooks, and showed that while there are similarities with non-notebook code, debugging in notebooks leverages notebook-only properties such as code cell independence and hidden state. Out of the seven identified strategies, five had been previously identified in the literature as debugging strategies for non-notebook code, while two are novel to the notebook environment. Future work could involve collaborating with members of the original RMarkdown study to compare the error finding strategies of data scientists between the respective studies. We hope our insights will help both notebook tool designers and educators improve how data scientists discover errors more easily in their notebooks.

ACKNOWLEDGMENTS

We thank Kelly Bodwin and Ian Flores Siaca for sharing the initial RMarkdown notebooks; Hunter Glanz for reviewing our Python notebook translations; and Amelia McNamara, Amal Abel-Ghani, Philipp Burckhardt, Allison Theobald and Greg Wilson for the study idea. We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada and Venture for Canada.

REFERENCES

- [1] Abdulaziz Alaboudi and Thomas D. LaToza. 2020. Using Hypotheses as a Debugging Aid. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, Dunedin, New Zealand, 1–9. <https://doi.org/10.1109/VL/HCC50065.2020.9127273>
- [2] Abdulaziz Alaboudi and Thomas D. LaToza. 2021. An Exploratory Study of Debugging Episodes. *CoRR* abs/2105.02162 (2021). [arXiv:2105.02162](https://arxiv.org/abs/2105.02162) <https://arxiv.org/abs/2105.02162>
- [3] Kelly Nicole Bodwin, Ian Flores Siaca, and Derek Robinson. 2022. Materials for "Looks okay to me": A study of best practice in data analysis code review. (April 2022). <https://doi.org/10.5281/zenodo.6419727>
- [4] Andrew W Brown, Kathryn A Kaiser, and David B Allison. 2018. Issues with data and analyses: Errors, underlying themes, and potential solutions. *Proceedings of the National Academy of Sciences* 115, 11 (2018), 2563–2570.
- [5] Alberto Cardoso, Joaquim Leitão, and César Teixeira. 2019. Using the Jupyter Notebook as a Tool to Support the Teaching and Learning Processes in Engineering Courses. In *The Challenges of the Digital Transformation in Education*, Michael E. Auer and Thrasvoulos Tsiatsos (Eds.). Springer International Publishing, Cham, 227–236.
- [6] Souti Chattopadhyay, Ishita Prasad, Austin Z. Henley, Anita Sarma, and Titus Barik. 2020. 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*. Association for Computing Machinery, New York, NY, USA, 1–12. <https://doi.org/10.1145/3313831.3376729>
- [7] Pierre Dragicevic, Yvonne Jansen, Abhaneel Sarma, Matthew Kay, and Fanny Chevalier. 2019. Increasing the Transparency of Research Papers with Explorable Multiverse Analyses. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. <https://doi.org/10.1145/3290605.3300295>
- [8] Egon G Guba. 1981. Criteria for assessing the trustworthiness of naturalistic inquiries. *Educational Technology research and development* 29, 2 (1981), 75–91.
- [9] Andrew Head, Fred Hohman, Titus Barik, Steven M Drucker, and Robert DeLine. 2019. Managing messes in computational notebooks. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. 1–12.
- [10] Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E John, and Brad A Myers. 2018. 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)*. Association for Computing Machinery, New York, NY, USA. <https://doi.org/10.1145/3173574.3173748>
- [11] Andreas P. Koenzen, Neil A. Ernst, and Margaret-Anne D. Storey. 2020. Code Duplication and Reuse in Jupyter Notebooks. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, Dunedin, New Zealand, 1–9. <https://doi.org/10.1109/VL/HCC50065.2020.9127202>
- [12] Irene Korstjens and Albine Moser. 2018. Series: Practical guidance to qualitative research. Part 4: Trustworthiness and publishing. *European Journal of General Practice* 24, 1 (2018), 120–124.
- [13] Amelia McNamara, Amal Abel-Ghani, Ian Siaca Flores, Kelly Bodwin, Allison Theobald, Philipp Burkhardt, and Greg Wilson. 2022. Looks okay to me: A study of best practice in data analysis code review. (2022). in preparation.
- [14] Laurie Murphy, Gary Lewandowski, Renée McCauley, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: the good, the bad, and the quirky—a qualitative analysis of novices' strategies. *ACM SIGCSE Bulletin* 40, 1 (2008), 163–167.
- [15] Peter Parente. 2014. Estimate of public Jupyter notebooks on GitHub. (2014). <https://github.com/parente/nbestimate> (Accessed on 01/16/2022).
- [16] Jeffrey M Perkel. 2018. Why Jupyter is data scientists' computational notebook of choice. *Nature* 563, 7732 (2018), 145–147. <https://doi.org/10.1038/d41586-018-07196-1>
- [17] João Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. 2019. A Large-Scale Study About Quality and Reproducibility of Jupyter Notebooks. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, Montreal, QC, Canada, 507–517. <https://doi.org/10.1109/MSR.2019.00077>
- [18] El Kindi Rezig, Ashrita Brahmaraoutu, Nesime Tatbul, Mourad Ouzzani, Nan Tang, Timothy Mattson, Samuel Madden, and Michael Stonebraker. 2020. Debugging large-scale data science pipelines using Dagger. *Proceedings of the VLDB Endowment* 13, 12 (Aug. 2020), 2993–2996. <https://doi.org/10.14778/3415478.3415527>
- [19] Adam Rule, Aurélien Tabard, and James D. Hollan. 2018. Exploration and Explanation in Computational Notebooks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. 1–12. <https://doi.org/10.1145/3173574.3173606>
- [20] Adam A Smith. 2016. Teaching computer science to biologists and chemists, using jupyter notebooks: tutorial presentation. *Journal of Computing Sciences in Colleges* 32, 1 (2016), 126–128. <https://doi.org/10.5555/3007225.3007252>
- [21] Jeremy Tuloup. 2021. JupyterLab 3.0 is released!. The 3.0 release of JupyterLab brings (2021). <https://blog.jupyter.org/jupyterlab-3-0-is-out-4f58385e25bb> (Accessed on 07/28/2021).
- [22] Jiawei Wang, Tzu-yang Kuo, Li Li, and Andreas Zeller. 2020. *Restoring Reproducibility of Jupyter Notebooks*. Association for Computing Machinery, New York, NY, USA, 288–289. <https://doi.org/10.1145/3377812.3390803>
- [23] Jiawei Wang, Li Li, and Andreas Zeller. 2020. Better Code, Better Sharing: On the Need of Analyzing Jupyter Notebooks. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER '20)*. Association for Computing Machinery, New York, NY, USA, 53–56. <https://doi.org/10.1145/3377816.3381724>
- [24] Nathaniel Weinman, Steven M. Drucker, Titus Barik, and Robert DeLine. 2021. Fork It: Supporting Stateful Alternatives in Computational Notebooks. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. <https://doi.org/10.1145/3411764.3445527>
- [25] Charles J. Weiss. 2021. A Creative Commons Textbook for Teaching Scientific Computing to Chemistry Students with Python and Jupyter Notebooks. *Journal of Chemical Education* 98, 2 (2021), 489–494. <https://doi.org/10.1021/acs.jchemed.0c01071>
- [26] Jacqueline Whalley, Amber Settle, and Andrew Luxton-Reilly. 2021. *Novice Reflections on Debugging*. Association for Computing Machinery, New York, NY, USA, 73–79. <https://doi.org/10.1145/3408877.3432374>
- [27] Kanit Wongsuphasawat, Yang Liu, and Jeffrey Heer. 2019. Goals, Process, and Challenges of Exploratory Data Analysis: An Interview Study. *CoRR* abs/1911.00568 (2019). [arXiv:1911.00568](https://arxiv.org/abs/1911.00568) <https://arxiv.org/abs/1911.00568>
- [28] Chenyang Yang, Shurui Zhou, Jin LC Guo, and Christian Kästner. 2021. Subtle bugs everywhere: Generating documentation for data wrangling code. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Vol. 11.
- [29] Andreas Zeller. 2009. CHAPTER 1 - How Failures Come to Be. In *Why Programs Fail (Second Edition)* (second edition ed.), Andreas Zeller (Ed.). Morgan Kaufmann, Boston, 1–23. <https://doi.org/10.1016/B978-0-12-374515-6.00001-0>