Towards Understanding Fine-Grained Programming Mistakes and Fixing Patterns in Data Science

WEI-HAO CHEN, Purdue University, USA
JIA LIN CHEOH, Purdue University, USA
MANTHAN KEIM, Purdue University, USA
SABINE BRUNSWICKER, Purdue University, USA
TIANYI ZHANG, Purdue University, USA

Programming is an essential activity in data science (DS). Unlike regular software developers, DS programmers often use Jupyter notebooks instead of conventional IDEs. Moreover, DS programmers focus on statistics, data analytics, and modeling rather than writing production-ready code following best practices in software engineering. Thus, in order to provide effective tool support to improve their productivity, it is important to understand what kinds of errors they make and how they fix them. Previous studies have analyzed DS code from public code-sharing platforms such as GitHub and Kaggle. However, they only accounted for code changes committed to the version history, omitting many programming mistakes that are resolved before code commits. To bridge the gap, we present an in-depth analysis of the fine-grained logs of a DS competition, which includes 390 Jupyter Notebooks written by 67 participants over six weeks. In addition, we conducted semi-structured interviews with 10 DS programmers from different domains to understand the reasons behind their programming mistakes. We identified several unique programming mistakes and fix patterns that had not been reported before, highlighting opportunities for designing new tool support for DS programming.

CCS Concepts: • **Software and its engineering → Development frameworks and environments**; **Software maintenance tools**.

Additional Key Words and Phrases: Computational Notebook, Data Science, Programming Practice

ACM Reference Format:
Wei-Hao Chen, Jia Lin Cheoh, Manthan Keim, Sabine Brunswicker, and Tianyi Zhang. 2025. Towards Understanding Fine-Grained Programming Mistakes and Fixing Patterns in Data Science. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE082 (July 2025), 23 pages. https://doi.org/10.1145/3729352

1 Introduction

Data Science (DS) plays an important role in providing data-driven insights for decision-making in different domains, such as finance, retail, and marketing. Programming is an essential activity in data science. Data scientists need to write code for data cleaning, analysis, modeling, visualization, etc. However, compared with regular software developers, many data scientists are not specialized in programming [1, 38]. Their expertise lies more in statistics, information science, and target domains such as finance and marketing. Furthermore, data scientists often use computational notebooks, such as Jupyter Notebook [39, 86] and R Markdown [73, 84], for programming. Unlike conventional programming environments, programs in computational notebooks are organized as

Authors' Contact Information: Wei-Hao Chen, Purdue University, West Lafayette, USA, chen4129@purdue.edu; Jia Lin Cheoh, Purdue University, West Lafayette, USA, jcheoh@purdue.edu; Manthan Keim, Purdue University, West Lafayette, USA, keimm@purdue.edu; Sabine Brunswicker, Purdue University, West Lafayette, USA, sbrunswi@purdue.edu; Tianyi Zhang, Purdue University, West Lafayette, USA, tianyi@purdue.edu.

ACM 2994-970X/2025/7-ARTFSE082
https://doi.org/10.1145/3729352

a collection of *cells*, which can be executed in a non-linear order [17, 67]. This facilitates a mix of live coding and visualizations [78], making them ideal for data exploration and analysis [70].

Given the differences DS programming and conventional software development, it is essential to understand the types of programming mistakes DS programmers make and how they fix them, in order to design effective tool support for data science. Prior studies [12, 24, 55, 56, 59] have analyzed Jupyter notebooks mined from platforms such as GitHub and Kaggle. However, these analyses are limited to static snapshots preserved in version histories, thereby omitting mistakes that were corrected prior to being committed.

To the best of our knowledge, no existing studies have investigated the error distribution, debugging activities, and fixing patterns of data scientists. To bridge this gap, we organized a six-week data science competition and collected fine-grained programming and debugging data from 390 Jupyter notebooks written by 67 participants. Compared to previous work, we analyzed fine-grained changes made by the participants, the execution histories of each cell, and the output logs of each execution. We summarized participants' debugging activities in a state diagram and found that they adopted a variety of strategies beyond editing the erroneous cell, often in an iterative manner, to debug and fix those errors.

Our study reveals several new findings. First, more than half of the coding errors occurred during the data exploration (32%) and data preprocessing (26%) stages. Second, many errors were `ValueErrors` (21%) and `NameErrors` (20%), in contrast to general Python scripting, where `TypeError` was the most common error [53]. While many errors can be fixed locally in the same code cell, many errors (30%) require editing another cell (i.e., *global fixes*). Among all local fixes, the most common editing pattern is *Change Parameters* (35%), followed by *Fix Syntax Errors* (12%), and *Rename Variable* (12%). In this work, we focused on erroneous cells that produced compilation or runtime errors. Other types of mistakes, such as incorrect visualizations, remain an interesting direction for future work, as discussed in Section 7.

To gain a deeper understanding of why DS programmers make these errors and what kind of tool support they need, we conducted semi-structured interviews with 10 DS programmers (6 from industry and 4 from academia). Most interviewees (8/10) identified data preprocessing and data exploration as the most error-prone stages. Common issues in data preprocessing include dirty data (9/10), unclear data formats (8/10), and a lack of domain knowledge (5/10). In terms of issues in data exploration, unfamiliarity with the dataset (6/10) and too many columns (4/10) are the reasons why it is error-prone. Additionally, complex cell dependency issues could complicate debugging, as reported by several participants (6/10). We also found that DS programmers relied on print statements and executing cells one by one to narrow down errors.

In summary, this work makes the following contributions:

- **Programming Mistake Distribution:** We examined a taxonomy of coding errors and analyzed their distributions across different stages of the data science workflow.
- **Debugging & Fixing Patterns:** We conducted an in-depth analysis of debugging and fixing practices in DS programming. We identified 5 distinct debugging operations and examined the transition states in the debugging traces. Furthermore, we identified 12 fixing patterns that DS programmers used to fix errors.
- **Interview Study:** We conducted semi-structured interviews with 10 DS programmers from industry and academia to understand the reasons behind the observed errors.

2 Background

Jupyter Notebook [3] is a programming environment that allows interactive literate programming [40] and documentation. A notebook is composed of *cells*. There are three types of cells: *code* *cells* for writing code, *markdown cells* for documentation, and *output cells* where plots and results are rendered. Figure 1 shows an example.

Although code cells are arranged in a top-down manner, they can be executed in any order. When executing code cells, the Python kernel in Jupyter Notebook maintains the execution history and the runtime values of variables in previously executed code cells. In Jupyter notebooks, each code cell is labeled with a number on the left, indicating the *index* of the cell in the execution history. For example, in Figure 1, the programmer first executed the two code cells, made some edits to the second code cell, and then re-executed the second cell. As a result, the first code cell is indexed as [1], while the second code cell is indexed as [3] since it was executed twice. In this work, we focused on code cells as our goal is to identify programming mistakes and fix patterns in code. For simplicity, we refer to a *code cell* as a *cell* henceforth.

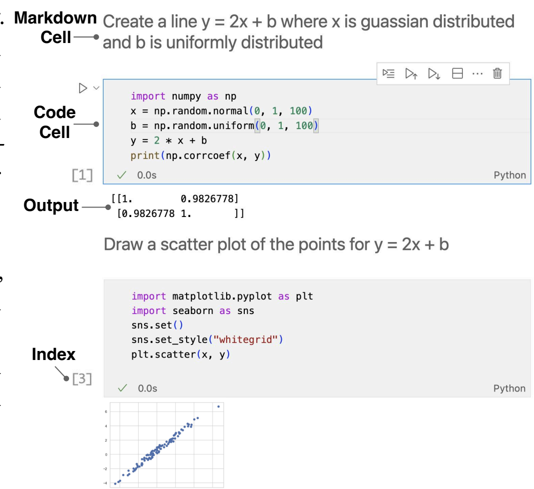


Fig. 1. An example of a notebook.

3 Research Questions

We investigate the following research questions:

- **RQ1.** *What kinds of mistakes do data science programmers make when writing code?* Previous work [5, 20] focuses on collecting bugs and observing programming practices using coarse-grained data mined from GitHub and Stack Overflow. However, these sources typically contain snapshots of notebooks that omit intermediate errors that DS programmers made and fixed before code commits. Furthermore, these studies do not provide insights into which stages of the data science workflow are more error-prone. To answer this RQ, we collected data from a six-week data science competition and conducted an in-depth analysis of the errors made by our participants. We analyzed the frequency of each error type and their distribution across different data science stages. We found that most errors occur in the early stages, such as data preprocessing and data exploration, as detailed in Section 5.1.
- **RQ2.** *What activities and operations do data science programmers perform to diagnose those programming mistakes?* Several studies have explored debugging behaviors in traditional IDEs [7, 9]. However, Jupyter Notebook is significantly different from these environments, e.g., allowing running cells in an arbitrary order, no breakpoints, no stepping through, etc. Currently, there is a lack of studies and datasets that examine debugging practices in data science. To answer this RQ, we analyzed DS programmers' debugging behavior and summarized it in a state diagram, as shown in Figure 5. We found that debugging activity is highly iterative, as discussed in Section 5.2.
- **RQ3.** *What kinds of edits do data science programmers make to fix errors?* Previous studies [45, 87] have explored fixing patterns in debugging Python scripts. Still, none of them focus on fine-grained editing patterns in computational notebooks. To answer this RQ, we identified a taxonomy of 12 frequent fixing patterns. Moreover, we analyzed their usage frequency, most co-occurring fixing patterns, and usefulness in solving different errors. We found that *Changing Parameters* is the most frequently used fix pattern, as discussed in Section 5.3.

RQ4. *Why do data science programmers make those errors and what kinds of tool support do they need?* In previous RQs, we analyzed error distributions and debugging & fixing patterns in our DS competition dataset. However, these questions did not help us understand the reasons behind these errors and whether these patterns apply beyond the DS competition setting. To address this RQ, we conducted follow-up interviews with 10 DS programmers (6 from industry and 4 from academia) from different domains (e.g., IT, Finance, Insurance, etc.) to gain a deeper understanding of their debugging practices and challenges. Our results show that complex cell dependencies could hinder the debugging process. DS programmers desired tools to manage complex cell dependencies and track hidden variable states to prevent unexpected behaviors, as discussed in Section 5.4.

4 Methodology

To answer the RQs, we organized a six-week online data science competition and collected fine-grained programming information through Jupyter Notebook instrumentation. We then conducted follow-up interviews to gain a deeper understanding of the programming hurdles and needs of DS programmers. Figure 2 provides an overview of our analysis procedure.

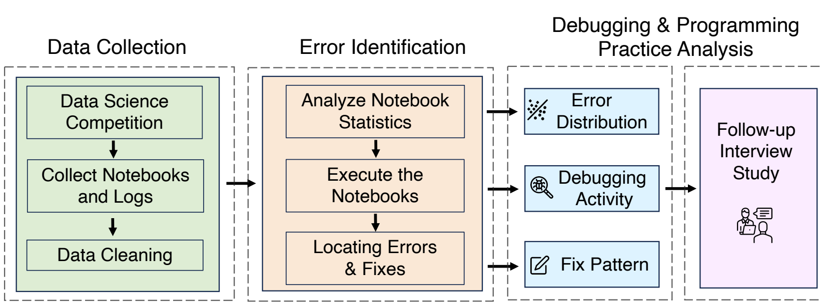


Fig. 2. An Overview of the Analysis Procedure.

4.1 Data Collection

We organized a six-week online DS competition and advertised it through social media and emails. Participants were asked to develop a data science project to build and evaluate a prediction model using an unprocessed, real-world dataset from a state department. We gave them a document about the dataset and the end goal. They needed to do the planning themselves and build the DS pipeline from scratch. Thus, tasks such as data preprocessing, feature engineering, and modeling, are integral parts of this project. This mimics a common working scenario for data scientists—they receive raw data from a customer or manager and use statistical methods to analyze it [38]. We chose prediction rather than other end goals (e.g., insight discovery) because prediction is a common end goal in data science [71] and provides a clear metric for evaluating participants' performance.

To encourage participation, the top three winners would receive \$500 awards. While many people signed up for the competition and actively participated, 67 successfully completed the competition, i.e., building a model that is runnable on our held-out test set. We recruited participants from diverse backgrounds, including 24% from Computer Science, 20% from Business, 20% from Information Systems, 8% from Data Science, 6% from Mathematics/Statistics, 4% from Mechanical Engineering, 4% from Electrical and Computer Engineering, and 14% from other majors.

These participants included undergraduate students (22%), master's students (33%), Ph.D. students (16%), industry practitioners (25%), and others (4%). They had a median of 2.5 years of programming experience. Since our competition allowed participants to re-submit their notebooks, we collected a total of 1,310 notebooks from these 67 participants.

Jupyternotebok Instrumentation. We asked participants to use an instrumented Jupyter notebook to write code. We used IPython's build-in commands [2] to track fine-grained cell edits and execution history. Specifically, we took a snapshot of each cell whenever the programmer executed a cell, as shown in Figure 3. This instrumentation design was inspired by a previous finding that DS programmers often made frequent edits and re-executed cells to check intermediate results [86].

An alternative design could be to capture snapshots at fixed intervals, but this would capture incomplete edits if a programmer was still editing at the end of an interval. This alternative design could also lead to redundant snapshots if no edits were made during a period of time. Another alternative design could be to collect keystroke data. However, it would cause excessive I/O overhead and raise privacy concerns. Thus, after careful consideration, we chose the current instrumentation design.

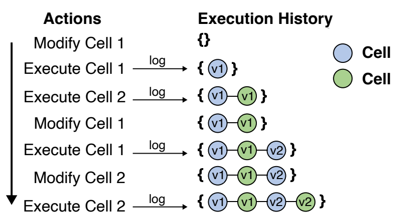


Fig. 3. An Example of Execution History.

Data Cleaning. We carefully filtered out any duplicate submissions or incomplete notebooks from the 1,310 notebooks. Since some participants made more re-submissions than others, we did not want their mistakes to be over-represented in our analysis. Thus, we performed a downsampling to sample at most 6 notebook submissions from each participant. Note that we chose to analyze multiple submissions from a participant rather than only the final submission, since we observed that participants typically did not make the same mistakes again in later submissions. Only analyzing the final submission would miss those early mistakes. Moreover, the final submission often involve small edits such as hyperparameter tuning. Participants barely made coding mistakes in the final submission. In the end, we sampled 390 notebook submissions. We determined our sample size using Cochran's formula [41, 88] with a 95% confidence interval and a 5% margin of error.

Table 1 shows the statistics of our sample set. Specifically, # of Code Cells refers to the number of code cells that appeared in the execution history of a notebook submission. # of Imported Modules refers to the number of unique modules imported from third-party packages in a notebook submission. Cell Coupling measures the interdependency between cells in a notebook [24].

Table 1. Dataset Statistics

	Min	Med	Max
# Code Cells	4	24	318
# Imported Modules	0	8	31
Lines of Code	6	73	1372
Lines of Comments	0	3	292
Cyclomatic Complexity	1	1	24
Cell Coupling	0	14	12694

4.2 Programming Error Identification and Analysis

To answer RQ1, we need to first locate the programming mistakes from the log data. As shown in Figure 3, our log data contains snapshots of code cells in the execution history but does not contain the output of each execution due to storage limits. Thus, we re-ran each cell in the history to identify whether it contained a programming mistake. An erroneous cell is identified if its output contains an error message. Since some erroneous cells were repeatedly executed and appeared multiple times in the execution history, we removed duplicated cells that threw the same error message. In total, we identified 839 unique erroneous cells.

After some manual analysis, we noticed that sometimes, participants made multiple attempts to fix an error and re-executed the erroneous cell after each attempt. This led to multiple erroneous cells with slight differences appearing in the execution history, although these cells originated from the same erroneous cell. For such erroneous cells with the same origin, we only kept the first cell so that we did not inflate the occurrence of their errors in our analysis. Since there were 839 erroneous cells, which was not too many, the first author manually went through them and filtered the cells that originated from the same cell. This ended up with 529 erroneous cells.

Finally, we need to classify each cell into different stages of the DS workflow to understand the error distribution. The first author manually inspected the 529 erroneous cells and classified them into one of the ten DS stages defined by Ramasamy et al. [56], including *Data Loading*, *Data Preprocessing*, *Data Exploration*, *Modeling*, *Evaluation*, *Prediction*, *Visualization*, *Result Saving*, *Comment Only*, and *Helper Function*. While Ramasamy et al. trained a machine learning model to classify code cells to different DS stages, we chose manual analysis for two reasons. First, the classification model developed by Ramasamy et al. only achieved a 71% F1 score, which would inevitably introduce classification errors in our analysis. Second, since we only had 529 erroneous cells to classify, the sample size was feasible for manual classification.

4.3 Debugging Activity Analysis

To answer RQ2, we need to analyze what operations have been performed before fixing an error. For each erroneous cell, we extract the log data between the erroneous cell and the corresponding fixed cell in the execution history. We call this a *debugging trace*, as illustrated in Figure 4. Recall that we have identified the erroneous cells in Section 4.2, but we have not identified the corresponding fixed cells yet. Thus, for each erroneous cell, the first author manually inspected the subsequent cells in the execution history to identify the first cell that looked similar to the erroneous cell but no longer threw the error message.

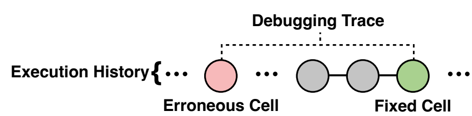


Fig. 4. An Example of a Debugging Trace.

We identified five basic operations in the debugging traces between the 529 pairs of erroneous cells and the corresponding fixed cells:

- **Rerun Previous Cell**: The programmer re-executed a previous cell, **not** the erroneous cell, without making any modifications.
- **Rerun Erroneous Cell**: The programmer re-executed the original erroneous cell without making any modifications.
- **Edit Erroneous Cell**: The programmer edited the original erroneous cell.
- **Edit Previous Cell**: The programmer edited a previous cell, **not** the erroneous cell.
- **Create New Cell**: The programmer created a completely new cell.

Based on these operations, we lifted a state diagram from the 529 debugging traces, as shown in Figure 5. The start state is an erroneous cell, while the end state is the error being fixed. Each state in the middle represents a basic operation. The edge between two state nodes is labeled with the transition probability (i.e., the probability of one state occurring after another state in the data). For instance, `Edit Previous Cell` $\xrightarrow{35\%}$ `Rerun Erroneous Cell` means that from the 529 debugging traces, 35% of the `Edit Previous Cell` operation are followed by the `Rerun Erroneous Cell` operation.

4.4 Fixing Pattern Analysis

To answer RQ3, we need to identify the edits performed by participants. While our dataset only includes 529 debugging traces, multiple cells may be edited in a debugging trace and each cell may also be edited in several locations. Thus, to reduce the manual effort, we built a rule-based method to automatically infer the fix patterns based on program differences computed by GumTree [23]. We describe the details below.

Manual Inspection. To build the rule-based method, we first sampled 100 debugging traces and followed the open coding procedure in qualitative analysis [10] to identify the patterns to be inferred. Since each debugging trace may involve edits to multiple cells, we compared the first and

Table 2. Fix Patterns and Inference Rule Implementation

Fix Pattern	**Inference Rule**
Add New Method Call	Insert(t_o, t_n, i) ∧ NodeType(t_n, MethodCall)
Add New Attribute	Insert(t_o, t_n) ∧ NodeType(t_n, Attribute)
Change Parameters	Update(t_o, t_n) ∧ (NodeType(t_n, Parameters) ∨ NodeType(t_n, ParameterValue)) ∧ (NodeSet(t_o) ≠ NodeSet(t_n))
Change Key/Index	Update(t_o, t_n) ∧ (NodeType(t_n, String) ∨ NodeType(t_n, Number)) ∧ IsInBrackets(NodeValue(t_o, oldVal), NodeValue(t_n, newVal))
Rename Variable	Rename(t_o, t_n) ∧ NodeType(t_n, Variable) ∧ NodeValue(t_o) ≠ NodeValue(t_n) ∧ (NodeSet(t_o) ≠ NodeSet(t_n))
Remove Method Call	Delete(t_o) ∧ NodeType(t_o, MethodCall)
Remove Attribute	Delete(t_o) ∧ NodeType(t_o, Attribute)
Delete Old Lines	Delete(t_o) ∧ (NumLines(t_o) > NumLines(t_n))
Add New Lines	Insert(t_o, t_n, i) ∧ (NumLines(t_n) > NumLines(t_o))
Commented Out Code	Update(t_o, t_n) ∧ NodeType(t_n, Comment) ∧ Contains(NodeValue(t_o), Trim(NodeValue(t_n), "#"))
No Changes	Equal(t_o, t_n)
Fix Syntax Errors	SyntaxError(t_o) ∧¬ SyntaxError(t_n)

GumTree Edit Predicate	**Syntactic Predicate**	**Auxilary Predicate**
Insert(t_o, t_n, i) true if node t_n is inserted as the i-th child of t_o.	**NodeType(t, Name)** true if the t's type is Name.	**Rename(t_o, t_n)** true if node t_o is renamed as t_n.
	NodeValue(t) returns the value of node t.	**IsInBrackets(v)** true if the value v is within brackets.
Delete(t) true if node t is deleted in the AST tree.	**NodeSet(t)** returns set of nodes related to t.	**SyntaxError(t)** true if there's a syntax error in node t.
	Child(t_o, t_n) true if node t_o is a child of node t_n.	**Trim(t, string)** trims leading and trailing characters in t.
Update(t_o, t_n) true if node t_o is updated with node t_n.	**Contains(t, v)** true if the node t contains the code v.	**NumLines(t)** returns number of lines in the string t.
	Equal(t_o, t_n) true if nodes t_o and t_n are equal.	

last versions of any cells edited during the debugging trace. Two authors independently inspected the program differences and summarized the fixes in each debugging trace. After inspecting all 100 traces, they met together, compared the fixes they summarized, and came up with an initial taxonomy of fix patterns. They continued inspecting more debugging traces together and kept refining the taxonomy (e.g., adding new patterns, merging existing patterns, renaming patterns, etc.). They stopped after examining another 100 debugging traces, since the taxonomy converged. The final taxonomy includes 12 fix patterns, as shown in Column Fix Pattern in Table 2.

Pattern Inference Rules. For each fix pattern, we designed an inference rule based on the AST edits computed by GumTree [23]. Column Rule in Table 2 describes the implementation logic for each inference rule. Each rule is composed of three types of predicates:

- **GumTree Edit Predicates.** GumTree computes three types of basic edits on AST nodes, including *insert*, *delete*, and *update*. We represent them in three corresponding logic predicates, as described in Column GumTree Edit Predicate in Table 2.
- **AST Node Predicates.** AST node predicates describe the characteristics of AST nodes, such as AST node types, values, and structural relationships. Column Syntactic Predicate in Table 2 lists the syntactic predicates used to implement the inference rules.
- **Auxiliary Predicates.** Auxiliary predicates are helper functions that check specific properties of source code or perform specific operations on different types of data, such as identifying a particular syntax error in source code and timing a string value. Column Auxilary Predicate in Table 2 lists the semantic predicates used to implement the inference rules.

These predicates serve as the building blocks for defining the inference rule for each fix pattern, as illustrated in Column Inference Rule in Table 2. For example, *Change Parameters* checks whether any changes have been made to either the parameter name or the value of an existing parameter, then checks the node sets of both to ensure that an update is made in one of these two sets.

Accuracy of the Pattern Inference Rules. To evaluate our rule-based pattern inference method, we sampled 230 debugging traces not included in the initial manual inspection as the validation set. This sample size is statistically significant with a confidence level of 95% and a margin of error of 5%. To reduce bias, the second author, who was not involved in the initial manual inspection, labeled the fix patterns in this validation set. The ground truth contains 335 manually labeled fix patterns. Overall, our method infers the fix patterns with 83% precision and 84% recall.

4.5 Interview Study

To answer RQ4, we conducted semi-structured interviews to understand why the observed errors happen and what kinds of support do DS programmers need. We provide details of our interview protocol, participants' backgrounds, and analysis procedure in the following subsections.

Interview Protocol. We followed guidelines in empirical software engineering [60, 63] to design a semi-structured interview[1]. The interview began with a short introduction to our study and a request for permission[2]. Then, we asked high-level questions about: (1) the background of the interviewees, such as their current job and how long they have been working in DS, (2) common errors occurring in different DS stages, (3) their programming and debugging practices within Jupyter Notebooks, and (4) debugging features and support needed. We interviewed 10 DS programmers from both industry and academia. Each interview took between 30 to 40 minutes. The interviews were recorded and then transcribed for further analysis.

Participants. We recruited 10 data science practitioners through personal networks, industry collaborations, and social media. To ensure diversity, we selected participants with varied educational backgrounds and professional roles. Six held degrees in Computer Science, two in Statistics, one in Finance, and one in Accounting. In terms of roles, four were PhD students with extensive experience in data analysis and modeling, two were data engineers, two were software engineers, one was a data scientist, and one was a quantitative analyst. For programming experience, two participants reported having 2 to 5 years of experience, and eight had more than 5 years. Their data science experience also varied, with one having less than 2 years, four having 2 to 5 years, and five having more than 5 years.

Analysis. We transcribed interview recordings to text using the Microsoft audio transcription service. The first author conducted an open-coding phase [81] using a professional qualitative data analysis software called NVivo. This coding phase was done thoroughly by highlighting everything that is relevant or interesting. A code was generated by summarizing a relevant phrase or sentence with a short descriptive text.[3] The first author then conducted an inductive thematic analysis [14], grouping related codes into themes. We observed a data saturation at interview #8. After interview #8, no additional themes emerged from the remaining interviews. These emerging themes were regularly discussed with the entire research team. Additionally, the second author independently inspected the generated codes and themes, validating how the raw data supported them and adjusting their descriptions and boundaries. Finally, the two authors refined the codes and themes together over multiple sessions, addressing any disagreements.

5 Results

5.1 RQ1. Common Errors Made by DS Programmers

We report the common error types based on the error messages of the 529 errors below:

- *ValueError (21%)*: This error occurs when a function receives an argument with the correct data type but an inappropriate value. The example below shows a value error that occurs when feeding the wrong length of the prediction array to the `fit` function.

```
from sklearn.linear_model import LinearRegression
```

[1]Interview Guide: https://github.com/ferranschen/Notebook-Analysis-FSE2025/blob/main/InterviewStudy/Interview.pdf
[2]Consent Form: https://github.com/ferranschen/Notebook-Analysis-FSE2025/blob/main/InterviewStudy/Consent.pdf
[3]Code Book: https://github.com/ferranschen/Notebook-Analysis-FSE2025/blob/main/InterviewStudy/Code_Book.md

```python
import numpy as np
X = np.array([[1, 2], [2, 3], [3, 4]])
# y should be of length 3
y = np.array([1, 2])
model = LinearRegression()
model.fit(X, y)
```

- *NameError (20%)*: This error occurs when a variable, function, or class cannot be found. This error can occur frequently in Jupyter notebooks. The non-linear execution of cells in these notebooks can lead to undefined variables, functions, or classes.

```python
import pandas as pd
df = pd.DataFrame({'col1': [1, 2, 3]})
# Dataframe is not imported
print(dataframe)
```

- *AttributeError (12%)*: This error occurs when programmers attempt to access a method or attribute that does not exist in an object. This error occurs frequently because DS libraries often share similar functions. For instance, confusion may arise between the functions in NumPy [72] and Pandas [47]. The following example shows a misuse of the `to_frame` function of Pandas on a Numpy array.

```python
import numpy as np
import pandas as pd
arr = np.array([1, 2, 3, 4, 5])
# Calling a pandas function on a numpy array
arr.to_frame()
print(model.coef_)
```

- *KeyError (11%)*: This error occurs when a dictionary key is not found. For example, when users try to get a non-existent column in Pandas. The following example shows that an error occurs when accessing a non-existent column from a dataframe.

```python
import pandas as pd
df = pd.DataFrame({'col1': [1, 2, 3]})
# Accessing a non-existent column.
print(df['col2'])
```

- *SyntaxError (9%)*: This error happens for various reasons, such as missing brackets or semi-colons, misspelled keywords, and incorrect indentation.
- *TypeError (8%)*: This error occurs when an operation or function is applied to an object of an inappropriate type.
- *NotFoundError (5%)*: This error typically relates to file operations or HTTP requests where the requested resource is not found.
- *BadRequest (5%)*: This competition allows programmers to access datasets via Google BigQuery remotely. This error arises when attempting to fetch data from an invalid URL.
- *IndexError (3%)*: This error occurs when programmers try to access an element from a list using an incorrect index that does not exist.
- *ModuleNotFound (2%)*: This error occurs when a used module is not installed in the system.
- *Misc (2%)*: This category is usually a catch-all for errors that do not occur frequently.

The most commonly encountered errors are `ValueError 21%` , `NameError 20%` , and `AttributeError 12%`. This differs from typical Python scripting, where `TypeError` is the most prevalent error [53, 54]. Errors in DS code often stem from issues such as incorrect data types, references to undefined variables or data columns, and improper use of object attributes. These are common errors encountered when processing datasets in DS tasks. We provide more implications in Section 6.2.

Table 3. Error Distribution across DS Stages. Cells in red indicate the most frequent error in each stage.

Stage	AttrErr	TypeErr	ValErr	NameErr	NotFnd	BadReq	KeyErr	SyntaxErr	IndexErr	ModNotFnd	Misc	Total
Data Loading	9	3	0	25	28	27	6	2	1	0	0	101
Date Preprocessing	25	17	32	25	0	0	25	12	4	0	0	140
Data Exploration	24	18	37	31	0	0	20	22	14	0	4	170
Modeling	1	0	8	3	0	0	1	1	0	0	1	15
Prediction	0	1	2	1	0	0	0	1	0	0	2	7
Evaluation	2	0	0	1	0	0	0	0	0	0	0	3
Visualization	9	5	13	16	0	0	3	8	0	0	2	56
Result Saving	0	0	0	0	0	0	0	0	0	0	0	0
Comment Only	0	0	0	0	0	0	0	0	0	0	0	0
Helper Functions	0	0	19	1	0	0	0	4	0	12	1	37
Total	70	44	111	103	28	27	55	50	19	12	10	529

Furthermore, Table 3 shows the majority of errors (32%) occur during the `Data Exploration` stage. The second largest number of errors (26%) occurs during the `Data Preprocessing` stage, followed by the `Load Data` stage (19%). The error distribution in the early stages of DS tasks also highlights several design opportunities, as we will discuss later in Section 6.2.

Root Causes and Severity of Errors. To understand the root causes and severity of errors, we randomly sampled 150 error traces. The first author analyzed each error trace and then performed open coding. The last author, who was not involved in the initial coding, validated the results before writing the report. We classified root causes into four primary categories: *incorrect API usage*, *dataset unfamiliarity*, *incorrect execution order*, and *typos or syntactic oversights*. The majority of *AttributeError* instances were attributed to incorrect API usage (79%), followed by typos or syntactic oversights (14%) and execution order issues (7%). *NameError* was mostly caused by incorrect execution order (73%), while typos accounted for the remaining 27%. *KeyError* primarily resulted from dataset unfamiliarity (55%), with typos accounting for 30% and incorrect execution order for 15%. Similarly, *ValueError* was largely due to dataset unfamiliarity (95%), with typos making up 5%. *TypeError* was mainly caused by dataset unfamiliarity (85%), with incorrect API usage responsible for the remaining 15%. *IndexError* usually resulted from dataset unfamiliarity (100%), while *SyntaxError*, *ModuleNotFoundError*, *NotFoundError*, and *BadRequest* were all attributed to typos or syntactic oversights (100%).

Impact of Programming Expertise. To analyze the impact of programming expertise on error types, participants are categorized based on their programming experience into novices (less than one year, $N = 18$), intermediate programmers (one to three years, $N = 21$), and experts (more than three years, $N = 28$). Table 4 presents the error distributions. We conducted pairwise Wilcoxon Signed-rank tests on the error type distributions and found no statistically significant differences between the groups (p-values = 1.0000, 0.8124, and 0.8885, respectively). The results suggest that programming expertise has a limited effect on the overall distribution of error types. However, when comparing specific error types, we found that novice programmers made more errors such as `SyntaxError` and `BadRequest`. These errors are typically associated with syntactic oversights. In contrast, expert programmers were more prone to `TypeError` and `KeyError`. As discussed in the previous paragraph, these two types of errors were often caused by data unfamiliarity. We suspect this is because professional data scientists may not spend enough time reading the data documentation or exploring the data but rather go straight to build the DS pipeline based on their past experience and learn the dataset on the fly by tinkering [15]. For example, when accessing a column in a data frame, they may simply type down the column name based on their memory and check if it is correct. If not, they will

Table 4. Error Distribution

Error Type	Novice	Intermediate	Expert
ValueErr	23%	23%	19%
NameErr	22%	22%	17%
AttrErr	14%	11%	11%
KeyErr	3%	12%	14%
SyntaxErr	13%	6%	9%
TypeErr	3%	9%	11%
NotFndErr	5%	5%	6%
BadRequest	11%	3%	3%
IndexError	1%	3%	5%
ModNotFnd	2%	3%	2%
Misc	2%	2%	2%

quickly try another possible name or look up the data schema. This trial-and-error programming style is a common practice among experienced programmers [15, 17]. This is also evidenced by the small number of iterations expert programmers took to fix a `TypeError` or a `KeyError` in our competition (Mean 1.11 and 2.75, respectively).

Answer to RQ1.

The majority of errors are identified during early stages, such as the `Data Exploration` stage (32%), `Data Preprocessing` stage (26%), and `Load Data` stage (19%). The most frequent errors are `ValueError` (21%), `NameError` (20%), and `AttributeError` (12%). These errors mainly stem from *dataset unfamiliarity*, *incorrect API usage*, *execution order issues*, and *syntactic oversights*. However, programming experience has little impact on their overall distribution.

5.2 RQ2. Debugging Activities

Figure 5 shows that DS programmers usually make direct edits to the original erroneous cell when fixing errors. Of all error states, 35% flow to the `Edit Erroneous Cell` operation. Of these edits, 64% result in a successful fix. However, there are also other debugging strategies such as `Edit Previous Cell` (19%), `Rerun Previous Cell` (25%), and `Create New Cell` (16%). These operations involve modifying cells other than the one containing the error, suggesting that the root causes often lie elsewhere. We refer to these as *Global Fixes*. Our results indicate that a non-trivial portion of debugging efforts involve operations beyond editing the erroneous cell. This underscores the importance of understanding the broader context in which an error occurs and highlights the need for debugging tools to consider not only the error cell but also the overall code structure and flow within a notebook.

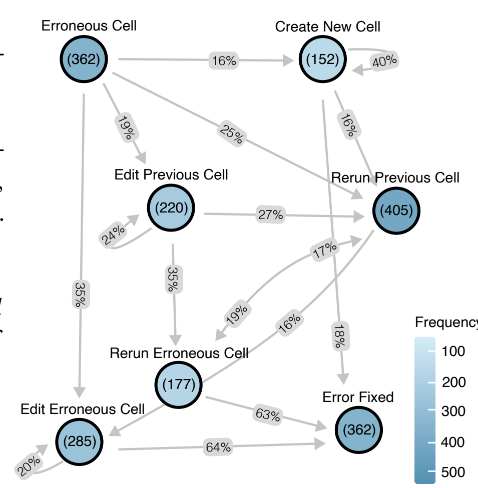


Fig. 5. State Diagram of Debugging Operations.

Additionally, some operations tend to repeat themselves, for instance, `Edit Previous Cell`. This suggests that the debugging process is often iterative. We also noticed that many operations are followed by `Rerun Previous Cell`, suggesting that running the previous cell is a common strategy for testing after editing. This suggests a need for tool support specific to this iterative debugging style. We provide more discussion in Section 6.2.

Answer to RQ2.

To fix errors, programmers not only edit the erroneous cells but also use various debugging strategies, such as editing the previous cells, rerunning the erroneous/previous cells, and creating new cells. Additionally, the results show that some debugging operations are iterative in the debugging trace.

5.3 RQ3. Fixing Patterns Used To Fix Errors

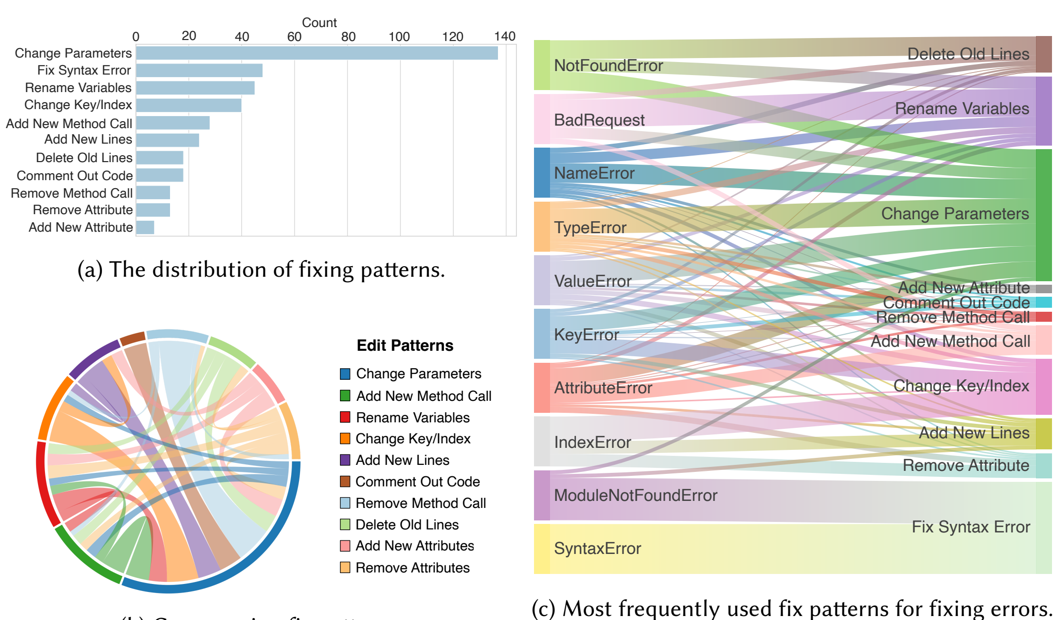
We found that debugging typically involves multiple iterations, with an average of 3.59 steps per trace. This highlights the iterative nature of DS programmers' debugging process, where several rounds of edits are often needed to resolve an error.

Moreover, Figure 6a shows the distribution of the fixing patterns. The most prevalent fixing pattern is *Change Parameters*, accounting for 35% of all fixing patterns. This is followed by *Fixing Syntax Errors* (12%), *Renaming Variables* (12%), *Changing Key/Index* (10%), and *Using New Methods* (7%). These patterns differ from the general Python programming context, where changing variable types and assignment expressions are the most common fix patterns [87]. *Changing parameters* in function calls in Jupyter notebooks implies that DS programmers often refine their computational experiments by tweaking parameters rather than restructuring code logic.

In addition, we found that these fixing patterns are not mutually exclusive since DS programmers may apply multiple patterns when fixing errors. For example, a programmer might change a parameter and then comment out code to fix a bug. The chord diagram in Figure 6b shows how different fix patterns are used together.

Our result shows that the most frequent co-occurring fix patterns are *Change Parameters* (38%), *Add New Method Call* (12%), and *Rename Variables* (12%). This suggests that debugging in data science is often complex. In terms of correlation of fix patterns and error types, as shown in Figure 6c, we found that *Change Parameters* (21%) is the most frequently used fix pattern to fix various errors. *Change Key/Index* is effective in fixing *IndexError*, while *Fix Syntax Errors* (20%) is only limited to solve simple errors such as *SyntaxError* and *ModuleNotFoundError*. The correlation between fix patterns and different error types could provide insights for future work as we will discuss in Section 6.2.

Usefulness of Error Messages in Fixing Errors. One may wonder how useful error messages are in guiding DS programmers to fix errors. To investigate this, the first author performed a



(a) The distribution of fixing patterns.

(b) Co-occurring fix patterns.

(c) Most frequently used fix patterns for fixing errors.

Fig. 6. Analysis of Edit Patterns: Distribution, Co-occurrence, and Usage for Error Fixing.

stratified sampling of 20 error messages for each error type and analyzed their usefulness. We found that error messages typically contain basic debugging information, such as the line number and a brief reason for the error. This basic debugging information is more useful for simpler errors, such as *SyntaxError* (20/20), *ModuleNotFoundError* (20/20), and *BadRequest* (20/20). However, such information is not as helpful for more complex errors, e.g., *NameError* (5/20), *AttributeError* (3/20), *ValueError* (2/20), *TypeError* (5/20), *KeyError* (6/20), *IndexError* (4/20), *AttributeError* (3/20), etc. For instance, *NameError* is often caused by incorrect execution order. The error messages provide limited information about the wrong execution order. Moreover, for errors arising from unfamiliarity with the dataset, such as *ValueError*, and *IndexError*, error messages do not offer data insights to help programmers better understand their data. For errors often caused by incorrect API usage, such as *AttributeError*, error messages usually contain lengthy tracebacks of function calls, including many third-party exception points that most programmers cannot read or modify. This may introduce additional noise into the debugging path.

Answer to RQ3.

Among all local fixes, the most common editing strategy to fix errors is *Change Parameters* (35%), followed by *Fix Syntax Errors* (12%), *Rename Variable* (12%), *Change Key/Index* (10%), and *Add New Method Call* (7%). In addition, DS programmers usually require multiple debugging iterations (3.59 steps) to fix errors. Most frequent co-occurring fix patterns are *Change Parameters* (38%), *Add New Method Call* (12%), and *Rename Variables* (12%). The most frequently used fix pattern is *Change Parameters* (20%).

5.4 RQ4. Reasons Behind DS Coding Errors and Tool Support

8 out of 10 interviewees mentioned that data preprocessing and data exploration are the most error-prone stages in the DS workflow. This result is consistent with our previous finding in RQ1 that most errors occur in the early stages of the DS workflow. We then asked follow-up questions to gain a deeper understanding of these errors. To prevent bias, we didn't share our findings with participants beforehand. Instead, we asked them generic questions about common coding errors in Jupyter notebooks, their debugging and fixing practices, and the challenges they face while debugging. Below, we use blue highlighting to indicate thematic codes that are relevant to errors occurring during the DS workflow and the programming practices of DS programmers.

Why data preprocessing is error-prone. One of the main reasons data preprocessing is error-prone is the data itself. 9 out of 10 participants reported that data is often dirty, including issues like missing values, incomplete data, and inconsistent data types. Furthermore, 8 out of 10 participants highlighted the unclear format of the data, including issues such as unclear data size, dimensions, and various file formats. As P7 said, "*These datasets can be quite dirty and noisy. One common challenge is dealing with unstructured data, which can be more complex than numerical or categorical values. Analysis becomes challenging due to the irregular formats of the data.*" 5 participants mentioned that lack of domain knowledge can pose difficulties because DS programmers need certain domain knowledge to transform the data. P5 said, "*We need to apply our business rules to check data validity. Using custom criteria from our domain experts, we can determine if the data quality is acceptable.*" Integrating different data sources could also be challenging (4/10). P9 said, "*We need to map different data, and we often make assumptions, such as using IDs in two tables as the key to join. However, this can sometimes be problematic.*" We also observed some other challenges, such as the large data size (4/10), the difficulty of automating the data cleaning process (3/10), and the complexity of transforming unstructured data into a meaningful format (2/10).

Why data exploration is error-prone. 6 out of the 10 participants said unfamiliarity with the dataset is the reason why exploring data is hard. As P9 said, "*This lack of familiarity with data can cause errors. We might know the column names and values, but without descriptions, it can be challenging to explore the data effectively.*" Also, participants mentioned the dataset might have too many columns (5/10) and it is hard to decide which features are important (4/10). P10 said, "*We have high-dimensional data, like datasets with 120 columns, which can be tricky to handle. It's challenging to decide how to visualize all the features simultaneously.*" DS Programmers sometimes have wrong assumptions about the data (4/10). P8 said, "*When I try to explore an entry of what I believe is a vector, I find out it's not, and I can't process it the way I intended. This process is error-prone and involves a lot of trial and error.*" Other challenges include inconsistent data types (3/10) and a lack of domain knowledge (3/10).

Debugging Practices. DS programmers often use "print" to debug (9/10). Some programmers search online for debugging help (6/10). Others organize their code into functions and test them (5/10). When they encounter errors, they read the error logs and trace back to the bug (3/10). They will execute cells individually to narrow down the errors (3/10) and rerun from the first cell (3/10). As P10 said, "*What I always do is separate my code into parts, dividing it into separate cells. I run the code sequentially from the first cell to the last. Whenever an error arises, I print the error messages and then go back to the cell right around the error. I execute and rerun those particular cells until I solve the problem.*" This is similar to the iterative debugging activity observed in RQ2.

Debugging Challenges in Jupyter Notebooks. 6 participants mentioned that large notebooks can create severe dependency issues, and they need to memorize cell dependencies. As P4 said, "*I do realize cell dependency is an issue because whenever you're doing a lot of preprocessing, you end up with numerous columns to memorize and preprocess.*" Among all dependency issues, complex variable dependency issues (4/10) are frequently mentioned. To solve these dependency issues, programmers usually choose to rerun from the beginning (3/10). P8 said,"*You have to rerun everything, which is time-consuming, especially when debugging and needing to change initial values to find errors.*" Rerunning behavior is also observed in RQ2, indicating that debugging in notebooks might involve managing complex dependency issues.

Improvement and Opportunity. DS programmers emphasized the need for tools to manage complex cell dependencies and track hidden variable states across multiple executions of cells to prevent unexpected behaviors. Additionally, they desired features that display the current variable value and on-hover features to display API usage and variable types. There was also a call for support for breakpoints in notebooks to facilitate more efficient debugging.

Answer to RQ4.

Data preprocessing and data exploration are the two most error-prone stages in the DS workflow. DS programmers relied on "print" statements and leveraged cell structure in the notebooks to narrow down the error. The biggest challenge in debugging is managing cell dependencies. DS programmers desire better tools to handle these dependencies and more robust debugging support within the notebook environment.

6 Discussion

6.1 Comparison to Previous Findings

We performed a systematic literature review following the guidelines by Keele et al. [36]. Specifically, we searched over 28 SE conferences (e.g., ICSE, ESEC/FSE, ASE, ISSTA, ISSRE, etc.) and 20 HCI conferences (e.g., CHI, UIST, CSCW, etc.). We used 20 search keywords, such as “Data Science”, “Programming Practice”, “Bug/Error Analysis”, and “Debugging Patterns” to identify related work. We found 88 related papers that contain at least one of the keywords in their title or abstract. After manually reviewing these papers, we identified 11 papers focused on DS debugging and programming challenges. We summarized the comparison in Table 5.

Table 5. Comparison with Previous Findings

Category	**Existing Findings**	**Our Findings in This Work**
Bug Taxonomy	[20] identified a taxonomy of 12 bug types in notebooks, including kernel bugs, conversion bugs, and portability bugs through analysis of Stack Overflow posts and GitHub bug fix commits.	• We analyzed errors using fine-grained execution histories rather than relying on coarse-grained data mined from GitHub and Stack Overflow. • We identified the distribution of 11 error types across 10 different DS stages in the DS pipeline. • Most errors occurred in the early stages of the DS pipeline, such as the data processing or data exploration stages. • *ValueError* and *NameError* are the most frequently occurring errors in the context of DS programming.
	[86] identified 3 high-level bug types, including API misuse, typos, and incorrect data modeling in the data wrangling stage.	
	[5] found 8 high-level bug types, including Coding Bug, Data Bug, and Logic Bug, in notebooks through mining Stack Overflow and bug fix commits of GitHub projects.	
Debugging Pattern	[58] identified 7 high-level error identification strategies, including using search engines, checking assumptions, and starting over to find pre-seeded Python errors in an observational user study.	• We identified 12 fine-grained fixing patterns that DS programmers frequently use to fix errors in Jupyter Notebooks. • We found *Changing parameterx* is the most frequently used fixing pattern. • Debugging activities focus not only on the erroneous cell but also on the overall code flow within notebooks. • The debugging process is highly iterative. • We analyzed the correlation between error types and fix patterns. We found that *changing parameters* fixes various errors, while *fixing syntax errors* is limited to solving simpler errors.
	[45]Identified 7 debugging patterns, including checking stack traces and version comparison to resolve cross-project correlated bugs in scientific Python projects.	
	[87] identified 29 fixing patterns, including changing assignment expressions, modifying method calls, and handling exceptions, used in Python scripting, not DS code in Jupyter Notebooks.	
	[25] identified 7 high-level debugging patterns, including iterative debugging and diff-based debugging, through a user study involving a multiverse analysis tool.	
Qualitative Studies	[17] identified 9 high-level challenges, including setup, exploration and analysis, managing code, and reliability in using notebooks, by interviewing 20 data scientists and analyzing 156 surveys.	• Compared to previous findings, our study focused more on debugging challenges in DS programming. • We found concrete reasons why data preprocessing and data exploration are more error-prone, which were not reported by previous studies. For instance, we found that debugging challenges often stem from unclear data formats and a lack of domain knowledge, which complicate the understanding and transformation of data. • We confirmed that the cell dependency issue is a severe problem in debugging Jupyter Notebooks, aligning with results in previous findings [17]. • We found that DS programmers usually execute cells individually to narrow down the errors, and sometimes rerun the notebook from the beginning.
	[21] identified 10 high-level issues in the reuse and sharing of notebooks, including modularization, versioning, and data privacy, by interviewing 17 data scientists and analyzing 132 surveys.	
	[89] identified 10 high-level challenges in collaboration among DS workers, including workflow complexity, tool fragmentation, and lack of documentation, through surveys of 183 data scientists.	
	[37] identified 10 high-level challenges in exploratory programming, including difficulties in managing exploration history and cognitive burden in tracking multiple attempts, by surveying 60 data scientists.	

6.2 Implications & Opportunities

Supporting Data-Centric Debugging in DS Programming. In RQ1, we found that most of the errors occur during the early stages of the data science pipeline, with the majority of them being identified during the `Data Exploration` stage (32%), `Data Preprocessing` stage (26%). In addition, in RQ4, 8 out of 10 interviewees confirmed that these early stages are more error-prone. They mentioned that the data is often dirty and that the data schema is usually complex. Similar issues have also been observed by Chattopadhyay et al. [17] in which data cleaning is identified as a major pain point. However, traditional debuggers, such as Python’s pdb, mainly focus on catching code errors rather than *data errors*. Although they allow inspection of variable values, they are not designed to display the contents of large datasets, such as data frames with millions of rows, or to navigate nested data structures (e.g., multi-layered JSON objects) that are common in data science. As a result, DS programmers often resort to using "print" statements to understand their data.

Several recent approaches have been proposed to support data-centric debugging. One line of work focuses on comparing data differences [42, 65, 76]. For example, Diff in the Loop (DITL) [76] automatically captures snapshots of data tables following code edits and then visualizes the differences between these snapshots to help users quickly spot unintended program behavior. In contrast to DITL, data lineage tracking [66, 68] offers a more comprehensive view of data transformations from raw input to final output by mapping the entire preprocessing workflow. For instance, a data lineage tracking feature may provide a visual map detailing each preprocessing step, such as cleaning, encoding, and normalization, and show how these steps change the data. Future work could build on these existing approaches to develop debugging features that combine visualizations of data differences with representations of the full data transformation process.

Supporting Iterative Debugging in DS Programming. In RQ2, we observed that DS programmers often edit and re-run notebook cells multiple times to fix errors. This involves exploring different code edits or parameter configurations until the error is fixed. As a result, DS programmers could benefit from features that allow them to track and compare various execution states through these iterative edits. Previously, Weinman et al. [79] proposed Fork It, which allows programmers to "fork" the execution state of a notebook, and explore different implementation alternatives—an approach reminiscent of differential testing [22, 27, 46]. However, Fork It still requires programmers to manually compare different execution paths to identify the deviations. Thus, a promising future direction is to reduce the manual effort by automatically tracking the edit history and highlighting differences in intermediate states. However, existing tools only perform differential analysis on text outputs. Supporting fine-grained differential analysis on richer types of outputs, such as data tables and plots, remains an interesting future direction.

Leveraging Repair Patterns and Data Context for Debugging in DS Programming. In RQ3, we identified various fixing patterns, including *Change Parameters* (35%), *Fix Syntax Errors* (12%), and *Rename Variable* (12%). We also identified the correlation between the fixing patterns and errors. For example, *Change Parameters* was frequently used in resolving various errors, while *Change Key/Index* was more effective in fixing *IndexError*. These insights could inform the future development of more fine-grained automated debugging tools.

One promising direction is improving Automated Program Repair (APR) in DS programming. Although recent LLM-based APR tools [82], such as AlphaRepair [83], have shown state-of-the-art performance in generating fix patches for buggy programs, they primarily focus on syntax and logic errors rather than data-centric errors. However, errors in DS programming usually stem from data errors, such as missing values and incorrect data types. To support program repair for DS programs, an APR tool should leverage the data schema and data dependencies in addition to the program dependencies and execution history. For example, future APR tools could incorporate richer data context, such as the meaning of the data being processed and the dataset characteristics.

Managing Cell Dependencies in Notebooks. In RQ4, our interview study with 10 DS programmers revealed additional insights into the challenges faced by DS programmers and potential areas for improvement. Participants highlighted the complexity of cell dependencies in large notebooks (6/10) and the need to memorize these dependencies (6/10). This finding aligns with our observations in RQ2. Some recent work has leveraged dataflow analysis to examine cell dependencies, reconstructing execution orders using a Cell Dependency Graph (CDG) to enhance notebook reproducibility [77]. However, this approach primarily focuses on execution order reconstruction. It lacks a visualization or interactive UI for programmers to navigate and understand the dependency, track variable flows, and adjust execution sequences without the need for memorization. Incorporating interactive cell dependency visualization in Jupyter Notebooks to explore and adjust execution sequences would be an interesting direction for future work.

7 Threats to Validity

Internal Validity. Our study involves some manual analysis. The manual analysis tasks in RQ1 and RQ2 include removing erroneous cells from the same origin, classifying the DS stage of each erroneous cell, and identifying the corresponding fixed cell of each erroneous cell. These tasks are simple and straightforward. Thus, we do not think the subjectivity of manual analysis would be a big threat to internal validity. However, the manual analysis in RQ3, which involves open coding and pattern summarization, is more open-ended and thus can be affected by personal experiences and biases. To mitigate this threat, two authors first independently performed the open coding and worked together to summarize the patterns. A third author who was not involved in the manual inspection process further validated the inferred patterns.

Another potential threat to internal validity is that we only focused on erroneous cells that produced compilation or runtime errors. These errors typically produce error messages and error types that are easier to detect. Other errors, such as logic errors or errors producing wrong graphs, are more difficult to identify. It requires a deeper analysis of deviations from expected behavior or user studies to understand how developers identify and resolve silent failures in practice. Finally, we down-sampled 6 notebooks per participant because the average number of submissions per participant was 5.86, which is close to 6. To ensure that the exclusion of some notebooks does not introduce bias in error type distributions, we conducted a Wilcoxon Signed-rank test to compare error type distributions between the analyzed and excluded datasets. We found that the difference is not statistically significant (p-value=0.9588). We provided analysis results in our repo at https://github.com/ferranschen/Notebook-Analysis-FSE2025/tree/main/DataAnalysis/error_type_distribution.pdf.

External Validity. One potential threat to external validity is that we only analyzed Python code written in Jupyter notebooks. We cannot guarantee that our findings are generalizable to other programming languages and programming environments in data science, such as R code in R Markdown notebooks. Another threat to validity is that we observed some task-dependent errors, such as *NotFoundError* and *BadRequestError* in our competition setting. While it is common for DS programmers to use cloud APIs to access their datasets [49], certain errors may not occur when the dataset is downloaded and analyzed locally. Another potential threat to validity is that the participants were not familiar with the dataset in the DS competition. While this mirrored the real-world scenarios where data scientists frequently work with new datasets and spend considerable time cleaning data [16, 18, 19, 26, 28, 34], we did not fully capture the error patterns that emerge when practitioners have already developed extensive experience with specific datasets. Moreover, the competition in our study focused on building a prediction model. While prediction tasks are common in data science, the error distribution identified in our study may not generalize to other tasks such as exploratory analysis and insight discovery, which emphasize more on the data exploration and visualization steps in the data science pipeline.

8 Related Work

8.1 Empirical Studies on Data Science Practices

To gain insights into DS programmers' coding practices, researchers have conducted empirical studies analyzing code mined from Kaggle or GitHub [12, 24, 29, 33, 56, 59, 73, 90]. For instance, Ramasamy et al. [56] analyzed the workflow of DS programmers by mining notebooks on GitHub. Their analysis focused on providing evidence of the iterative nature of data science. In our study, we used workflow analysis to discover the debugging activities of different debugging operations. Biswas et al. [12] studied the data science pipeline in three different settings: theory, in-the-small, and in-the-large. They identified the most representative pipelines in each setting and characterized them. In contrast, our study focused more on identifying the characteristics of the errors that DS programmers made in each data science stage rather than on identifying different pipelines. Grotov et al. [24] revealed the structural and stylistic differences between Python scripts and Jupyter Notebooks. Vidoni et al.[73] investigated self-admitted technical debt in R. Islam et al. [29] examined the executability of R Markdown files mined from GitHub. Compared to previous work, our work focused more on the programming mistakes that DS programmers make when using Jupyter Notebooks because we are particularly interested in understanding how data science programmers make errors in cell-oriented and out-of-order execution environments [67].

There are also several qualitative studies that investigate the practices of DS programmers through interviews and surveys [13, 17, 21, 37, 58, 89]. For instance, Zhang et al. [89] conducted a large-scale survey on how DS programmers work and cooperate in a large corporation. Chattopadhyay et al. [17] conducted a mixed-method study to identify pain points for DS programmers using computational notebooks. Epperson et al. [21] examined DS programmers' sharing and reuse practices, highlighting five prevalent strategies that promote or hinder reuse. Kery et al. [37] conducted interviews to investigate the exploratory programming practices of DS programmers. Robinson et al. [58] conducted an observational study on how DS programmers identify potential errors in Python Jupyter notebooks.

The most relevant research to ours has been conducted by Santana et al. [20], and Ahmed et al. [5]. They performed a bug analysis of data analytical programs mined from GitHub and Stack Overflow, mainly focusing on *the types and characteristics of bugs*. However, their results failed to identify debugging activities or bugs fixed before git commits. In contrast, our approach focuses on *how DS programmers debug and fix errors*, an aspect that has remained largely unexplored in the existing literature. We accomplish this using notebook and system logs to restore all execution histories. Additionally, we examine bug distribution across data science stages. Finally, our fine-grained dataset enables us to provide error-fixing efforts for each error using GumTree [23].

8.2 Tool Support for Data Science

The research community has developed various tools to enhance different aspects of the data science workflow. Vizsmith [8] enables code reuse for visualizations by mining visualization code from Kaggle notebooks. Vu et al. [74] introduced a semi-automated method for reducing input data in workflows while maintaining specified outcomes. CombyInferPy [48] automatically analyzes new changes in data science library APIs, facilitating the update of large projects to newer library versions. DSInfoSearch [64] supports the experimentation process by providing context-aware ranked data science experiments. Yang et al. [86] created WRANGLEDOC using program synthesis to help DS programmers generate documentation for their data-wrangling code. Later, Yang et al. [85] developed a static analysis approach to detect common forms of data leakage in data science code. Safe-DS [57] offers a domain-specific language for data science that can catches common type errors. Subotić et al. [67] proposed a framework for static analysis of notebooks. SOAR [50] introduces a synthesis approach for data science API refactoring that requires no training data.

Despite the significant advancements in the research community, a notable scarcity of tools to help DS programmers debug remains. As observed by Chattopadhyay et al.[17] in their user study, DS programmers frequently encounter challenges tracing code flow due to the out-of-order execution properties inherent in notebooks [67]. This can lead to dependency issues and cause errors propagating through subsequent cells, ultimately leading to "dependency hell" in the notebooks.

To improve debugging support for DS programmers, it is essential first to understand their debugging activities. However, previous studies have only examined code from GitHub or Kaggle, which misses crucial details on how DS programmers test and edit cells in situ. Git commits only provide a static snapshot of users' behavior rather than their real-time behavior. In contrast, our study is the first to address this by creating a fine-grained dataset to identify common errors across data science stages, analyze editing patterns, and model debugging traces of data science code.

8.3 Error Analysis of Other Kinds of Programs

There are also growing interests of error analysis covering various kinds of applications, such as machine learning [5, 30, 69, 90], mobile applications [6, 11, 91], web applications [51, 52, 62], operating systems [4, 31, 44], and blockchain-based systems [75]. Furthermore, there are several curated datasets of real-world software bugs. For example, Defects4J [32] contains 395 Java bugs, Bugs.jar [61] includes 1,158 Java bugs along with their fixes, ManyBugs dataset [43] holds 185 C language bugs, and BugsInPy [80] documents 493 bugs in Python programs. Recently, there have been larger datasets such as ManySStuBs4J [35], which consist of 153,652 bugs.

Despite these efforts, there is a notable lack of research focusing on the errors made by DS programmers using Jupyter Notebook. Our research aims to bridge this gap by analyzing fine-grained debugging activities of DS programmers within the Jupyter Notebook [39], which differs from traditional programming IDEs. Furthermore, prior error analyses have focused on identifying bug types [20]. By contrast, we explore various facets, such as common errors in different stages, editing patterns, and debugging activities when using Jupyter Notebook.

9 Conclusion

In this study, we investigated the fine-grained debugging practices of DS programmers. We examined the internal logs of each notebook from a six-week DS competition. These logs contained a total of 390 Jupyter Notebooks, authored by 67 participants over six weeks. This rich data, covering all code changes, cell execution order, and output logs, provided us with an in-depth view of data science programming practices by identifying different facets of debugging practices, including common errors across different data science stages, editing patterns, and debugging activities. In addition, we conducted semi-structured interviews with 10 DS programmers from both industry and academia to understand the reasons behind these coding errors. Our study is the first to investigate the fine-grained debugging activities of DS programming within Jupyter Notebook. It provides implications for developing more effective data science support tools, offering a more comprehensive understanding of the debugging practices of DS programmers.

10 Data Availability

The code and data have been made publicly available at https://github.com/ferranschen/Notebook-Analysis-FSE2025.

11 Acknowledgments

We thank all the participants in our data science competition for their valuable contributions. We also appreciate the valuable feedback provided by the anonymous reviewers.

References

[1] 2019. The Data Scientist Profile 2019 - Skills, Experience, Education Of 1,001 Data Scientists. https://365datascience.com/career-advice/career-guides/data-scientist-profile/.

[2] 2024. IPython. https://ipython.readthedocs.io/.

[3] 2024. Jupyter Notebook. https://jupyter.org/.

[4] Iago Abal, Claus Brabrand, and Andrzej Wasowski. 2014. 42 variability bugs in the linux kernel: a qualitative analysis. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 421–432.

[5] Shibbir Ahmed, Mohammad Wardat, Hamid Bagheri, Breno Dantas Cruz, and Hridesh Rajan. 2023. Characterizing Bugs in Python and R Data Analytics Programs. *arXiv preprint arXiv:2306.08632* (2023).

[6] Tamjid Al Rahat, Yu Feng, and Yuan Tian. 2019. Oauthlint: An empirical study on oauth bugs in android applications. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 293–304.

[7] Abdulaziz Alaboudi and Thomas D LaToza. 2023. What constitutes debugging? An exploratory study of debugging episodes. *Empirical Software Engineering* 28, 5 (2023), 117.

[8] Rohan Bavishi, Shadaj Laddad, Hiroaki Yoshida, Mukul R Prasad, and Koushik Sen. 2021. Vizsmith: Automated visualization synthesis by mining data-science notebooks. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 129–141.

[9] Moritz Beller, Niels Spruit, Diomidis Spinellis, and Andy Zaidman. 2018. On the dichotomy of debugging behavior among programmers. In *Proceedings of the 40th International Conference on Software Engineering*. 572–583.

[10] Bruce Lawrence Berg. 2001. *Qualitative research methods for the social sciences.* Allyn & Bacon.

[11] Pamela Bhattacharya, Liudmila Ulanova, Iulian Neamtiu, and Sai Charan Koduru. 2013. An empirical analysis of bug reports and bug fixing in open source android apps. In *2013 17th European Conference on Software Maintenance and Reengineering*. IEEE, 133–143.

[12] Sumon Biswas, Mohammad Wardat, and Hridesh Rajan. 2022. The art and practice of data science pipelines: A comprehensive study of data science pipelines in theory, in-the-small, and in-the-large. In *Proceedings of the 44th International Conference on Software Engineering*. 2091–2103.

[13] Kelly Nicole Bodwin, Ian Flores Siaca, Amelia McNamara, Philipp Burckhardt, Allison Theobold, Amal Abdel-Ghani, and Greg Wilson. 2022. "Looks okay to me": A study of best practice in data analysis code review. *ICOTS* (2022).

[14] Virginia Braun and Victoria Clarke. 2006. Using thematic analysis in psychology. *Qualitative research in psychology* 3, 2 (2006), 77–101.

[15] Carrie J Cai and Philip J Guo. 2019. Software developers learning machine learning: Motivations, hurdles, and desires. In *2019 IEEE symposium on visual languages and human-centric computing (VL/HCC)*. IEEE, 25–34.

[16] Steven P Callahan, Juliana Freire, Emanuele Santos, Carlos E Scheidegger, Cláudio T Silva, and Huy T Vo. 2006. VisTrails: visualization meets data management. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. 745–747.

[17] 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*. 1–12.

[18] Bhavya Chopra, Anna Fariha, Sumit Gulwani, Austin Z Henley, Daniel Perelman, Mohammad Raza, Sherry Shi, Danny Simmons, and Ashish Tiwari. 2023. CoWrangler: Recommender System for Data-Wrangling Scripts. In *Companion of the 2023 International Conference on Management of Data*. 147–150.

[19] Tamraparni Dasu and Theodore Johnson. 2003. *Exploratory data mining and data cleaning.* John Wiley & Sons.

[20] Taijara Loiola de Santana, Paulo Anselmo da Mota Silveira Neto, Eduardo Santana de Almeida, and Iftekhar Ahmed. 2022. Bug Analysis in Jupyter Notebook Projects: An Empirical Study. *ACM Transactions on Software Engineering and Methodology* (2022).

[21] Will Epperson, April Yi Wang, Robert DeLine, and Steven M Drucker. 2022. Strategies for reuse and sharing among data scientists in software teams. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*. 243–252.

[22] Robert B Evans and Alberto Savoia. 2007. Differential testing: a new approach to change detection. In *The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers*. 549–552.

[23] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 313–324.

[24] Konstantin Grotov, Sergey Titov, Vladimir Sotnikov, Yaroslav Golubev, and Timofey Bryksin. 2022. A large-scale comparison of Python code in Jupyter notebooks and scripts. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 353–364.

[25] Ken Gu, Eunice Jun, and Tim Althoff. 2023. Understanding and supporting debugging workflows in multiverse analysis. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–19.

[26] Sumit Gulwani. 2016. Programming by examples-and its applications in data wrangling. In *Dependable Software Systems Engineering*. IOS Press, 137–158.

[27] Muhammad Ali Gulzar, Yongkang Zhu, and Xiaofeng Han. 2019. Perception and practices of differential testing. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 71–80.

[28] Junjie Huang, Daya Guo, Chenglong Wang, Jiazhen Gu, Shuai Lu, Jeevana Priya Inala, Cong Yan, Jianfeng Gao, Nan Duan, and Michael R Lyu. 2024. Contextualized Data-Wrangling Code Generation in Computational Notebooks. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1282–1294.

[29] Md Anaytul Islam, Muhammad Asaduzzman, and Shaowei Wang. 2024. On the Executability of R Markdown Files. In *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*. IEEE, 254–264.

[30] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A comprehensive study on deep learning bug characteristics. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 510–520.

[31] Matthieu Jimenez, Mike Papadakis, and Yves Le Traon. 2016. An empirical analysis of vulnerabilities in openssl and the linux kernel. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 105–112.

[32] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*. 437–440.

[33] Malin Källén, Ulf Sigvardsson, and Tobias Wrigstad. 2021. Jupyter notebooks on github: characteristics and code clones. *The Art, Science, and Engineering of Programming* 5, 3 (2021).

[34] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. 2011. Wrangler: Interactive visual specification of data transformation scripts. In *Proceedings of the sigchi conference on human factors in computing systems*. 3363–3372.

[35] Rafael-Michael Karampatsis and Charles Sutton. 2020. How often do single-statement bugs occur? the manysstubs4j dataset. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 573–577.

[36] Staffs Keele et al. 2007. *Guidelines for performing systematic literature reviews in software engineering*. Technical Report. Technical report, ver. 2.3 ebse technical report. ebse.

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

[38] Miryung Kim, Thomas Zimmermann, Robert DeLine, and Andrew Begel. 2016. The emerging role of data scientists on software development teams. In *Proceedings of the 38th International Conference on Software Engineering*. 96–107.

[39] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B Hamrick, Jason Grout, Sylvain Corlay, et al. 2016. Jupyter Notebooks-a publishing format for reproducible computational workflows. *Elpub* 2016 (2016), 87–90.

[40] Donald Ervin Knuth. 1984. Literate programming. *The computer journal* 27, 2 (1984), 97–111.

[41] Zoe Kotti, Georgios Gousios, and Diomidis Spinellis. 2022. Impact of software engineering research in practice: A patent and author survey analysis. *IEEE Transactions on Software Engineering* 49, 4 (2022), 2020–2038.

[42] Po-Ming Law, Rahul C Basole, and Yanhong Wu. 2018. Duet: Helping data analysis novices conduct pairwise comparisons by minimal specification. *IEEE transactions on visualization and computer graphics* 25, 1 (2018), 427–437.

[43] Claire Le Goues, Neal Holtschulte, Edward K Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The ManyBugs and IntroClass benchmarks for automated repair of C programs. *IEEE Transactions on Software Engineering* 41, 12 (2015), 1236–1256.

[44] Zhenpeng Lin, Yueqi Chen, Yuhang Wu, Dongliang Mu, Chensheng Yu, Xinyu Xing, and Kang Li. 2022. GREBE: Unveiling exploitation potential for Linux kernel bugs. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2078–2095.

[45] Wanwangying Ma, Lin Chen, Xiangyu Zhang, Yuming Zhou, and Baowen Xu. 2017. How do developers fix cross-project correlated bugs? a case study on the github scientific python ecosystem. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 381–392.

[46] William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[47] Wes McKinney et al. 2011. pandas: a foundational Python library for data analysis and statistics. *Python for high performance and scientific computing* 14, 9 (2011), 1–9.

[48] Hailie Mitchell. 2022. Automatically Fixing Breaking Changes of Data Science Libraries. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–3.

[49] Paul Timothy Mooney. 2022. Kaggle Survey 2022: All Results. https://www.kaggle.com/code/paultimothymooney/kaggle-survey-2022-all-results Accessed: 2025.

[50] Ansong Ni, Daniel Ramos, Aidan ZH Yang, Inês Lynce, Vasco Manquinho, Ruben Martins, and Claire Le Goues. 2021. Soar: a synthesis approach for data science api refactoring. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 112–124.

[51] Frolin Ocariza, Kartik Bajaj, Karthik Pattabiraman, and Ali Mesbah. 2013. An empirical study of client-side JavaScript bugs. In *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 55–64.

[52] Frolin S Ocariza Jr, Karthik Pattabiraman, and Benjamin Zorn. 2011. JavaScript errors in the wild: An empirical study. In *2011 IEEE 22nd International Symposium on Software Reliability Engineering*. IEEE, 100–109.

[53] Wonseok Oh and Hakjoo Oh. 2022. PyTER: effective program repair for Python type errors. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 922–934.

[54] Yun Peng, Shuzheng Gao, Cuiyun Gao, Yintong Huo, and Michael Lyu. 2024. Domain knowledge matters: Improving prompts with fix templates for repairing python type errors. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.

[55] Deepthi Raghunandan, Aayushi Roy, Shenzhi Shi, Niklas Elmqvist, and Leilani Battle. 2022. Code code evolution: Understanding how people change data science notebooks over time. *arXiv preprint arXiv:2209.02851* (2022).

[56] Dhivyabharathi Ramasamy, Cristina Sarasua, Alberto Bacchelli, and Abraham Bernstein. 2023. Workflow analysis of data science code in public GitHub repositories. *Empirical Software Engineering* 28, 1 (2023), 1–47.

[57] Lars Reimann and Günter Kniesel-Wünsche. 2023. Safe-DS: A Domain Specific Language to Make Data Science Safe. In *2023 IEEE/ACM 45th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE, 72–77.

[58] Derek Robinson, Neil A Ernst, Enrique Larios Vargas, and Margaret-Anne D Storey. 2022. Error identification strategies for Python Jupyter notebooks. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*. 253–263.

[59] 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.

[60] Per Runeson and Martin Höst. 2009. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering* 14 (2009), 131–164.

[61] Ripon K Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul R Prasad. 2018. Bugs. jar: A large-scale, diverse dataset of real-world java bugs. In *Proceedings of the 15th international conference on mining software repositories*. 10–13.

[62] Marija Selakovic and Michael Pradel. 2016. Performance issues and optimizations in javascript: an empirical study. In *Proceedings of the 38th International Conference on Software Engineering*. 61–72.

[63] Forrest Shull, Janice Singer, and Dag IK Sjøberg. 2008. *Guide to advanced empirical software engineering*. Vol. 93. Springer.

[64] Shangeetha Sivasothy. 2021. DSInfoSearch: supporting experimentation process of data scientists. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1033–1037.

[65] Arjun Srinivasan, Matthew Brehmer, Bongshin Lee, and Steven M Drucker. 2018. What's the difference? evaluating variations of multi-series bar charts for visual comparison tasks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. 1–12.

[66] Holger Stitz, Samuel Gratzl, Harald Piringer, Thomas Zichner, and Marc Streit. 2018. Knowledgepearls: Provenance-based visualization retrieval. *IEEE Transactions on Visualization and Computer Graphics* 25, 1 (2018), 120–130.

[67] Pavle Subotić, Lazar Milikić, and Milan Stojić. 2022. A static analysis framework for data science notebooks. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*. 13–22.

[68] MingJie Tang, Saisai Shao, Weiqing Yang, Yanbo Liang, Yongyang Yu, Bikas Saha, and Dongjoon Hyun. 2019. Sac: A system for big data lineage tracking. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1964–1967.

[69] Ferdian Thung, Shaowei Wang, David Lo, and Lingxiao Jiang. 2012. An empirical study of bugs in machine learning systems. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering*. IEEE, 271–280.

[70] John Wilder Tukey et al. 1977. *Exploratory data analysis*. Vol. 2. Springer.

[71] Wil Van Der Aalst and Wil van der Aalst. 2016. *Data science in action*. Springer.

[72] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. 2011. The NumPy array: a structure for efficient numerical computation. *Computing in science & engineering* 13, 2 (2011), 22–30.

[73] Melina Vidoni. 2021. Self-admitted technical debt in r packages: An exploratory study. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 179–189.

[74] Anh Duc Vu, Timo Kehrer, and Christos Tsigkanos. 2022. Outcome-preserving input reduction for scientific data analysis workflows. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–5.

[75] Zhiyuan Wan, David Lo, Xin Xia, and Liang Cai. 2017. Bug characteristics in blockchain systems: a large-scale empirical study. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 413–424.

[76] April Yi Wang, Will Epperson, Robert A DeLine, and Steven M Drucker. 2022. Diff in the loop: Supporting data comparison in exploratory data analysis. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. 1–10.

[77] Jiawei Wang, Tzu-yang Kuo, Li Li, and Andreas Zeller. 2020. Assessing and restoring reproducibility of Jupyter notebooks. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 138–149.
[78] 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*. 53–56.
[79] 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* (Yokohama, Japan) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA, Article 307, 12 pages. doi:10.1145/3411764.3445527
[80] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, et al. 2020. BugsInPy: A database of existing bugs in Python programs to enable controlled testing and debugging studies. In *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 1556–1560.
[81] Michael Williams and Tami Moser. 2019. The art of coding and thematic exploration in qualitative research. *International management review* 15, 1 (2019), 45–55.
[82] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1482–1494.
[83] Chunqiu Steven Xia and Lingming Zhang. 2022. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 959–971.
[84] Yihui Xie, Joseph J Allaire, and Garrett Grolemund. 2018. *R markdown: The definitive guide*. CRC Press.
[85] Chenyang Yang, Rachel A Brower-Sinning, Grace Lewis, and Christian Kästner. 2022. Data leakage in notebooks: Static detection and better processes. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.
[86] Chenyang Yang, Shurui Zhou, Jin LC Guo, and Christian Kästner. 2021. Subtle bugs everywhere: Generating documentation for data wrangling code. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 304–316.
[87] Yilin Yang, Tianxing He, Yang Feng, Shaoying Liu, and Baowen Xu. 2022. Mining Python fix patterns via analyzing fine-grained source code changes. *Empirical Software Engineering* 27, 2 (2022), 48.
[88] Carmen Zannier, Grigori Melnik, and Frank Maurer. 2006. On the success of empirical studies in the international conference on software engineering. In *Proceedings of the 28th international conference on Software engineering*. 341–350.
[89] Amy X Zhang, Michael Muller, and Dakuo Wang. 2020. How do data science workers collaborate? roles, workflows, and tools. *Proceedings of the ACM on Human-Computer Interaction* 4, CSCW1 (2020), 1–23.
[90] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An empirical study on TensorFlow program bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 129–140.
[91] Bo Zhou, Iulian Neamtiu, and Rajiv Gupta. 2015. A cross-platform analysis of bugs and bug-fixing in open source projects: Desktop vs. android vs. ios. In *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*. 1–10.

Received 2025-02-26; accepted 2025-04-01