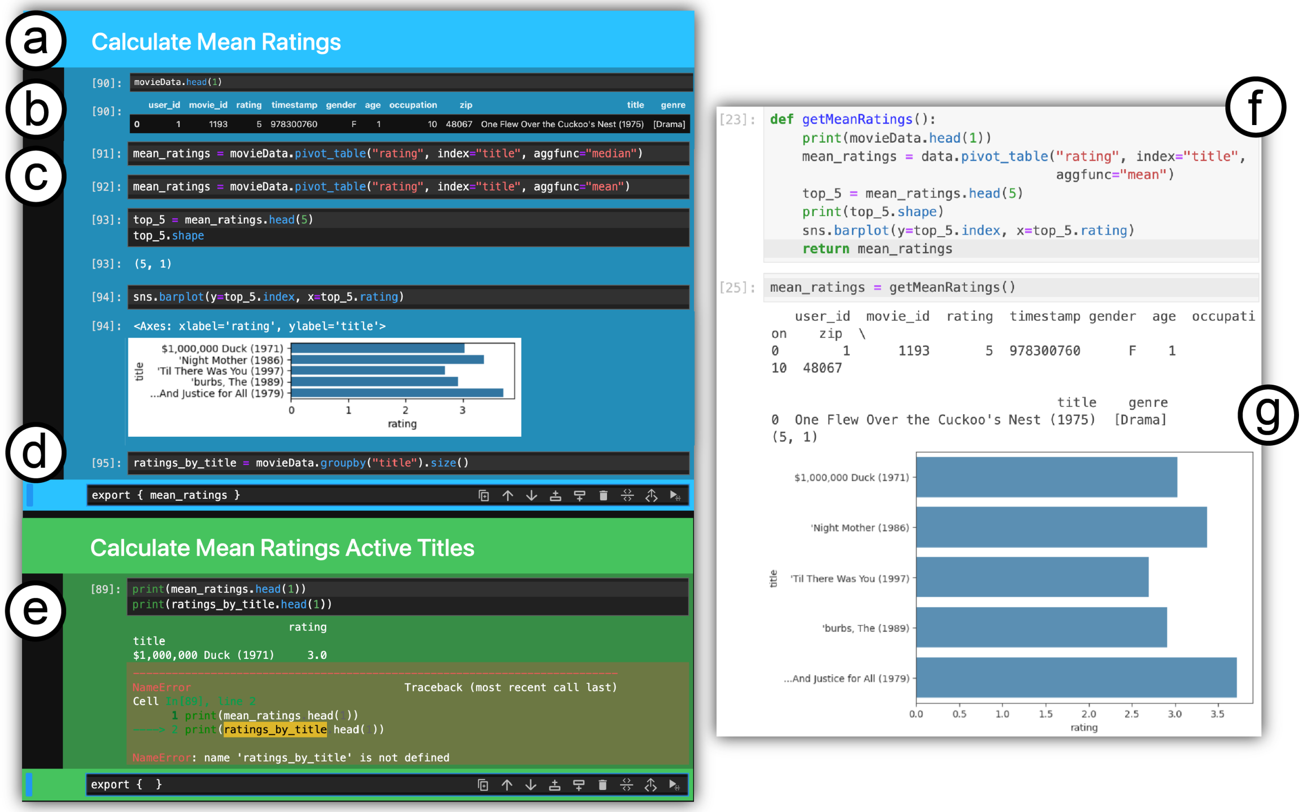
Pagebreaks: Multi-Cell Scopes in Computational Notebooks

Eric Rawn
erawn@berkeley.edu
University of California, Berkeley
Berkeley, CA

S. E. Chasins
schasins@berkeley.edu
University of California, Berkeley
Berkeley, CA



```
[90]: movieData.head(1)

[90]:    user_id  movie_id  rating  timestamp  gender  age  occupation    zip                                  title    genre
      0        1      1193       5  978300760       F    1          10  48067  One Flew Over the Cuckoo's Nest (1975)  [Drama]

[91]: mean_ratings = movieData.pivot_table("rating", index="title", aggfunc="median")

[92]: mean_ratings = movieData.pivot_table("rating", index="title", aggfunc="mean")

[93]: top_5 = mean_ratings.head(5)
      top_5.shape

[93]: (5, 1)

[94]: sns.barplot(y=top_5.index, x=top_5.rating)

[94]: <Axes: xlabel='rating', ylabel='title'>

[95]: ratings_by_title = movieData.groupby("title").size()

export { mean_ratings }
```

```
[89]: print(mean_ratings.head(1))
      print(ratings_by_title.head(1))

                             rating
      title
      $1,000,000 Duck (1971)     3.0
      ---------------------------------------------------------------------------
      NameError                                 Traceback (most recent call last)
      Cell In[89], line 2
            1 print(mean_ratings.head(1))
      ----> 2 print(ratings_by_title.head(1))

      NameError: name 'ratings_by_title' is not defined

export {  }
```

```
[23]: def getMeanRatings():
          print(movieData.head(1))
          mean_ratings = data.pivot_table("rating", index="title",
                                          aggfunc="mean")
          top_5 = mean_ratings.head(5)
          print(top_5.shape)
          sns.barplot(y=top_5.index, x=top_5.rating)
          return mean_ratings

[25]: mean_ratings = getMeanRatings()

         user_id  movie_id  rating  timestamp gender  age  occupati
      on    zip  \
      0        1      1193       5  978300760      F    1
      10  48067

                                         title    genre
      0  One Flew Over the Cuckoo's Nest (1975)  [Drama]
      (5, 1)
```

Figure 1: A *Pagebreak* is a language construct for adding a scope around multiple Jupyter Notebook cells. Left: A notebook using Pagebreaks. A Variables defined within the indented cells can only be used within a defined portion of the notebook. Unlike defining the same variables inside a function, the Pagebreak does not interfere with key *exploratory interactions*, such as B: interleaving code and its output, C: using cells as impromptu versions, and D: iteratively composing programs with single-line cells. E: Only variables exported at the bottom of a Pagebreak are available for other cells. For example `mean_ratings` is available in the bottom Pagebreak, but `ratings_by_title` is not. Right: In contrast, functions provide scopes at the cost of these exploratory interactions. F: Functions prevent iterative composition or using cells as impromptu versions inside a function body. G: Functions prevent interleaving code and output; instead all printed output from a function appears together at the call site, not interleaved with the function body. Example adapted from [30].

ABSTRACT

Global variables lie at the root of many programmer complaints about computational notebooks. While programmers in other environments often address these barriers with function scopes, notebook programmers use functions less often. Analyzing the interaction between user behaviors, the programming language, and the notebook environment, we propose one possible explanation: that functions interfere with using notebooks in the exploratory ways users value. For example, because partial functions are not parseable,

CHI '25, April 26-May 1, 2025, Yokohama, Japan
© 2025 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-XXXX-X/18/06.
https://doi.org/10.1145/3706598.3713620

they cannot be run in isolation, so programmers cannot split function bodies across cells to iteratively tweak and rerun the last few lines. To explore how to offer non-global scopes without hampering exploratory notebook interactions, we built Pagebreaks, a small language construct for adding scopes around multiple Jupyter Notebook cells. In an in-situ study, we explored how programmers used Pagebreaks to manage variables with non-global scopes but also to visually and conceptually organize programs in a way akin to functions.

CCS CONCEPTS

• **Human-centered computing** → **Interactive systems and tools**; • **Software and its engineering** → *Development frameworks and environments.*

KEYWORDS

Computational Notebooks; Exploratory Programming; Data Science; Scope

ACM Reference Format:
Eric Rawn and S. E. Chasins. 2025. Pagebreaks: Multi-Cell Scopes in Computational Notebooks. In *CHI Conference on Human Factors in Computing Systems (CHI '25), April 26-May 1, 2025, Yokohama, Japan.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3706598.3713620

1 INTRODUCTION

Computational notebook users often experience barriers and confusions related to the pervasive use of global variables [7, 9, 43, 49, 52, 57]. For example, they must often (i) trace global variables throughout a large notebook [13, 23, 52], (ii) avoid accidentally overwriting commonly used names (e.g. `df` or `model`) [13, 57], and (iii) when a notebook contains multiple definitions in different cells, they must remember which cell they have run more recently [36, 44].

In non-notebook programming environments, programmers often use functions to reduce the number of global variables they need to track. Functions accomplish this by introducing *non-global scopes.* In lexically scoped languages [1], like almost all modern programming languages, a name defined in a non-global scope may only be used in a defined segment of the source file (e.g. between lines 22 and 29). For instance, if the name `x` is defined inside function `foo`, and the body of `foo`'s function definition appears in lines 130–145 of the source file, `x` may only appear in lines 130—145. Outside of notebook programming, non-global scopes—constraining where names can be used—are a popular approach for reducing the confusion associated with global variables [14]. According to previous qualitative findings, however, functions are sometimes used sparsely in notebook programs [10, 20].

We explore why notebook programmers might be reluctant to use functions: we find that functions impede the *exploratory* interactions many users value about notebooks. Kery identifies three core exploratory behaviors: (1) using cells as impromptu alternatives, (2) interleaving code and its output, and (3) composing a program by iteratively editing and running a one-line cell representing the next line to be added [20, 45]. Importantly, all of these behaviors rely on multiple cells having access to the same variables. We bring together these and other qualitative findings about user behaviors with an analysis of both the programming language (Python) and the programming environment (Jupyter [11]/IPython [32]). Looking at Python and Jupyter, we know that function definitions cannot be split across multiple cells—they must appear entirely within a single cell. This means that variables defined inside a function are accessible only inside a single cell. Now we can observe that exploratory behaviors and functions are in tension: For the exploratory behaviors above, multiple cells read and write the same variables, but variables defined in functions can only be read and written in a single cell.

This work introduces a construct for getting a key benefit of functions—non-global scopes—without impeding exploratory interactions. We present the *Pagebreak*, a language construct implemented for Jupyter Notebooks which allows users to create non-global scopes around *multiple* notebook cells. We used both of the analyses described above—the analysis of how functions interfere with notebook programming and the analysis of how global variables confuse notebook programmers—to frame design requirements for our construct. We hypothesize that non-global scopes may be able to address notebook programmers' issues if they could be separated from the aspects of functions that hamper exploratory behaviors. We implement Pagebreaks in a pair of Jupyter and IPython plugins which together allow users to create scopes around groups of cells via the notebook UI.

We conducted an in-situ [42] user study with five participants, asking them to use Pagebreaks in their own work for two weeks. We conducted a thematic analysis [4] of the 1–1.5 hour interviews we conducted with each participant about their experience. Our analysis focused in particular on what sessions revealed about the role of scoping constructs—including Pagebreaks and functions—in participants' day-to-day notebook programming. Participants reported that Pagebreaks helped them address many of the issues we previously identified, and did not seem to impede exploratory behaviors. Besides these questions, the themes developed from our analysis show the range of organizational roles Pagebreaks played for participants. We also observed that participants used the word "function" to refer to a great variety of different program components — sometimes referring to one cell, to some of the code within one cell, or sometimes to groups of cells. Notably, this included code which *did not have function definitions*, including Pagebreaks. We discuss how this connects to the roles that Pagebreaks and scopes play in their programming process.

We present the following contributions:

(1) The design of a language construct, the *Pagebreak*, and its implementation for Jupyter Notebooks. A Pagebreak introduces a scope around one or more cells.
(2) An in-situ qualitative user study in which we interviewed participants after they used Pagebreaks in their own notebooks over the course of two weeks. We present findings on the role of Pagebreaks, but also scopes more broadly, including function scopes.

2 RELATED WORK

2.1 How Notebook Programmers Program

Notebook programmers commonly write code with highly exploratory goals [2]: to investigate an idea, experiment with an approach, or test a potential solution. In notebook programming, Kery observed exploratory "expand-reduce" behavior: iteratively developing small (1-2 line) cells and organizing them into larger groups later [23]. Programmers used cells to represent different "logical units" of work, where cells that were alternatives of each other or accomplished related work were grouped into different "thematic regions" of the notebook [23]. Splitting cells into smaller units was also identified as a “fairly instantaneous and straightforward” debugging strategy, allowing code to be split "into different cells and then slowly stitching them together until you get something that works right” [7], echoing Kery's notion of "expand-reduce" behavior (also discussed by Robinson et al. in their study of notebook debugging strategies [41]). Cells also sometimes serve as improvisational versions when programmers selectively run cells that serve as alternatives [13, 23].

Because exploratory programmers often do not prioritize modularity or reusability (at least not in the moment), the "investment" required to organize exploratory code into functions was often found too onerous [20, 31]. "High-Quality Code" practices such as organizing code into functions, or moving code into separate Python scripts and modules were described as "counter-productive" for exploratory uses of notebooks [36]. Wrapping existing code into functions was described by Dong et al. [10] as a "cleaning activity", done at the end of the programming task, which only occurred within notebooks in their sample intended for use in a larger pipeline or workflow.

2.1.1 Quantitative Research On Notebook Programming. Quantitative research has focused primarily on large-scale analyses of corpora like Github, analyzing the structure and development of notebooks as they're put into version control. This has included analyses to restore appropriate dependencies to Jupyter Notebooks [55], understanding sensemaking behaviors [37], cleaning behaviors [10], and characterizing notebook code into distinct steps of a workflow [38]. Other work has focused on studying code duplication in notebooks [25, 27]. These studies reveal important aspects of working with Jupyter Notebooks. Our qualitative study, rather than focusing on specific steps like documenting, cleaning, data preparation, notebook organization, and visualization, instead focuses on characterizing participants' interactions with scopes, especially with functions and Pagebreaks.

2.2 Obstacles in Notebook Programming

Previous work has identified many issues notebook programmers face. Because cells can be manually executed one-at-a-time, running cells out of the programmer's intended order can cause unexpected results [7, 9, 43, 49, 52, 57]. For instance: accidentally omitting a cell from execution can cause runtime errors [36], cells may be organized in an order other than the intended execution order, causing problems when users attempt to run the entire notebook linearly [44], and non-idempotent operations can create notebook states which are difficult to reason about and reproduce [49]. Because cells read from and write to a shared global state, defining the same variable in multiple cells can make it difficult to trace the code that last defined it—i.e. the value that other cells actually accessed [52]. This leads some programmers to continually rename modified dataframes and other repeatedly modified objects with similar names (e.g. "`df_1` and `df_2`" [57]). This addresses one confusion but often introduces new ones [13, 57].

While using cells as alternatives or iterations might be an effective exploratory strategy, previous work finds that programmers often struggle to manage the complexity of these notebooks once they get sufficiently large [13, 23]. Programmers will often move code into separate notebooks to manage notebook length, starting explorations anew by copying over just the relevant code. Alternatively, they may engage in "cleaning" [10, 13, 23] — deleting cells which were made to debug or explore an alternative and combining cells together into larger units of work — which often requires significant transformation of their programs and the loss of their exploration history.

2.3 Research Systems in Computational Notebooks

Previous systems research on computational notebooks has primarily focused on either notebook organization or adding additional context into the programming environment.

Previous work has explored visually organizing notebooks through cell folding interactions [44], annotation and hierarchical organization [6], searching notebooks with natural language and visualization [28], visualizing notebooks [51, 58], two-dimensional notebook layouts [12], and notebook cleaning through slicing techniques [13]. The Pagebreaks implementation uses cell folding and labeling ideas from Rule et al. [44] (now a part of Jupyter by default), but this is not central to Pagebreaks. Where Head et al. [13] focus on stripping extraneous cells from a "messy" notebook for sharing in the post-exploration phase, Pagebreaks instead focuses on supporting notebook programmers in avoiding issues and confusions throughout their development.

Previous work has also investigated ways to show various kinds of contextual information in notebook interfaces with margin notes [19], collaborative discussions [54], and automatic capture of contextual interactions [33].

Other research has focused on tools for recording, organizing, and managing notebook version history [21, 22]; bringing direct manipulation interactions into notebook environments through code generation [24]; interactive visualization construction [59]; and embedding notebooks inside other notebooks [15].

A large amount of work on notebooks has focused on reproduceability [3, 8, 34, 35, 39, 44, 55], documenting and sharing notebooks with others [18, 53, 56], and creating notebooks with unambiguous run orders [40]. These concerns often occur after programmers complete exploratory work; as Rule et al. [45] argue, exploratory behaviors are in direct tension with using notebooks to share and explain. At this post-exploration stage, programmers create a notebook to showcase results, give a reproducible analysis to a collaborator, or incorporate code into a production pipeline. While these concerns are important, our contribution here is focused on supporting individual notebook users in their exploratory programming.

Each of these projects explores an important area of computational notebooks. While we share a common focus on studying and supporting notebook users, the current project focuses on how to support notebook programmers in organizing their notebook *state* using scopes.

2.3.1 State Management in Computational Notebooks. We are aware of two prior research systems aimed at managing computational notebook state, although neither explore the possibilities of programmer-defined scopes. Weiman et al. [57] serializes the entire global state after each cell run. This allows users to inspect the state before and after a cell run. By feeding a serialized state into a fresh IPython session, programmers can also "fork" state, then compare the states side-by-side in the same Jupyter interface. In Dataflow notebooks [26] users explicitly annotate how they want variables to flow between cells, building up a directed acyclic graph via using cell identifiers with variables names. Cells may not use a variable unless the programmer has added an edge between the source node (the cell that defines the variable) and the sink node (the cell that uses the variable) by using the source node's identifier within the sink node. Within a single cell, variables may be shared freely. Although this approach certainly does not prevent multiple cells from using the same variables, it does mean that variables are only shared by default within a single cell. The Dataflow intervention was aimed at replication [26], rather than exploration, so we should not expect a good fit with our own goals of supporting exploratory interactions. Indeed, our analysis revealed the importance of freely sharing the same variables across *multiple* cells during exploration, so we did not consider an explicit per-variable dataflow annotation approach for our purposes. Note that no prior work has proposed a system that shares variables freely across multiple cells—other than the standard approach of using global variables. Still, at a high level these works share the goal of helping notebook programmers handle the messiness of notebook state, at some stage in the notebook programming process.

A plugin for IPython and Jupyter has explored an approach for splitting global state [47]. While this approach may be helpful for some users, it still allows cells anywhere in the notebook to modify and define any other variable, a key part of what lexical scopes are designed to prevent.

2.3.2 Notebook Analysis Techniques. Techniques for analyzing Jupyter notebooks have included ways to detect *name-value inconsistencies* [31] and data leakage between training and test sets [60]. Subotić et al. introduced a formal notebook execution model to compute the abstract state of the notebook under different combinations of run-orders [50]. Other systems have analyzed execution order to detect "stale" state [29]. Head et al. utilize traditional program slicing techniques to extract "clean" slices of a "messy" notebook [13], while Shankar et al. later developed a dynamic, notebook-specific slicing technique [48]. This work does not introduce a new analysis technique. However, because some analysis tools are aimed at finding instances of problems that Pagebreaks can solve (e.g. stale state), it is easy to see how they are complimentary. For example, if a programmer had not used Pagebreaks during their development process to date, they could use a whole-notebook analysis tool to find places where Pagebreaks might help them avoid future problems.

3 FUNCTIONS IMPEDE KEY NOTEBOOK INTERACTIONS

Previous work has observed that notebook programmers tend not to use functions during iterative or exploratory use of notebooks [20, 31], and that they often write functions as a post-hoc organization step [10]. This leads us to ask: *Why might notebook programmers use functions less often?* In this section, we suggest an answer based on combining (i) existing qualitative findings about notebook programmers with an an analysis of (ii) the programming environment (Jupyter/IPython) and (iii) the programming language (Python).

Technical Observations: Functions live in one cell, and variables defined in functions live in one cell We'll first make two technical observations about the JupyterLab/IPython environment:

Observation 1:
Function definitions cannot be split across multiple cells.

When a user runs a notebook cell, IPython parses and runs just that cell; the notebook environment thus enforces that each cell is parseable in isolation. Because partial function definitions are not individually parseable in Python, the entire function definition must appear in a single cell.

Observation 2:
Variables defined within a function are not persisted in the global state.

Python uses lexical scoping [1] and standard function scopes [16]. With lexical scoping and function scopes, variables defined within a function body can only be accessed within that same function body. Function scopes thus give both programmers and compiler developers useful guarantees about when a name can no longer be used. [1] Global variables, in contrast, are "in scope" in any part of the source code. We use the term *global state* to refer to the set of all global variables whose definitions have already been run. All cells in a notebook can read, update, or introduce global variables. In a notebook programming environment, global variables represent the primary means of communicating across cells; one cell can write to a global variable, and a cell executed later can read from it. In contrast, variables defined in a function body are not written into the global state—they are available strictly within the function body. Since functions are constrained to appear in their entirety in a single cell, this means that variables defined in a function are available only within a single cell.

User Operations: Functions limit organization across cells and variable inspection Next, we connect these observations to user "operations" within Jupyter notebooks, asking: *What operations become difficult or impossible in Jupyter Notebooks due to Observations 1 and 2?* First, organizing code across cells—splitting cells, migrating code between cells, and merging cells—becomes more difficult. Because functions must be defined in individual cells, code within function bodies cannot be split across multiple cells without removing it from the function definition, changing the program's control flow. Utilizing cells as ways to manually direct control flow thus becomes more difficult with the use of functions. Second, because local variables defined within functions are not

[1]This is not the case in all languages, such as languages that use dynamic scope, or in some cases in Python, such as running Python in a debug mode, or using the "global" keyword.

persisted in the global state, users cannot inspect or use those variables in later cells. Communicating between cells thus becomes more difficult when we use functions.

Exploratory Interactions: Cells as alternatives, interleaving code and output, and small cells for exploring next steps Finally, we connect the operations that are difficult with functions—organizing code across cells, inspecting variables after the cell in which they're defined—to exploratory interaction patterns. Existing qualitative findings about notebook usage tell us why losing these operations might hamper an exploratory workflow. Kery identifies three practices for the exploratory notebook programming style they termed "expand-reduce behavior": 1. Cells functioned as small alternative versions, allowing a programmer to experiment with different approaches, 2. Because the cell output appears immediately after the cell, small cells allowed programmers to *interleave* code and output, aiding inspection and reasoning, and 3. Novice programmers tend to use single-line cells during their development process, repeatedly editing and running a one-line cell representing the next line to be added to a larger program [20].

With our technical observations in hand, we can see how function use inhibits each of these practices:

Cells as alternatives: Kery et al. [20] identified an "alternatives workflow," in which programmers manually decide control flow by running one of two cells (or groups of cells). This typically involves a structure (shown in Figure 2) in which: (i) one or more "upstream" cells write values into the global state, setting up all inputs to the alternative cells, (ii) two or more "alternative" cells read the same values from the global state and write *different* values (with the same names) to the global state, and (iii) one or more "downstream" cells read the values written by the most recently executed alternative. The programmer directs control flow manually by executing the upstream cells, one of the alternative cells, then the downstream cells. Clearly, since this alternatives workflow centers on manually directing control flow via picking which cell to run, and function bodies must be contained within a single cell, programmers cannot use this workflow *within* a function body. Achieving this workflow also requires all three of the components—upstream, alternatives, and downstream—to have access to the same variables. This means that not only is it impossible to use this workflow *within* a function body, but even using a function for *part* of the process interferes with the workflow. That is, if the upstream cells define variables within a function scope, the alternatives cannot read them. If the alternatives define variables within a function scope, the downstream cells that react to the various alternatives cannot read those. While there are of course other non-cell ways of exploring alternatives, the key point here is that functions hamper this specific cell-based alternatives workflow.

Small cells for displaying intermediate values interleaved with code: Because function bodies cannot be split across cells, users cannot print intermediate outputs alongside the statements that produce them. While adding `print` statements within a function body allows programmers to display *all* printed intermediate values at once, those printed values appear together—not interleaved with the relevant statements—and they appear where the function is called rather than where the function is defined. Recall that notebook programmers use small cells in order to show intermediate outputs right alongside the line that produced them. `print` statements thus do not allow notebook programmers to recover the benefits of interleaving code and output for function bodies.

Building programs one line at a time: Following the same logic as above, users cannot iteratively compose a program a single line at a time when that composition is within a function. Kery [20] identified that especially novice programmers valued being able to see the immediate consequences of running one- or two-line cells. They used repeated runs of these small cells to confirm assumptions or catch confusions before moving on. This workflow relies on statements being easily spread between multiple cells. It also relies on the program's behavior remaining stable if a list of cells is run in sequence or if the cells' contents appear together within a single cell in the same order. If moving a line of code between cells also means moving it in or out of a function body, these are not safe assumptions.

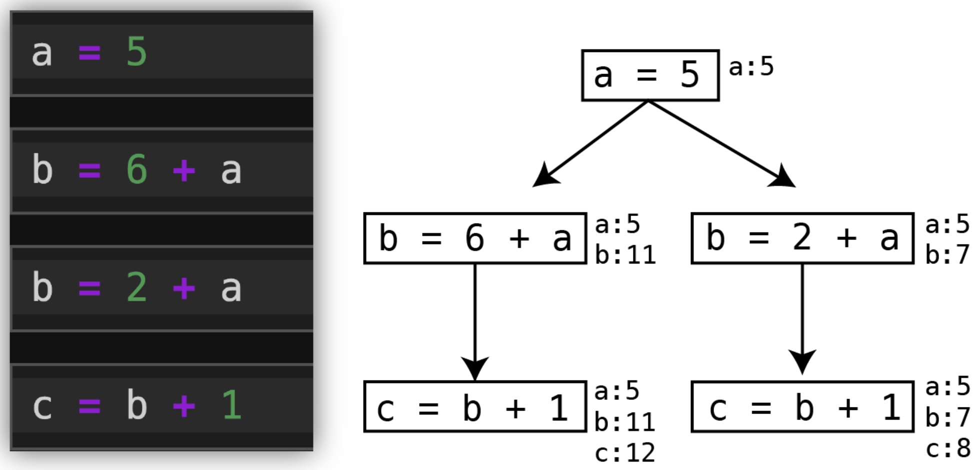
Functions Interfere with Exploratory Workflows From this analysis, we can offer one possible hypothesis for why functions may not appeal to notebook users: ***Functions interfere with the iterative, flexible style which many users value about exploratory notebook programming.*** Specifically, we hypothesize that functions seem to interfere with users' ability to 1. use cells as alternatives, 2. display intermediate outputs alongside the code that produces them, and 3. flexibly inspect the results of just 1–2 lines of code appended to a longer code block. From our analysis of operations, we posit that these interactions demand: 1. In order to use cells to manually direct control flow, programmers need to flexibly move code in and out of cells without changing overall program behavior 2. In order to communicate between cells, programmers need multiple neighboring cells to share easy access to variables.

By analyzing the programming environment, language, and findings about users together, we can now place a design requirement on our system: that any intervention must *preserve* these three exploratory interactions in order to avoid the same drawbacks of functions in notebook environments.

4 GLOBAL VARIABLES ARE AT THE ROOT OF MANY NOTEBOOK PROGRAMMING STRUGGLES

In the previous section we hypothesized that using functions in notebooks seems to inhibit many of the exploratory interactions that draw users to notebook programming. Notebook programmers flexibly split programs into cells as a way to manually direct control flow, which allows them to: create impromptu alternatives, interleave code and output, and iteratively develop 1- or 2-line code snippets at the end of their programs. Because cells must be parseable in isolation from the rest of the program, any given function or object is constrained to just one cell; thus, programmers can't use variables defined in function scopes to communicate between cells. Instead, communicating across cells means writing and reading global variables.

Using global variables as the primary way to communicate between cells, however, seems to come with costs. Given how often use of global variables is discouraged in traditional programming environments [14, 46], it seems unsurprising that notebook programmers report a number of confusions and issues with managing a large number of them. As we discussed in Section 2.2, notebook



```
a = 5

b = 6 + a

b = 2 + a

c = b + 1
```

Figure 2: Notebook programmers can use cells as *alternatives* by manually changing the execution order. Left: A sample program. An "upstream" cell `a = 5` writes `a`, a global variable. Two alternative cells, `b = 6 + a` and `b = 2 + a`, both read the global variable `a`. They both also write a global variable, `b`, although they write different values to `b`. Finally a "downstream" cell, `c = b + 1`, reads the global variable `b`. Right: A representation of two different executions of the notebook on the left. The programmer can explore the alternatives one at a time. If the programmer runs cells 1, 2, and 4, they see the outcome on the left. If they run cells 1, 3, and 4, they see the outcome on the right. By only executing one of the two alternative cells, the programmer makes `c` dependent on only one alternative, allowing them to experiment with the results of each alternative iteratively.

programmers can face compounded difficulties because cell run order is not fixed, so the program text is not enough to tell a programmer which variable definition ran last.

Some programmers attempt to mitigate the hardships of tracing global variables through notebooks by using a series of related names for key variables—e.g. datasets that are modified throughout a notebook. One participant in Weinman et al. [57] recounts using "either uninformative names like `df_1` and `df_2`, or increasingly onerous names like `df_no_personality`", but would then often "forget about the names and then have to go up and reread how they defined it." While Weinman et al. discuss this user's issues in the context of defining alternatives, the underlying technical issue is a scope problem: because any cell can redefine any variable in the global state, programmers struggle to guess which cell or sequence of cells updated a variable's value en route to the current state. In the context of alternatives, this means programmers struggle to ensure that two related sets of variables do not interact without making new variable names.

The number of global variables and the distance between writes and reads can also confuse programmers. Notebook programmers struggle when they change a variable's name without changing cells elsewhere in the notebook that read from the old name [52]. Head and Kery both found that the complexity imposed by keeping track of a growing global state and run order, in part, created practical length limits on notebooks [13, 23], leading programmers to start new notebooks when the existing one became too difficult to understand. Because the global state is persistent over cell runs, Rule et al. recommend regularly restarting and running the entire notebook in linear order to ensure the current global state is reproducible [43]. This advice, however, does not fit situations in which programmers want to run only one alternative cell in a set of alternatives. It also imposes a time cost if some computations are time-intensive. Managing many global variables, aside from leading to errors, also then seems to heighten the burdens of tracing and understanding a notebook.

In short, managing a large global state can be confusing because the programmer must keep track of many interacting variables at once. This can be especially problematic in notebooks, where running cells to direct control flow can result in the assignment of different values to those variable names depending on the cell execution order.

The issues mentioned above are closely related to what lexical scopes are intended to address in other programming environments. Lexical scopes separate programs into discrete source code segments, in which programmers can tell whether names are valid based on their place in the text of the program. In essence, scopes segment programs into smaller, discrete units that programmers can reason about in isolation [14].

5 IMPLEMENTING NOTEBOOK SCOPES WITHOUT FUNCTIONS

So far we have examined both how functions impede users and how global variables impede users. If we want to avoid both functions and global variables, what tools do we have available? In the previous section, we traced the connection between global variables and many of the well-known difficulties associated with notebook programming. In other programming environments, we often avoid proliferating global variables by using functions and function scopes. But in a notebooks context, functions hamper desirable exploratory interactions, as we explored in Section 3. This suggests a next question: Can we get scopes without functions?

Here we note that a function serves many different purposes—it names a chunk of code, provides a way to run the code multiple times, a way to name parameters, provides new control flow options based on being able to return mid-function, *and* it introduces a scope. However, we can achieve that last purpose without reproducing this whole list of disparate roles. That is, we can introduce a stripped-down language construct that does much *less* than functions, while still offering scopes.

Our analysis in the previous sections has outlined a set of design requirements for an intervention:

- Writing functionless programs means giving up scopes, which produces large global states. To do better than functionless programming, **our new construct should give programmers a way to communicate between cells without writing into global state**.
- Using functions to achieve scopes means giving up cell-based manual control over control flow. To do better than using functions, **our new construct should allow programmers to independently execute an arbitrary number of communicating cells**.

Taken together, these requirements suggest a construct which **includes multiple cells inside a single non-global scope**.

From these design requirements we built *Pagebreaks*, which allows contiguous groups of Jupyter notebook cells to share a non-global scope. The name is an analogy to "page breaks" in a document editor, which visually segment long documents into conceptually distinct regions. Pagebreaks segment a notebook program into regions that share a scope. While a new Pagebreak also produces visual segmentation, it most importantly changes the meaning of the program. Pagebreaks is implemented as a Jupyter interface plugin with a paired IPython extension. The plugin allows users to organize cells into Pagebreaks through the Jupyter interface, and it transforms the Python programs behind the scenes when the programmer executes a cell.

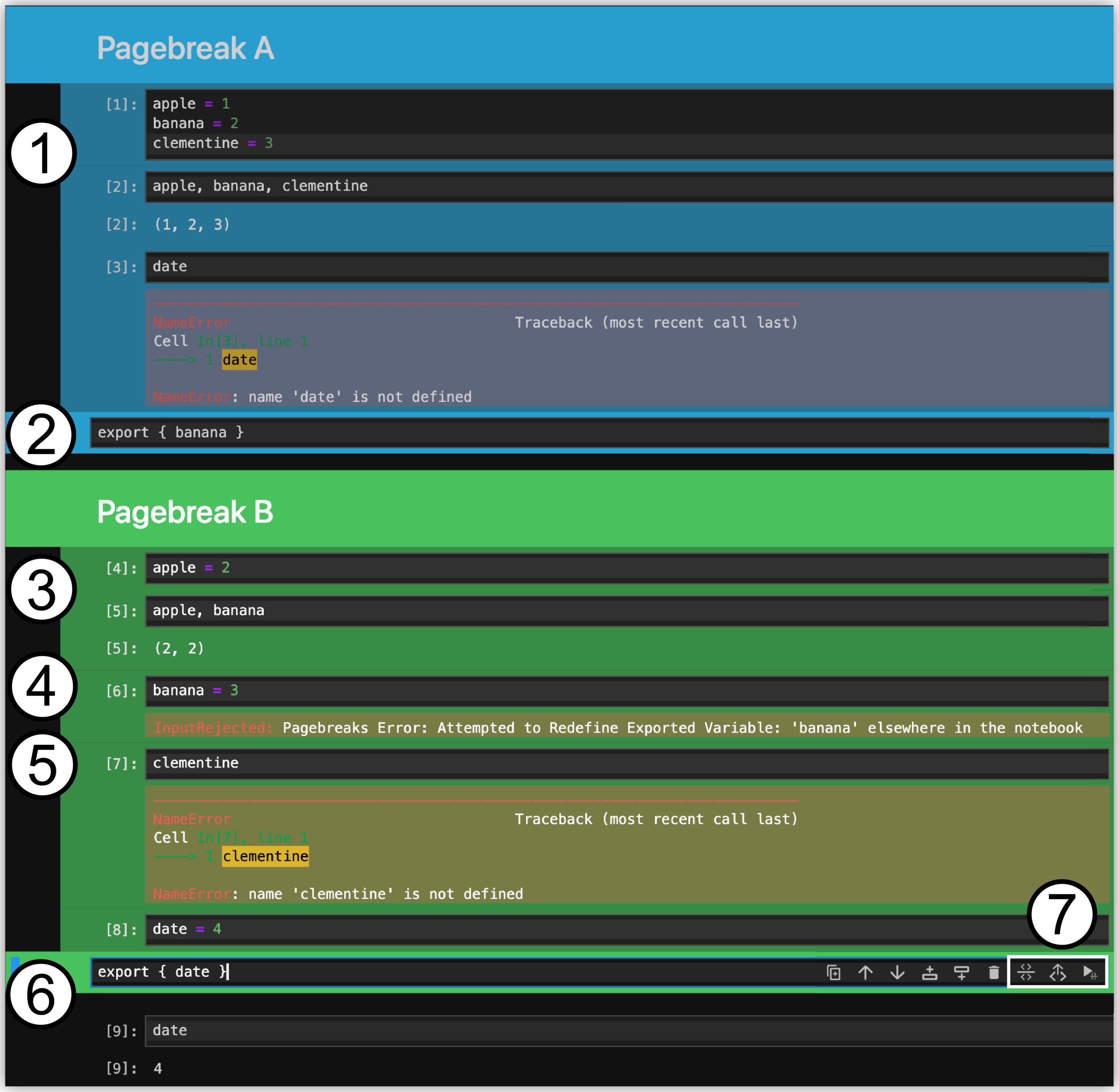
A single Pagebreak is a grouping of cells that share a scope. Variables defined in that scope are available to all cells in the Pagebreak. Variables defined in a nested scope—e.g. a function defined in a cell inside the Pagebreak—remain available only in the nested scope, as we would expect in Python.

Programmers add Pagebreaks to a notebook using a button, as shown in Figure 3, which creates a "header" markdown cell and a "footer" raw text cell. The code cells between the header and footer are indented to visually indicate the extent of the Pagebreak, reminiscent of the indentation of statements within a function definition. Users do not have to change the code inside their code cells to use Pagebreaks; the location of a cell within the header and footer cells indicates that it will execute within the Pagebreak scope. When the user executes a cell, the Pagebreaks system automatically sends the information about the cell's scope to the IPython environment.

Users can choose to "export" variables from a Pagebreak, making them readable by later cells, by including them in the "footer" cell, marked by the `export {...}` syntax. Variables exported in this way remain modifiable inside the Pagebreak where they are defined, but cells in later Pagebreaks cannot modify them. We highlight two key Pagebreaks design choices:

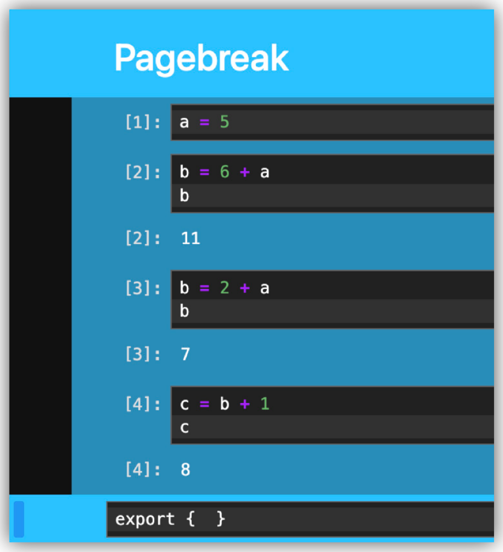
Pagebreak Scopes Are Linearly Ordered. Following the existing metaphor of the Jupyter notebook as an ordered list of cells, variables "exported" from a Pagebreak become available to all later (i.e. visually below in the notebook) Pagebreak scopes, and remain inaccessible to earlier (i.e. visually above in the notebook) Pagebreak scopes. Prior work indicates that running cells in non-linear order is a key component of exploratory workflows; however, this workflow did not include reading variables that are defined later in the notebook, a practice that would guarantee that linear notebook runs will error. For example in Figure 3, Pagebreak B defines and exports a variable `date`, which is available for all Pagebreaks visually below Pagebreak B (and any cell within Pagebreak B), but is not available in Pagebreak A, as shown in cell `[3]`. (Cells in Pagebreak A would also be unable to define a variable `date`, since a `date` has already been exported by another Pagebreak.) The availability of exported variables depends on the order of the Pagebreaks when a cell is executed, not the order of Pagebreaks when the variable was first added to the program text. Thus if (in Figure 3) Pagebreak A was dragged below Pagebreak B and cell `[3]` in Pagebreak A was executed, it would run without error, because `date` is exported by a Pagebreak that would appear earlier in the notebook. Our linear order design also ensures that no cyclical dependencies between Pagebreaks are possible, which could be a source of confusion.

Read-Only Exports. While iteratively modifying a shared name was an important part of exploratory notebook interactions, redefinitions were also a source of confusion in larger notebooks, especially when it occurred far from the original definition. To support iterative modification of a shared name between nearby cells while seeking to alleviate the confusion of modifications to shared names throughout a large notebook, Pagebreaks enforces that exported variables are read-only outside of the Pagebreak which exports them. For example, in Figure 3, the variable `banana` is defined in Pagebreak A, and could be redefined by any cell in Pagebreak A without error; in Pagebreak B, however, attempting to define the variable `banana` results in an error from the Pagebreaks system (as shown in cell `[6]`), because `banana` has been exported by another Pagebreak. Because Python does not have a built-in way to enforce that a variable cannot be modified, the Pagebreaks system implements this internally. Before cell execution, Pagebreaks analyzes the cell's AST (Abstract Syntax Tree) to identify any attempted modifications to an exported variable outside its original Pagebreak, and, if it finds one, will throw an error. Since some modifications cannot be identified statically, Pagebreaks also uses dynamic analysis, comparing the value of exported variables before and after a cell run to look for changes; the system automatically reverts the notebook state and displays a warning message if a cell attempts to modify a read-only variable dynamically.



```
[1]: apple = 1
     banana = 2
     clementine = 3

[2]: apple, banana, clementine

[2]: (1, 2, 3)

[3]: date

NameError                                 Traceback (most recent call last)
Cell In[3], line 1
----> 1 date

NameError: name 'date' is not defined

export { banana }

[4]: apple = 2

[5]: apple, banana

[5]: (2, 2)

[6]: banana = 3

InputRejected: Pagebreaks Error: Attempted to Redefine Exported Variable: 'banana' elsewhere in the notebook

[7]: clementine

NameError                                 Traceback (most recent call last)
Cell In[7], line 1
----> 1 clementine

NameError: name 'clementine' is not defined

[8]: date = 4

export { date }

[9]: date

[9]: 4
```

Figure 3: The Pagebreaks Interface. 1. Pagebreak A includes three definitions of variables, for `apple`, `banana`, and `clementine`. All cells in Pagebreak A share access to these variables; e.g. cell `[2]` can read or write `apple`, `banana`, and `clementine`, which are defined in cell `[1]`. 2. Variable `banana` is exported from Pagebreak A and becomes available to later Pagebreaks, including Pagebreak B, in a read-only form. 3. Pagebreak B contains a new definition of `apple`. This introduces a new variable `apple` in Pagebreak B's scope, not a redefinition of the `apple` defined in Pagebreak A. Because `banana` has been exported from Pagebreak A, it can also be read in Pagebreak B. 4. Because exported variables are read-only, attempting to redefine `banana` within Pagebreak B produces an error. 5. Because variable `clementine` was only defined in Pagebreak A and not exported, attempting to print it in Pagebreak B produces an error, since `clementine` is not in scope in Pagebreak B. 7. We added three context menu buttons, from left to right: (1) Add New Pagebreak (2) Merge Pagebreak Above (3) Run All Cells in Pagebreak.



```
[1]: a = 5

[2]: b = 6 + a
     b

[2]: 11

[3]: b = 2 + a
     b

[3]: 7

[4]: c = b + 1
     c

[4]: 8

export {  }
```

Figure 4: We recreate the example program in Figure 2 within a Pagebreak to illustrate each of the exploratory interactions we aim to preserve: (1) Cells as Alternatives: Within a Pagebreak, neighboring cells have access to the same variables (above, variable `b`), so Pagebreaks support the alternatives workflow. (2) Interleaving Code and Outputs: A Pagebreak can include multiple cells, so code and output can be interleaved as usual, as shown above. (3) Building Programs a Line at a Time: Neighboring cells in a Pagebreak can read and write the same variables, so users can freely split one or two lines out of a longer code snippet, in order to iteratively edit and rerun one subprogram at a time. Here see that cell 4 can use `b` as part of a new snippet, even though `b` is defined in another cell.

5.0.1 Implementation. The Pagebreaks system implements scopes by transforming variable names before the Python interpreter executes a given cell. The Pagebreaks "backend," an IPython plugin, transforms the AST of a program before it is executed, prepending user-provided variable names with a prefix unique to each Pagebreak. For example, a variable `a` in Pagebreak `0` would be renamed `pb_0_a`. Only variables defined in the Pagebreak scope are transformed; variables defined in nested scopes inside the Pagebreak—e.g. within a function or class definition—are left unchanged, as are package imports. We provide an IPython “magic command” for seeing the list of untransformed names associated with each Pagebreak. Users can see the prefixes by inspecting the notebook state with the debugger or the built-in IPython magic `%who_ls`. Otherwise variable name prefixes are invisible to the user, allowing them to utilize non-global scopes without having to understand how they are implemented. Because we implement Pagebreak scopes by transforming the AST representation of each program, the Pagebreaks system can run as a layer between IPython and the Python interpreter without requiring a modification of either system.

For our in-situ study, we also implemented many features to integrate Pagebreaks into a Jupyter environment, such as collapsing and dragging entire Pagebreaks, merging Pagebreaks together, running all cells within a Pagebreak in order, and an option to save a notebook with its transformed variable names represented in the text of the cells, enabling users to share a notebook with others without the extension installed.

Pagebreaks is implemented in ~3500 lines of Typescript and ~1100 lines of Python, and is available open-source.[2]

6 USER STUDY DESIGN

To understand how Pagebreaks interacts with notebook programming behaviors in real situations, we introduced Pagebreaks to five experienced notebook programmers and invited them to use it in-situ [42] in their everyday work for about two weeks each.

6.0.1 Participants. We recruited participants from social media (Reddit and Jupyter Forums) and relevant academic email lists. We used a screening survey to select participants who were using Jupyter notebooks regularly in their work. At the beginning of the study we met with each participant to introduce them to Pagebreaks, address questions, and ensure they had sufficient information to give informed consent. Participants were compensated for the time they spent interviewing at $30/hr.

At the end of the trial period, we interviewed participants about their experiences for 1-1.5 hours, asking them to narrate the notebooks they worked on within a semi-structured interview. We asked them for feedback on Pagebreaks and to describe how they used it within the context of their programming. All interviews were recorded over Zoom and transcribed for analysis.

6.0.2 Analysis. We analyzed the interview data in a thematic analysis [4]. Since our study specifically investigates the role of scopes in notebook systems, our analysis emphasized a *theoretical* focus, in that the analysis prioritized producing a deep description of the role of scopes in particular, rather than seeking to characterize our participants' responses broadly. Apart from focusing on scopes, we were also driven by the framing described in Sections 3 and 4 to look for interactions between Pagebreaks and both the exploratory interactions and programming barriers identified in previous work. Additionally, for many of our participants distinguishing between a function and its specific effect as a scoping construct was not a familiar way of speaking. Thus our analysis was often *latent*, as we had to interpret the role of scopes (and other elements of the programming language and environment) within the experiences of our participants. For example, some of our participants used the word "function" to describe code without a function definition, or described Pagebreaks as "functions", which we analyze in Section 7.4. While these mark areas of emphasis in the study rather than formal dimensions [5], they highlight how we brought our specific interests and expertise as human-computer interaction and programming languages researchers to developing themes. The first author transcribed and performed an initial coding, and themes were developed iteratively between the first and second authors.

[2]Source available at *https://github.com/erawn/pagebreaks*

ID	Position	Domain of Expertise	Yrs. Programming	Yrs. using Notebooks
P1	Computer Science/Data Science	Student/Researcher	5	3
P2	Data Science	Student	3	2
P3	Computer Science	Student	8	1
P4	Computer Science	Researcher	6	5
P5	Computer Science	Researcher	14	9

Table 1: While our participants had varying backgrounds, each had significant experience with both programming in general and computational notebooks

7 FINDINGS

We present two overarching categories of findings from our analysis. First, we discuss *the role of Pagebreaks in our participants' programming practices*: Many participants found Pagebreaks useful in alleviating issues with global variables (Sec. 7.1) while still feeling free to use notebooks in familiar ways (Sec. 7.2). We also saw that Pagebreaks seemed to play many *organizational* roles for our participants (Sec 7.3). Second, in talking about the role of Pagebreaks, our participants often relied on comparisons to functions, and so our second category of findings focuses on *the role of functions and scopes in general* in notebook programming (Sec 7.4). Our participants helped us understand many of the barriers they face when they try to use functions and how they avoided those barriers, both before the study period with their usual notebook programming strategies and within our study period by using Pagebreaks.

Because we asked participants to use Pagebreaks however it was useful to them in their everyday work, there were varying levels of engagement. P4 used Pagebreaks extensively, creating a notebook entirely in Pagebreaks which was the focus of their notebook programming during the trial period. P2 used Pagebreaks to organize a data cleaning workflow for a current side project. P3 was a data science student, who used Pagebreaks for some of their assignments, but ran into problems using the course's autograder system with Pagebreaks. P5 used a very lengthy notebook during the study period, adding a small number of pagebreaks because the notebook was predominantly being used to execute the already written code on a remote server, and not for active development. P1 worked on a large research project within a single notebook, but because of changing project timelines, completed the research as the trial period was beginning, and used Pagebreaks in order to understand its functionality but not in their active work. The interviews shifted focus based on participants' usage: for P4, P2 and P3, the interview focused on how they had used Pagebreaks in the study period. For P5 and P1, the interview focused on the notebooks work they had done before the study period, narrating the development of their notebook and discussing how they would like to use Pagebreaks, now that they had used it enough to understand its functionality.

7.1 Participants Used Pagebreaks to Help Address Issues with Global Variables

Participants (P3, P4, P1) discussed a number of ways Pagebreaks helped alleviate some of the issues of managing a large number of global variables. First, P3 and P4 mentioned being able to reuse common variable names (e.g. `model`, `train`, `test`) in different Pagebreaks without fear that they would accidentally redefine a variable elsewhere in the notebook. P4: "If I'm trying to implement something and I defined something down below just to test something, and I gave it the same name as something else, within Pagebreaks I know I don't have to worry about that (P4)". P1 mentioned difficulties with overwriting variables which were used throughout the notebook, such as dataset variables like `df` or `data`, and becoming confused about which variable definition had run most recently. Because P1 had written this code directly prior to the study period, they said that Pagebreaks would have been helpful in avoiding that issue, and they wished they had had it earlier.

Participants also mentioned that Pagebreaks enabled them to avoid giving successive names to different versions of variables (P4 and P3), a practice designed to mitigate global variable confusion. P4 mentioned that, in contrast to using Pagebreaks, in traditional notebooks, "I had to get very creative with the names ... I had to give [each new definition] a new variable name ... like [process1, process2]". P3 said that in "Pagebreaks you can really just have the same [variable names] ... you can call them both 'model' without having it conflict" and said it makes notebooks "clearer".

7.2 Participants Continued to Engage in Exploratory Behaviors While Using Pagebreaks

In discussing how they had used Pagebreaks, many participants mentioned using notebooks in exploratory ways, including making alternatives (P4, P2, P3, P1), interleaving code and output (P3), and iteratively composing programs (P3). P4 used neighboring Pagebreaks to explore two different "scenarios (P4)", utilizing the same variable names and common functions to compare results. P2 used Pagebreaks to hold different versions of the same dataset variable in different stages of cleaning. P3 highlighted that before using Pagebreaks, they would use cells to organize code, which sometimes resulted in "putting everything into one big cell (P3)" which did not interleave code and output. Pagebreaks allowed them to "break that up into a process and put more explanation" by using multiple cells to express a section of their program and interleave intermediate results. P3 also drew attention to how Pagebreaks supported iterative composition: "To me the benefit of Jupyter notebooks is that you can run things line-by-line in cells ... and I think of Pagebreaks doing that but to a higher level (P3)".

Because we did not ask explicitly about these behaviors for fear of biasing the data, and our goal with Pagebreaks was not necessarily to enhance these exploratory interactions but only to leave them unaffected, we did not expect participants to actively mention these exploratory interactions unless Pagebreaks had interfered.

Nevertheless, some participants did explicitly mention experiences remaining stable; P4 and P3 mentioned that they felt their interactions with notebooks to be largely unchanged. P4 said that Pagebreaks aided their programming process "but not in a way requiring changes from me (P4)" and that the adoption of Pagebreaks "was a pretty seamless transition (P4)". P3 said something similar: "I feel like I'm still tied to using Jupyter in the same way, but you have this [i.e. Pagebreaks], which makes it nicer (P3)". No participants mentioned that Pagebreaks made their exploratory interactions or work with notebooks in general more difficult.

7.3 Participants Used Pagebreaks for Organization

Our participants described a number of different ways Pagebreaks helped them *organize* their code. While these were often described together, in our analysis we separated different facets of what this organization meant for our participants.

7.3.1 Compartmentalization. Many participants (P4, P1, and P2) mentioned that a benefit of Pagebreaks was being able to focus on just one part of their notebook at a time. By scoping a section of the notebook, in other words, they had an assurance that their code in one Pagebreak would not affect their code in others: "It gives me [a] guarantee that I know that what I've run is what is meant to be running (P4)". This seems to have helped prevent a class of confusing issues, when common names have different values than expected because of the run order of the cells (detailed in 7.1), but importantly do not error and "lead me down the wrong track because the order I ran the cells was not the correct one (P4)".

7.3.2 Dataflow Organization. One participant also mentioned that Pagebreaks helped them be more cognizant of the variables coming into and out of each Pagebreak by requiring variables to be explicitly exported: "it makes me think more about the namespace of variables ... [makes me] more intentional about ... [what] I want to export from one Pagebreak to another ... [and] the things I want to reuse (P4)."

7.3.3 By-Purpose Organization. Apart from the actual scope enforced by Pagebreaks, many participants mentioned that Pagebreaks helped them group cells which accomplished a discrete task, encouraging them to reflect on the purpose of their code throughout development (P2 and P3). P2 said that they were "more conscious (P2)" of what they intend to accomplish when they make a new Pagebreak, and that Pagebreaks encourages them to divide the notebook into more sections based on those intentions. P3 drew an explicit comparison to the way they used cells to organize code, describing Pagebreaks as "another layer of organization on top of [notebook cells] (P3)" which each contain "a subset of cells (P3)" which have a defined purpose.

Both P3 and P2 also mentioned being able to give the Pagebreak a title (because the top cell of a Pagebreak is a markdown cell), which was an important part of organizing by purpose: "...it was very useful when I named the Pagebreak what I did [within] it (P2)".

7.4 Functions in Notebook Programming

Many participants used the term “function” to describe sections of their notebook programs which had no function definitions. P1 and P5 both referred to large cells with defined purposes (but without function definitions) as "functions" while explaining their notebooks to us, using words like "parameters" and "arguments" to refer to variables defined in previous cells which were used within the "function" below. We use "function" in quotes to refer to our participants' words and *function* without quotes to refer to a standard function definition.

Some of our participants used the language of "functions" to also describe their experiences with Pagebreaks. When P4 discussed how using Pagebreak prompted them to think about dataflow in their notebook (see Sec 7.3.2), they said they were "thinking of the Pagebreak in and of itself as a function (P4)" not just because of its role as a way to introduce scopes, but because they were "thinking of pagebreaks as functions, its now a pretty useful mental model, like 'whats the input and output of this Pagebreak' (P4)." P5 proposed using Pagebreaks in place of a function definition in order to persist intermediate variables, and P1 discussed using Pagebreaks to contain different "function" variants.

Because both the language of "functions" and function definitions were prominent in how our participants discussed their programming and their use of Pagebreaks, we discuss how our participants discussed “functions” alongside some of the issues our participants encountered when using functions. We also describe how participants used Pagebreaks to address some of these issues.

7.4.1 Participants Talked About Parameterization. Both P1 and P5 discussed cells in their notebooks as "functions," cells which accomplished key units of work which took in "parameters" or "inputs" in the form of global variables defined in other cells. By duplicating cells where "parameter" global variables were defined or iteratively redefining them in-place, then re-running the "function" cell, they could see the results (outputs from the "function" cell) across these multiple different "inputs." Working this way caused confusion, however, when variables in the "parameters" cells could be updated and the "function" cell was not rerun, or users could not remember which version of the "function" cell output corresponded to which version of the "parameter" cells.

By using global variables and manually directing control flow via executing cells, these participants used some cells as "parameters" to a corresponding "function" code cell, without using a function definition. P1 mentioned that Pagebreaks would have been helpful in these parameterization workflows, using a non-global scope to pair separate versions of these "parameter" variables with duplicates of a "function" cell they had made.

7.4.2 Outside Functions, Intermediate Values Persist By Default. When asked about the benefits of Pagebreaks in their notebook programming, P5 mentioned wanting to use Pagebreaks to organize a part of their program rather than use a function definition in order to persist intermediate variables in partial executions. For time-intensive computations, they explained, if there is an error within a function body, none of the variables defined within that function are persisted in the global state (and the function, of course, does not return). In a Pagebreak (or in a standard notebook, outside of a function) intermediate variables are still accessible after an exception. If intermediate variables are still accessible, P5 explained, they can continue the computation from the last intermediate variable definition that ran successfully. Crucially, this persistence of intermediate variables was beneficial in the case of an unexpected error, when the participant would not know in advance what variables to persist; to avoid having to guess what to persist, they avoided using a function altogether and persisted all of them as global variables.

On the other hand, P1 mentioned that global variables persisting on partial executions can sometimes cause additional confusions. When they used variant "parameter" cells which defined a series of global variables with different values and ran a large "function" cell which partially executed, or ran multiple "function" alternatives at the same time, they were unsure which values corresponded to which "function" execution, and under which parameters: "...especially when these cells sequentially have the exact same variable names, because they're essentially the same cells with different parameters, I don't even know what result holds right now ... obviously you can go from the run numbers, but while one cell is partially running, and I don't know what errored and what didn't—it's complicated (P1)." P1 mentioned that using a Pagebreak scope would have been helpful in order to separate these alternate versions, so that variables defined from partial execution were persisted separately from other alternatives.

7.4.3 Separation between Function Definition and Call Site was Sometimes Undesired. P1 had a practice of placing a function definition in the only cell that calls it. When multiple cells included calls to the same function, they would make these cells contiguous and put the function definition in the first cell that calls it. This practice helped them avoid tracing code throughout the notebook when running a cell with a function call: "I tend to keep [function calls and their definitions] quite close together, so I'm doing one computation in one particular section (P1)." In short, P1 wanted to use functions to avoid code duplication (they wanted the option to call a function repeatedly), but they did not want to use functions as a mechanism for transferring control from one region of the source code to another. Since the ability to transfer control between source code regions is often a benefit of using functions in other programming environments, P1's rejection of this function role is notable here.

8 DISCUSSION

8.1 In Defense of "Messy" Programming

One common theme of the language participants (P1, P4, P5) used to describe their notebook programming was "lazy" or "messy", especially with regard to why they had not used more function definitions in their notebooks. Given the many sources which urge using functions and avoiding running cells "out-of-order" [36, 43, 49], it is perhaps unsurprising that many of our participants spoke this way about their notebook programming practices.

One insight of this paper is to help explain *why* notebook users continue to program in ways they recognize as unstructured or error-prone: following the "best practices" of notebook programming might force them to give up the exploratory interactions they value. Our user study of Pagebreaks suggests that notebook programmers might not simply need "better habits", but better programming environments — environments which do not make them choose between exploration and organization. The participants which called themselves "lazy" recognized the value of functions in traditional software environments, but, for many of the reasons we have discussed, did not feel that the cost of function use was worth it — they made a reasonable choice about the most appropriate ways to use their programming systems. P4 discussed their choice to use functions in notebooks as a threshold: when the confusion caused by *not* using a function outweighed the burden imposed by using one (which was usually only when code was being repeated three or more times). The positive reception of Pagebreaks by our participants suggests that making a new Pagebreak presented a smaller burden than making a function, and so helped notebook programmers use more scopes. At least in this case here, when programmers are given constructs to better organize "messy" code which do not interfere with how they want to work, they seem to use them.

8.2 Breaking Open Functions

The idea of breaking apart functions and focusing just on non-global scoping was not obvious to researchers in advance, nor was it obvious for our participants: scopes were so closely associated with functions for our participants that for many we could explain what Pagebreaks did only by analogy to local variables within functions. Isolating scopes from functions with Pagebreaks, however, helped some of our participants avoid issues with global variables while supporting their exploratory notebook programming. As our framing in Sections 3 and 4 tries to argue, this separation was a key part of both Pagebreaks' design and, we think, its success.

Additionally, separating scopes from functions also helped us understand something about how programmers view both scopes and functions, which we would not have learned without Pagebreaks as an intervention. In our user study, we found many different roles that *just scopes* could play in notebook programming, and the ways "functions" appeared in programmers' descriptions of code which had no function definitions.

8.3 Designing for Programming Languages and Programming Environments Together

One critical piece of how we framed and designed Pagebreaks was thinking across the design of the language (Python) and the programming environment (IPython/Jupyter). Our initial analysis in Sections 3 and 4 connected user findings with environment implications and suggested that functions, as a language construct, presented specific difficulties when used in notebooks due to the interactions of the Python language (i.e. parseability, non-persistent local variables), the IPython system (i.e. individual cells must be parseable, only global variables persisted) and the Jupyter environment (i.e. functions cannot be defined across cells, passing references between cells requires global variables).

While the intended functionality of Pagebreaks (scopes across multiple cells) was conceptually simple, implementing it required engineering on each of these levels, which are often thought of separately: Pagebreaks transforms the ASTs of the programs based on the visual organization of cells in the Jupyter interface, creating dynamic "export" variables within the IPython instance.

Throughout this paper, we have described Pagebreaks as a "language construct" to refer specifically to the part of the work that changes the semantics of users' programs and the results of running those programs. In the course of implementing Pagebreaks, we also introduced changes to the programming environment—e.g. different color backgrounds for different Pagebreaks, visual indentation of code cells within a Pagebreak, adding buttons for creating and modifying Pagebreaks, etc. While programming "languages" and "environments" are often understood as distinct, we would echo the call of Jakubovic et al. to view both more holistically as common elements of programming *systems* [17]; even though we find it useful in this paper to talk about which parts of our work change program semantics in contrast to those which change interface elements, study participants' language suggests they do not draw this distinction in practice. In Jakubovic's terminology, our work is an attempt to understand Python, IPython, and Jupyter as interlocking elements of one common programming system, and designing an intervention for that system.

8.4 Designing for Programming Languages, Programming Environments, and Programming Audiences Together

Beyond thinking about the interactions between programming languages and programming environments, Pagebreaks also required integrating user findings to understand the environment and language implications of users' specific priorities and behaviors. Thinking across these three layers was important both for understanding our design choices and making sense of our participants' experiences.

Our findings on organization suggest that these three layers were required to understand our participants: in describing how Pagebreaks "organized (P2, P4)" their notebooks or made them more "clear (P2, P3)", participants referred jointly to how Pagebreaks visually organized their notebook interface through color-coding and visual hierarchy, how it organized code by-purpose into distinct units, how it compartmentalized their code by enforcing a scope boundary at the language level, and how it delineated the *dataflow* — the variables coming in and out of a Pagebreak — at the system level. While we separated these facets in our analysis, for many of our participants they seemed to play a single role of *organizing* their code.

Thinking about the language and system implications of user behaviors became necessary to understand our system design choices as well: P5 mentioned they would like to create Pagebreaks in the middle of existing notebooks in order to gradually transition it to entirely using Pagebreaks, while P3 envisioned using Pagebreaks only within the context of making alternatives, with the rest of the notebook not using Pagebreaks. This revealed an assumption we had made in the design process: that Pagebreaks were most easily understood as structuring a whole notebook, and that users should, therefore, start a notebook with Pagebreaks in mind or reorganize an existing notebook entirely into Pagebreaks.

When discussing the possibility of creating normal, global-scoped cells alongside Pagebreaks, P4 strongly disagreed, saying the change "goes against the point of Pagebreaks, having [cells] in between [Pagebreaks] (P4)." As P4 explained, the "point of Pagebreaks" for them was the assurance that, within a given Pagebreak, only variables defined within it or explicitly exported from other Pagebreaks were in scope. Allowing cells to directly write variables to the global state — bypassing Pagebreaks — threatened this assurance (at least for P4).

While the technical solution to this problem may be simply to have a setting to toggle between two modes, the insight is that scopes played different roles for users with different priorities. This decision on the design of Pagebreak scopes could not be separated from the practical role those scopes played for different users.

9 OPPORTUNITIES FOR DESIGN: THREE LENSES

Building on the insights from the discussion above, we turn them outwards towards future work, reframing them as lenses on design.

Problem Discoveries: Turn Complaints about Notebook Users Into Opportunities for Language Constructs.

Expanding on our discussion of "lazy" or "messy" notebook programming in Section 8.1, we propose seeing notebook programmers' lack of adherence to "best practices" as an opportunity for design. As this work tries to show, if notebook programmers are using functions less often, it might be because their systems have been poorly supporting function use in the ways they desire to work. In contrast to the usual problem discovery method of looking for problems, or looking for where users are struggling, might we also look at “best practices” recommendations to find jumping off points for design? Calls for best practices may point us to situations where programmer needs and goals are in tension with how their programming environments are shaping their behaviors.

Problem Definitions: Look at the Language, System, and Audience Together

As we discuss in Section 8.4, language designs that are effective in traditional programming environments (like function definitions in Python) may not be the right designs for other environments like notebooks. Future work in notebook programming environments which does not consider the language, the environment, and the audience together may be leaving important inspiration on the table.

Problem Solutions: Break Open Language Constructs

The key insight for the design of Pagebreaks was that scopes do not have to be bundled up in functions. As researchers, we had initially assumed (like our participants) that established language constructs like functions are the only ones available, and we initially focused on mitigating the problems of functions. Ultimately we picked a different path, introducing an alternative language construct with specific attention to how it fit into both the environment and the needs of our users.

In our own design process, listing functions' many different roles was a very useful exercise. Rather than taking a language construct like a function as fixed, there are opportunities to break a construct into its component parts, take one part in isolation, recombine parts, or combine parts of multiple mainstream constructs. We hope this work will inspire deeper reflection about the roles of language constructs and inspire more opportunities for redesigning them.

10 LIMITATIONS

Because our study was time-intensive from both a researcher and participant perspective, it was both difficult to recruit many suitable participants and support them during the study period (e.g. responding to bugs with fixes, assisting if confusions arose, etc). More participants, especially given the diversity of how users interact with notebooks, would have given us a greater diversity of experiences, or helped us see additional contexts where Pagebreaks was or was not successful.

Beyond the number of participants, the duration of the study (~2 weeks), was sometimes not sufficient for the often sporadic use of notebooks by our participants. Even when participants were using notebooks they sometimes were not actively writing code for periods longer than our study duration (as in P5's case). A longer study would have allowed us to observe Pagebreaks' use throughout a full development process.

Because one goal of Pagebreaks was to leave exploratory notebook interactions *unaffected*, we faced difficulty assessing this directly in qualitative analysis of interviews, since participants were much more likely to mention the *differences* they had noticed upon using Pagebreaks rather than what *did not* change about their programming practices. A different kind of analysis which focused on fine-grained, keystroke-level data, either through video taping or recording telemetry data about users' programming, might be required to rigorously assess how these behaviors did or did not change.

Looking towards a complete release of Pagebreaks, we look forward to implementing two features suggested by our study: (1) customizing the colors and styles of Pagebreaks visually through the settings menu (which was requested by P2 and P7 at the conclusion of the study) and (2) highlighting exported variable names with the color of their associated Pagebreak, helping users more easily identify exported variables throughout the notebook.

11 CONCLUSION

This work began with observing a tension: notebook programmers reported confusions with global variables but did not seem to use functions to address them. To understand why and work towards possible solutions, we analyzed the programming environment and programming language alongside user behaviors, finding that functions seem to inhibit many of the exploratory interactions users value. We then designed a system, Pagebreaks, by isolating the part of functions we needed—non-global scopes—and ensuring that our new language construct did not interfere with these exploratory interactions. We gave Pagebreaks to notebook programmers to use in their own work, and they reported Pagebreaks alleviated some of the issues with global variables and that they could still use their usual notebook programming interactions. Our study also revealed the many ways participants used Pagebreaks to *organize* their programs and how they used and talked about scoping constructs more broadly. We hope this work inspires others to break open language constructs, find opportunities in "lazy" programming practices, and think about environments, languages, and users as three parts of one whole.

ACKNOWLEDGMENTS

The authors would like to thank our study participants for their time and thoughtfulness, our reviewers for their suggestions and insights, and Justin Lubin and Parker Ziegler for their feedback and guidance. This work is supported in part by the National Science Foundation, under grants FW-HTF 2129008, CA-HDR 2033558, and 2243822 as well as by gifts from Google, G-Research, Adobe, and Microsoft. Chasins is a Chan Zuckerberg Biohub Investigator.

REFERENCES

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2014. *Compilers: principles, techniques, and tools* (second edition, pearson new international edition ed.). Pearson, Harlow.

[2] 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, Raleigh, NC, 25–29. https://doi.org/10.1109/vlhcc.2017.8103446

[3] Michael Brachmann, William Spoth, Oliver Kennedy, Boris Glavic, Sonia Castelo, Carlos Bautista, and Juliana Freire. 2020. Your notebook is not crumby enough, REPLace it. In *Proceedings of the 10th Conference on Innovative Data Systems*, Heiko Mueller (Ed.). Conference on Innovative Data Systems Research (CIDR), Amsterdam, Netherlands, 16.

[4] Virginia Braun and Victoria Clarke. 2006. Using thematic analysis in psychology. *Qualitative Research in Psychology* 3, 2 (Jan. 2006), 77–101. https://doi.org/10.1191/1478088706qp063oa QID: Q30476519.

[5] Virginia Braun and Victoria Clarke. 2019. Reflecting on reflexive thematic analysis. *Qualitative Research in Sport, Exercise and Health* 11, 4 (Aug. 2019), 589–597. https://doi.org/10.1080/2159676x.2019.1628806

[6] Souti Chattopadhyay, Zixuan Feng, Emily Arteaga, Audrey Au, Gonzalo Ramos, Titus Barik, and Anita Sarma. 2023. Make It Make Sense! Understanding and Facilitating Sensemaking in Computational Notebooks. https://doi.org/10.48550/arXiv.2312.11431 arXiv:2312.11431 [cs].

[7] 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 (CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3313831.3376729

[8] Frederick Choi, Sajjadur Rahman, Hannah Kim, and Dan Zhang. 2023. Towards Transparent, Reusable, and Customizable Data Science in Computational Notebooks. In *Extended Abstracts of the 2023 CHI Conference on Human Factors in Computing Systems (CHI EA '23)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/3544549.3585807

[9] Taijara Loiola De Santana, Paulo Anselmo Da Mota Silveira Neto, Eduardo Santana De Almeida, and Iftekhar Ahmed. 2024. Bug Analysis in Jupyter Notebook Projects: An Empirical Study. *ACM Trans. Softw. Eng. Methodol.* 33, 4 (April 2024), 101:1–101:34. https://doi.org/10.1145/3641539

[10] Helen Dong, Shurui Zhou, Jin L.C. Guo, and Christian Kästner. 2021. Splitting, Renaming, Removing: A Study of Common Cleaning Activities in Jupyter Notebooks. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*. IEEE, Melbourne, Australia, 114–119. https://doi.org/10.1109/ASEW52652.2021.00032 ISSN: 2151-0830.

[11] Brian E. Granger and Fernando Pérez. 2021. Jupyter: Thinking and Storytelling With Code and Data. *Computing in Science & Engineering* 23, 2 (March 2021), 7–14. https://doi.org/10.1109/MCSE.2021.3059263 Conference Name: Computing in Science & Engineering.

[12] Jesse Harden, Elizabeth Christman, Nurit Kirshenbaum, Mahdi Belcaid, Jason Leigh, and Chris North. 2023. “There is no reason anybody should be using 1D anymore”: Design and Evaluation of 2D Jupyter Notebooks. Graphics Interface, Victoria, British Columbia, Canada, 13. https://openreview.net/forum?id=Gkogn48LeI

[13] 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*. ACM, Glasgow Scotland Uk, 1–12. https://doi.org/10.1145/3290605.3300500

[14] C. A. R. Hoare. 1983. Hints on Programming Language Design. In *Programming Languages*, Ellis Horowitz (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 31–40. https://doi.org/10.1007/978-3-662-09507-2_3

[15] Joshua Horowitz and Jeffrey Heer. 2023. Engraft: An API for Live, Rich, and Composable Programming. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*. ACM, San Francisco CA USA, 1–18. https://doi.org/10.1145/3586183.3606733

[16] Jeremy Hylton. 2000. *Statically Nested Scopes.* Python Enhancement Proposal 227. Python Software Foundation. https://peps.python.org/pep-0227/

[17] Joel Jakubovic, Jonathan Edwards, and Tomas Petricek. 2023. Technical Dimensions of Programming Systems. *The Art, Science, and Engineering of Programming* 7, 3 (Feb. 2023), 13. https://doi.org/10.22152/programming-journal.org/2023/7/13

arXiv:2302.10003 [cs].

[18] DaYe Kang, Tony Ho, Nicolai Marquardt, Bilge Mutlu, and Andrea Bianchi. 2021. ToonNote: Improving Communication in Computational Notebooks Using Interactive Data Comics. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems (CHI '21)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3411764.3445434

[19] Jake Kara. 2021. *Margo: Margin Notes for Computational Notebooks.* A.L.M. Harvard University, United States – Massachusetts. https://www.proquest.com/docview/2539589152/abstract/E7A2AA87E6BE428APQ/1 ISBN: 9798738638749.

[20] Mary Beth Kery. 2021. *Designing Effective History Support for Exploratory Programming Data Work.* Ph. D. Dissertation. Carnegie Mellon University, Pittsburgh, Pennsylvania, United States.

[21] Mary Beth Kery, Amber Horvath, and Brad Myers. 2017. Variolite: Supporting Exploratory Programming by Data Scientists. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems.* ACM, Denver Colorado USA, 1265–1276. https://doi.org/10.1145/3025453.3025626

[22] Mary Beth Kery and Brad A. Myers. 2018. Interactions for Untangling Messy History in a Computational Notebook. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC).* IEEE, Lisbon, Portugal, 147–155. https://doi.org/10.1109/vlhcc.2018.8506576 ISSN: 1943-6106.

[23] 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.* ACM, Montreal QC Canada, 1–11. https://doi.org/10.1145/3173574.3173748

[24] Mary Beth Kery, Donghao Ren, Fred Hohman, Dominik Moritz, Kanit Wongsuphasawat, and Kayur Patel. 2020. mage: Fluid Moves Between Code and Graphical Work in Computational Notebooks. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology.* Association for Computing Machinery, New York, NY, USA, 140–151. https://doi.org/10.1145/3379337.3415842

[25] 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 ISSN: 1943-6106.

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

[27] Malin Källén and Tobias Wrigstad. 2021. Jupyter Notebooks on GitHub: Characteristics and Code Clones. *The Art, Science, and Engineering of Programming* 5, 3 (Feb. 2021), 15. https://doi.org/10.22152/programming-journal.org/2021/5/15 arXiv:2007.10146 [cs].

[28] Xingjun Li, Yuanxin Wang, Hong Wang, Yang Wang, and Jian Zhao. 2021. NBSearch: Semantic Search and Visual Exploration of Computational Notebooks. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems (CHI '21).* Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3411764.3445048

[29] Stephen Macke, Hongpu Gong, Doris Jung-Lin Lee, Andrew Head, Doris Xin, and Aditya Parameswaran. 2021. Fine-grained lineage for safer notebook interactions. *Proc. VLDB Endow.* 14, 6 (Feb. 2021), 1093–1101. https://doi.org/10.14778/3447689.3447712

[30] Wes McKinney. 2017. *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython.* "O'Reilly Media, Inc.", Sebastopol, CA.

[31] Jibesh Patra and Michael Pradel. 2022. Nalin: learning from runtime behavior to find name-value inconsistencies in jupyter notebooks. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22).* Association for Computing Machinery, New York, NY, USA, 1469–1481. https://doi.org/10.1145/3510003.3510144

[32] Fernando Perez and Brian E. Granger. 2007. IPython: A System for Interactive Scientific Computing. *Computing in Science & Engineering* 9, 3 (2007), 21–29. https://doi.org/10.1109/MCSE.2007.53

[33] Philip J. Guo and Margo Seltzer. 2012. BURRITO: Wrapping Your Lab Notebook in Computational Infrastructure. USENIX Association, Boston, MA, 4. https://www.usenix.org/conference/tapp12/workshop-program/presentation/guo

[34] 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 ISSN: 2574-3864.

[35] João Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. 2021. Understanding and improving the quality and reproducibility of Jupyter notebooks. *Empirical Software Engineering* 26, 4 (May 2021), 65. https://doi.org/10.1007/s10664-021-09961-9

[36] Luigi Quaranta, Fabio Calefato, and Filippo Lanubile. 2022. Eliciting Best Practices for Collaboration with Computational Notebooks. *Proc. ACM Hum.-Comput. Interact.* 6, CSCW1 (April 2022), 87:1–87:41. https://doi.org/10.1145/3512934

[37] Deepthi Raghunandan, Aayushi Roy, Shenzhi Shi, Niklas Elmqvist, and Leilani Battle. 2023. Code Code Evolution: Understanding How People Change Data Science Notebooks Over Time. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems.* ACM, Hamburg Germany, 1–12. https://doi.org/10.1145/3544548.3580997

[38] Dhivyabharathi Ramasamy, Cristina Sarasua, Alberto Bacchelli, and Abraham Bernstein. 2022. Workflow analysis of data science code in public GitHub repositories. *Empirical Software Engineering* 28, 1 (Nov. 2022), 7. https://doi.org/10.1007/s10664-022-10229-z

[39] Bernadette M. Randles, Irene V. Pasquetto, Milena S. Golshan, and Christine L. Borgman. 2017. Using the Jupyter Notebook as a Tool for Open Science: An Empirical Study. In *2017 ACM/IEEE Joint Conference on Digital Libraries (JCDL).* ACM/IEEE, Toronto, ON, Canada, 1–2. https://doi.org/10.1109/jcdl.2017.7991618

[40] Lars Reimann and Günter Kniesel-Wünsche. 2023. An Alternative to Cells for Selective Execution of Data Science Pipelines. In *2023 IEEE/ACM 45th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER).* IEEE/ACM, Melbourne, Australia, 129–134. https://doi.org/10.1109/ICSE-NIER58687.2023.00029 ISSN: 2832-7632.

[41] 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 (ICPC '22).* Association for Computing Machinery, New York, NY, USA, 253–263. https://doi.org/10.1145/3524610.3529156

[42] Yvonne Rogers and Paul Marshall. 2017. *Research in the Wild.* Springer International Publishing, Cham. https://doi.org/10.1007/978-3-031-02220-3

[43] Adam Rule, Amanda Birmingham, Cristal Zuniga, Ilkay Altintas, Shih-Cheng Huang, Rob Knight, Niema Moshiri, Mai H. Nguyen, Sara Brin Rosenthal, Fernando Pérez, and Peter W. Rose. 2019. Ten simple rules for writing and sharing computational analyses in Jupyter Notebooks. *PLOS Computational Biology* 15, 7 (July 2019), e1007007. https://doi.org/10.1371/journal.pcbi.1007007 Publisher: Public Library of Science.

[44] Adam Rule, Ian Drosos, Aurélien Tabard, and James D. Hollan. 2018. Aiding Collaborative Reuse of Computational Notebooks with Annotated Cell Folding. *Proceedings of the ACM on Human-Computer Interaction* 2, CSCW (Nov. 2018), 150:1–150:12. https://doi.org/10.1145/3274419

[45] 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.* ACM, Montreal QC Canada, 1–12. https://doi.org/10.1145/3173574.3173606

[46] Saman Amarasinghe, Adam Chlipala, Srini Devadas, Michael Ernst, Max Goldman, John Guttag, Daniel Jackson, Rob Miller, Martin Rinard, and Armando Solar-Lezama. 2014. Reading 3: Testing & Code Review. https://www.mit.edu/~6.005/fa14/classes/03-testing-and-code-review/#reading_3_testing__code_review

[47] Davide Sarra. 2024. Jupyter Spaces. https://github.com/davidesarra/jupyter_spaces original-date: 2018-04-10T18:07:16Z.

[48] Shreya Shankar, Stephen Macke, Sarah Chasins, Andrew Head, and Aditya Parameswaran. 2022. Bolt-on, Compact, and Rapid Program Slicing for Notebooks. *Proc. VLDB Endow.* 15, 13 (Sept. 2022), 4038–4047. https://doi.org/10.14778/3565838.3565855

[49] Jeremy Singer. 2020. Notes on notebooks: is Jupyter the bringer of jollity?. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2020).* Association for Computing Machinery, New York, NY, USA, 180–186. https://doi.org/10.1145/3426428.3426924

[50] 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 (ICSE-SEIP '22).* Association for Computing Machinery, New York, NY, USA, 13–22. https://doi.org/10.1145/3510457.3513032

[51] Krishna Subramanian, Ilya Zubarev, Simon Völker, and Jan Borchers. 2019. Supporting Data Workers To Perform Exploratory Programming. In *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems (CHI EA '19).* Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/3290607.3313027

[52] April Yi Wang, Anant Mittal, Christopher Brooks, and Steve Oney. 2019. How Data Scientists Use Computational Notebooks for Real-Time Collaboration. *Proc. ACM Hum.-Comput. Interact.* 3, CSCW (Nov. 2019), 39:1–39:30. https://doi.org/10.1145/3359141

[53] April Yi Wang, Dakuo Wang, Jaimie Drozdal, Xuye Liu, Soya Park, Steve Oney, and Christopher Brooks. 2021. What Makes a Well-Documented Notebook? A Case Study of Data Scientists' Documentation Practices in Kaggle. In *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems (CHI EA '21).* Association for Computing Machinery, New York, NY, USA, 1–7. https://doi.org/10.1145/3411763.3451617

[54] April Yi Wang, Zihan Wu, Christopher Brooks, and Steve Oney. 2020. Callisto: Capturing the "Why" by Connecting Conversations with Computational Narratives. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems.* Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3313831.3376740

[55] 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*. ACM, Virtual Event Australia, 138–149. https://doi.org/10.1145/3324884.3416585

[56] 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

[57] 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 (CHI '21)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3411764.3445527

[58] John Wenskovitch, Jian Zhao, Scott Carter, Matthew Cooper, and Chris North. 2019. Albireo: An Interactive Tool for Visually Summarizing Computational Notebook Structure. In *2019 IEEE Visualization in Data Science (VDS)*. IEEE, Vancouver, BC, Canada, 1–10. https://doi.org/10.1109/VDS48975.2019.8973385

[59] Yifan Wu, Joseph M. Hellerstein, and Arvind Satyanarayan. 2020. B2: Bridging Code and Interactive Visualization in Computational Notebooks. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. Association for Computing Machinery, New York, NY, USA, 152–165. https://doi.org/10.1145/3379337.3415851

[60] 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*. ACM, Rochester MI USA, 1–12. https://doi.org/10.1145/3551349.3556918