


# Output-Oriented Notebooks: Facilitating Rapid Exploration while Supporting Reproducibility

DAVID KOOP and COLIN J. BROWN, Northern Illinois University, USA

```
[init]: puppies = ['Australian Shepherd', 'Beagle', 'Bernese Mountain', 'Jack Russell']
puppies: ► ['Australian Shepherd', 'Beagle', 'Bernese Mountain', 'Jack Russell']

[info]: puppies = get_info(puppies$updated)
puppies: ► [{'breed': 'Australian Shepherd', 'height': (51, 58), 'weight': (16, 32), 'photo': <IPython.core.display.Image>},
..., {'breed': 'Golden Retriever', 'height': (51, 61), 'weight': (25, 34), 'photo': <IPython.core.display.Image>}]

[7ea7aa35]: photos = [d['photo'] for d in puppies$info]
photos: 
► [ , , , , ]

[updated]: puppies = puppies$init + ['Golden Retriever']
puppies: ► ['Australian Shepherd', 'Beagle', 'Bernese Mountain', 'Jack Russell', 'Golden Retriever']
```

Fig. 1. Output-Oriented Notebooks emphasize and label outputs, eliminate ambiguous variable references using named cells that scope variables, and promote succinct, expandable outputs that serve as landmarks connecting computational paths.

Computational notebooks have a number of features, including support for rapid exploration and editing, that have led to their adoption in a variety of fields. Notebooks combine text, code, and rich, often interactive, outputs in a single medium. They allow users to split code into cells, each of which can have a corresponding output, and these outputs often serve as the inspiration for the next step in an analysis. In most programming environments, variable names help users connect outputs from one expression to their use in others, but in notebook cells, the connection between the displayed output and the underlying variable can be unclear. We use named outputs and a dataflow structure to clearly link cells, improving comprehension and structuring the computation so that it can be more reproducible. This supports reuse by reducing ambiguity and mental strain when exploring and reviewing notebooks.

CCS Concepts: • **Human-centered computing** → **Interactive systems and tools**.

Additional Key Words and Phrases: Computational notebooks, output-driven analysis, cell dependencies, notebook visualization

## ACM Reference Format:

David Koop and Colin J. Brown. 2024. Output-Oriented Notebooks: Facilitating Rapid Exploration while Supporting Reproducibility. In *ACM CHI Workshop on Human-Notebook Interactions, May 11, 2024, Honolulu, HI*. ACM, New York, NY, USA, 13 pages. <https://doi.org/XXXXXXX.XXXXXXX>

## 1 INTRODUCTION

Computational notebooks have become important components of computational science, providing a scratchpad for efficient, persisted analysis. Notebooks weave text, code, and outputs in a single document, with outputs immediately

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

Manuscript submitted to ACM

```

[6]: puppies = ['Australian Shepherd', 'Beagle', 'Bernese Mountain', 'Jack Russell']
[8]: puppies = get_info(puppies)
[9]: photos = [d['photo'] for d in puppies]
      photos
[9]: [<IPython.core.display.Image object>,
      <IPython.core.display.Image object>,
      <IPython.core.display.Image object>,
      <IPython.core.display.Image object>,
      <IPython.core.display.Image object>]
[7]: puppies = puppies + ['Golden Retriever']

```

Fig. 2. The same notebook as shown in Fig. 1, but using Jupyter Notebook with default settings. Here, the execution order is indicated by the cell numbers, and references to variables are ambiguous. Outputs are only shown when requested, are unlabeled, lack rich outputs, and limit user control.

following the code that generates them. While notebooks can be used for quick analyses, they also provide important documentation of past investigations, helping authors and collaborators see exactly how an analysis was completed. Furthermore, notebooks are also being used in published results (e.g., [17, 31]). As notebooks are used for reflection and inspection, it becomes important that users can understand their past work and share it with others. Thus, we wish to minimize barriers to understanding and reuse.

In current notebooks, the freedom to write code often overlaps with the burden of later understanding it. While breaking code into smaller snippets interspersed with outputs and text makes finding components of a computation easier, the challenge is connecting the pieces together. All variables in a notebook are, by default, global, and thus a variable from one cell can be referenced by any expression in another cell. This extends to position; a reference in a cell above the cell where a variable is defined is allowed, though articles on best practices advise against this [24, 27]. These variables and their references implicitly define dependencies between the cells, but those references may be ambiguous [15]. The way a user defines and positions cells can be orthogonal to the order they are executed. This has its advantages as a reference to an earlier definition can be replaced with a new dataset or filtered version without deleting the initial reference in case the user wishes to return to it. That said, a user can be left wondering which definition a particular expression refers to, and a collaborator who opens the notebook later can be even more confused.

Adding more structure to the cells, then, can provide better experiences for users and readers alike. Our approach is to have users explicitly indicate those variables they will use elsewhere by listing or assigning them in the last line of code. Jupyter notebooks already consider expressions on the last line to be the outputs of the cell, and we restrict the use of variables across cells to only those defined as outputs. This nudges users to expose values they wish to use again. Because too many variable names can clutter human memory [29], reusing a variable name for a transformation or refinement of a value that still semantically represents the same entity should be encouraged. However, this may leave multiple cells with outputs of the same name. To address this ambiguity, we automatically track and persist the cells each reference is associated with and allow users to designate a particular “version” of the variable when a previous version is desired. This can then be augmented with new operations—for example, one that reruns cells as variables were originally referenced and another that uses the current variable values. Because we always persist the exact references at execution, we can deterministically reproduce the computations as they were originally rerun.

While structure helps users recompute and disambiguate references, navigating the notebooks remains a challenge. Specifically, cell positions need not be in executed order, and references can span the entire notebook. We argue that

outputs should be landmarks for users as they read notebooks. We know that users examine outputs in order to decide the next steps in a task like data exploration [19], and the convenience of seeing outputs inline with the code can help users locate regions of their notebooks. However, current notebook frameworks do not connect the displayed outputs with the underlying variables they represent. We propose to label outputs by the variables they depict and also allow users to tag the cells themselves. Then, as a user scrolls the notebook, they can more quickly locate content without manual organization.

At the same time, the outputs themselves should be compact, expandable, and interactive. Scrolling through tens of lines of output is tedious and such long displays do not allow the deep inspection often only done after the initial computation. When users want specific representations, they often must resort to code in order to obtain them. For example, a list of data frames is a mess in Jupyter, but an individual frame has a rich display. At the same time, the display of a full-resolution image is often unnecessary until a user wishes to examine it. While there are numerous elegant interactive widgets in notebooks, the means to control them has been missing. Hiding them completely eliminates clutter, but it also removes important landmarks. We propose mechanisms to grant the user control over outputs and collections of outputs while providing sensible default behaviors.

This paper focuses on improving exploration in notebooks while facilitating reuse through two core contributions:

- A dataflow framework that unambiguously defines connections between notebook cells while allowing users to reference outputs with memorable identifiers and tags.
- An elevation of outputs to be landmarks that guide exploration and allow less cumbersome inspection.

We believe that these contributions combine to improve overall notebook use. When outputs are richer, they serve as better landmarks for users following dependencies. When dependencies are well-defined, the notebook’s execution is more predictable. These contributions are made available as open-source in the dataflow notebook extension and kernel<sup>1</sup>, and the ipycollections library<sup>2</sup>.

## 2 DEFINITIONS

A *computational notebook* is a sequence of code and text blocks called cells (see Figs. 1 and 2). Generally, a user executes individual code cells one at a time, going back to edit and re-execute cells as desired. This is in contrast to scripts where all code is executed at once. In addition, new cells may be later inserted between existing, already-computed cells, so it is possible that the semantics of a variable change. There exist a variety of different computational notebook environments [2, 7, 14, 21, 22, 33, 34, 39], all of which use text and code cells with computational results shown inline. Generally, these environments mimic paper notebooks that document a scientist’s work and include text, computations, and visualizations. When archived, they provide a record of analysis and any discoveries. Importantly, notebooks persist *both* input code and output results, providing a natural fit for testing reproducibility versus source code, where outputs are generally stored in separate documents. Given the variety of users and goals, systems have added specific features. For example, Observable notebooks [22] adopt a reactive approach where a change in any cell’s result propagates to any dependent cells and also include methods to trigger automatic updates for animations. Collab [7] utilizes the cloud to offer notebooks with real-time collaboration.

A notebook is **reproducible** if another user (or the same user in a different session) with the same environment and access to data can generate the same results as stored in the output cells. One challenge in reproducing a notebook’s

<sup>1</sup><https://github.com/dataflownb/dfnotebook-extension>

<sup>2</sup><https://github.com/dataflownb/ipycollections-extension>

results are the dependencies between cells, especially if the notebook has been executed in a non-linear manner. Often, variable references can be used to help determine some structure. If one cell defines a variable  $x$  and no other cell defines that variable, any cell that references  $x$  needs to be executed after the defining cell. The second cell is *dependent* on the first. When a variable is assigned to a value more than once and in different cells, we have a potential *ambiguity* for any other references because those references could be to either of the assignments. Fig. 2 shows an example where this occurs with the `puppies` variable; does `get_info` refer to the original initialization or the list with "Golden Retriever" added? We can extend the notion of dependency via transitivity; given a cell  $c$ , any cell that must be executed before  $c$  is an *upstream* dependency, and any cell that requires  $c$  to be executed first is a *downstream* dependency. *Immediate* dependencies are directly related. When a cell is modified and executed, every downstream dependency is *stale* until they are re-executed.

*Jupyter*. In this paper, we will focus on one of the most popular computational notebook environments: Jupyter notebooks written in Python [14]. In that environment, cells are identified with a numeric identifier that is incrementally assigned when the cell is executed. However, executing a cell after it has been executed once overwrites the identifier with the latest count. When the notebook is reopened in a new session, the counter resets. Jupyter Notebooks keep history via the IPython system, but it is not persisted with the notebook so while users can review history in the current session, they cannot easily see previous session history.

### 3 RELATED WORK

#### 3.1 Issues in Notebook Environments

There have been a number of papers and talks that have highlighted issues with current notebook environments (e.g. [5, 8]) spanning many facets including environment issues [23], versioning [13], and reuse [37]. A key issue with editing notebooks is when cells are changed without feedback about the impact on other cells in the notebook (i.e., invalidating those cells or changing their results). This contrasts with other programming mediums like scripts, where the entire code is always rerun, meaning users see the impact of their changes. For notebooks, this allows more rapid iteration on ideas (e.g., fixing a bug in one part of an analysis without rerunning earlier preparation steps), but it can lead to states where the user is unclear about how the results are generated [23]. Even in real-time collaboration settings where users are working on a shared notebook, users require some level of coordination, and understanding other users' contributions can also be complicated by the non-linear structure of notebook work [36].

There have also been studies into the reproducibility and reuse of notebooks. In general, these studies highlight low rates of reproducing the original results [23, 37]. Being able to re-execute a notebook that was created by a different person, at a different time, or on a different machine, often requires some work to ensure that environments are configured correctly and any associated data are available [37]. Note that this is a complimentary issue to the interfaces in the notebook themselves. Our work impacts the reuse of notebooks even when these other factors remain fixed.

Notebooks use variables that are shared between cells so that a variable defined in one cell is accessible in another. This places some mental load on the user to remember what each variable represents and currently stores. In work on program comprehension, Jones [10] ran experiments to see when users remembered variable values when processing code or would have had to go back to look them up. He found that in many cases, people "would refer back" to the original assignment because they were unsure of the values. This has a parallel in notebooks but is arguably more complex. Instead of looking up an assignment statement, users will often create a new cell with the express purpose of seeing what the current value of a variable is. They also often don't use variables again [38]. An interface where

outputs are tagged with variable names simplifies this process. There has also been work on how identifier naming and style conventions affect code understanding. While memorable names are useful, similar names can be problematic [1]. Thus, a strategy where users introduce more and more variable names is likely to make it harder for users to recall the value associated with a variable. Another complication is dependence clusters or dependence "pollution," where many variables reference each other, meaning a change to the code can have a wide impact [3].

While some notebooks are treated like scratchpads [13], there is evidence that notebooks do evolve from more exploratory to explanatory [26]. In such cases, the ability to understand what has happened in the past is important to the notebook creators [13], and when collaboration is involved, this understanding is more important because users lose track of variable names [36].

Notebook outputs are important in helping users identify the next step in their work, but they can also clutter a user's screen due to the space they take up [13] and limit comparison due to the linear layout of the notebook [40]. At the same time, certain outputs are not helpful to users because they lack rich output representations. While notebooks usually support extensions to render custom displays of outputs, a variable that lacks a meaningful display is difficult to examine. In IPython in Jupyter, for example, collections like lists and dictionaries are rendered textually regardless of whether the items they hold have rich displays. Thus, a user must access individual elements to display them (e.g., `mylist[2]`). In addition, the amount of output displayed is often limited, either by the notebook front-end or the kernel. Some editors (e.g., Visual Studio Code) provide links for users to view full output when it is truncated, but while these options are customizable, most users rarely modify the defaults or their own settings during their work.

### 3.2 Improving Notebooks

*Navigation and Cleanup.* Work on improving the usability of notebooks has seen significant results that focus on helping users better understand and navigate the existing notebook structure. For messy notebooks, techniques have been developed to fold blocks of cells [28] or help users gather only those cells germane to a particular artifact [9]. It can also be important for users to understand how their actions affect the evolution of a notebook, and interfaces that present such information augments users' memories [11, 12].

*Execution Model.* One approach to addressing issues with out-of-order execution or stale references is to modify the execution semantics or the structure of the notebook. Notebook, for example, enforces a linear order for cells [30], while Observable [22] and ReactivePy [35] embrace a reactive execution where any cell change triggers all dependent cells to also execute. This reactive style works well for code that executes quickly but can be problematic for larger workflows where users might prefer to make multiple edits before triggering a re-execution. The original Dataflow Notebook changed execution semantics to provide dataflow execution where upstream dependencies are re-executed, but used meaningless identifiers to refer to variables [16].

*Alerting Users.* An alternative to modifying the way notebooks are structured or executed is to help users understand the effects their actions have on the rest of the notebook. Here, JupyterLab has introduced a change indicator for cells that adds a yellow color for cells whose content has been edited but not executed. NBSafety took this further by using static and dynamic analysis to add staleness indicators to cells to indicate when a change to another cell has made the execution of a cell unsafe, also indicating when cells are safe to re-execute, meaning dependencies are not stale [20]. Verdant allowed users to track versions of cells as they change while also providing interfaces to help users navigate these histories and extract the relevant cells for any output [12].

*Leveraging Interactive Outputs.* An important trend in augmenting notebooks is using rich output capabilities to drive exploration. Intermediate outputs lead to insights. Users often use short snippets of code because they wish to see outputs in order to make decisions [13], and breaking code snippets using intermediate variables can help understanding [4]. More generally, data dependencies are common in data analysis, where an output from one step serves as an input for the next [19]. Instead of writing code to clean data, Wrex allows users to generate code from changes made directly to data frames [6]. Similarly, in B2, interactive output visualizations may be manipulated directly by users in a separate panel, and those actions are translated back into code [40]. Importantly, the syntax for generating these interactive outputs in B2 (`<variable>.vis()`) highlights a standard path for users to derive output visualizations as a display of the underlying data rather than a procedure to create a visualization. Lux takes this further by having always-on visualizations for data so that users need not worry about potentially cumbersome steps to generating visualizations [18]. All of these approaches not only highlight the importance of interactive outputs in the notebook environment, but they also demonstrate the need for connections between the in-memory variables and the displayed outputs.

## 4 OUTPUT-ORIENTED NOTEBOOKS


Our work addresses two key challenges in the standard Jupyter notebook representation: (1) a reliance on code to inspect state and improve output display and (2) problems in understanding how cells are related. We guide users to write cells that list all outputs in the last line in a similar manner as one would write a return statement from a function. Then, for each cell, we automatically display *every* output separately, utilizing the rich output capabilities of Jupyter, with its name in the left margin. In addition, we enhance Jupyter’s base output capabilities for structures like lists and dictionaries to be interactive, allowing users to drill down and minimizing the need for code. At the same time, we enforce a dataflow structure on the cells through the named outputs: any reference to an output that is defined in another cell creates a dependency between those cells. To deal with ambiguities when identifiers with the same name are output in different cells, we introduce a scoping syntax that clearly identifies the referenced cell along with tags that allow users to add meaning to outputs without editing code. Finally, we introduce conventions that minimize changes from existing workflows by inferring unspecified dependencies automatically.

### 4.1 Improving Core Output Representations

Our goal in extending current notebook technologies is to help users better understand the work encapsulated in notebooks so they can more efficiently edit, reuse, and collaborate. Determining the relationships between the cells—dependencies—is an important lower-level task that helps users understand how any change will affect the rest of the notebook and its outputs. Navigating these paths requires linking variable references in code with the content of those variables—the output. Therefore, making outputs and their *names* more salient will help users classify and recall the underlying data. For simple values like numbers or strings, a textual display works well, but for more complex values like collections or datasets, we propose to succinctly summarize content while allowing the user to drill down for greater detail.

Output-Oriented Notebooks lean on existing rich output renderers but integrate them into more compact representations that allow users to both drill down by expanding collections and collapse unimportant components. The Jupyter framework provides a wide array of renderers for a variety of data types, and libraries can provide renderers for their objects as well (e.g., the pandas data analysis library provides an HTML table renderer). However, any Python collection is rendered as text, often making the values difficult to comprehend or uninformative (see Fig. 2). Our approach is

```

313 puppies: ▼ [ # len=5
314           0: ▼ { # len=4
315               'breed': 'Australian Shepherd',
316               'height': ► (51, 58),
317               'weight': ► (16, 32),
318               'photo': ▼
319                   
320               },
321           1: ► {'breed': 'Beagle', 'height': (33, 41)
322           2: ► {'breed': 'Bernese Mountain', 'height'
323           3: ► {'breed': 'Jack Russell', 'height': (2
324           4: ► {'breed': 'Golden Retriever', 'height'
325           },
326           ],
327       ]
328
329

```

Fig. 3. A more compact representation of output uses less screen real estate, but also allow users to drill down and expand outputs. Instead of displaying all of output at full-resolution, a user can choose to view specific data items in more detail.

to extend Python's textual rendering of data structures so that individual items are rendered as rich outputs using the existing renderers. Thus, a list of images is rendered as a comma-separated, bracketed list of thumbnails, and a dictionary with dataframe values renders those values as tables, etc. See Fig. 3 for an example.

Because the layout of all these potentially disparate values could quickly clutter the notebook, we choose to constrain their output, by eliding collections and compacting outputs to smaller dimensions or textual representations (see Fig. 1), but also allowing each output type to provide a compact representation. For example, a line plot could be compactly rendered as a sparkline. When a user wishes to view the full output, they can use a straightforward expand action that allows a particular item to fill the space (e.g., the image in Fig. 3). There is potential that too much compaction will force users to tediously expand and collapse items, so high-quality summaries are important.

Note that output also requires a name to link it to other code, and that can require a shift in what programmers compute. Generally, there are lines of code that do computations like processing data, but in order to understand that data, there are often lines of code that generate views of the data specifically for display. Thus, a visualization is often created using lines of code that map and transform the data. Those steps can be a detour to help the user understand the data rather than a transformation that will be utilized in future computations. While the expressiveness of code to generate visualizations is important, the display may only be indirectly connected to the variable that holds the entirety of the data. There have been some efforts to suggest visualizations for any output [18] or more clearly delineate that code which is only used to create the visualizations [40]. Some opportunities may exist to separate cells that generate visualizations or connect an output to the parent variable. For example, a data frame `df` for which the first ten rows are shown via `df.head(10)`, may be best labeled as `df`.

## 4.2 Deterministic Dependencies Define Execution

By elevating outputs, users can gain a greater understanding of the computations being performed, but the problem of ambiguous dependencies makes it difficult to connect the improved outputs and to their use in computations. When variables are assigned throughout code, the dependencies between those variables can be difficult to follow. In notebooks,



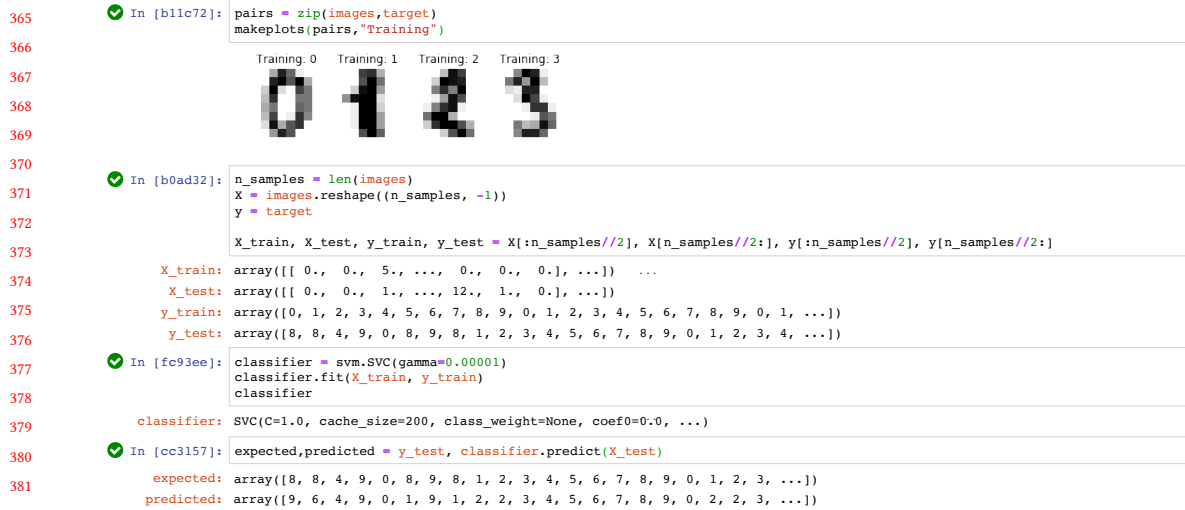


Fig. 4. Object-Oriented Notebooks in Jupyter Notebook. Cells can have multiple named outputs, and referencing an output in another cell creates a dependency between the two cells (references are shown in red).

this is compounded by the fact that the dependencies span cells. Because cells are executed individually and can be inconsistent with each other, knowing which cells to execute in order to ensure that the variables are properly defined is difficult. When references are ambiguous, this is not possible.

Output-Oriented Notebooks make important changes to both help users locate variable definitions and ensure that dependencies are not ambiguous. First, each cell is assigned a unique, persistent identifier so that any re-execution does not break references to cells. Second, while we could let any variable definition be exposed as a cell output, we build on the existing convention that cell outputs are listed as such in the last line of the cell. This is consistent with IPython's display rule that only the last expression should be displayed. In addition, through simultaneous assignment, cells may have *multiple* named outputs (see Fig. 4). Third, the cell is wrapped in a closure so that any local variables do not leak into other cells; it is not possible to refer to variables local to the cell in other cells. However, references to the exposed variables are now also references to intermediate outputs; instead of storing results in global variables that may or may not be shown, the data passed between cells is now always visible. Finally, we allow users to reuse variable names to mitigate variable recall issues and provide mechanisms to unambiguously refer to outputs with the same name.

Instead of using the mutable Jupyter cell numbers, dataflow notebooks use a persistent identifier that Jupyter assigns to a cell when it is created. This identifier is unique and does not change when the cell is edited or executed. Thus, one can fix bugs or update a computation without worrying about any references to that cell becoming stale, as happens with traditional notebooks. These identifiers also persist across sessions. When a user executes a cell that references an existing output, we can automatically *rewrite* the code to append this persistent id (e.g. `puppies$a73bd0`). While the unique cell identifiers are useful to track dependencies, the UUID strings are not helpful for users, so we allow users to add a meaningful *tag* to the cell to identify it. These tags are useful to disambiguate references to outputs that share the same name.

If we restrict output names to be unique across the entire notebook, this forces the user to remember more variable names that will likely have little meaning. Instead, we allow variables with the same name to be output in different



cells and use cell identifiers and tags to connect an output to a particular cell. Fig. 2 shows an example where puppies is defined multiple times, and without spending some time examining the code or trying different execution orders, we may have trouble obtaining the photos of all five puppies. Here, the reuse of the variable name is reasonable, but we have no tags to identify or follow the dependencies. Fig. 1 shows that by appending the cell tag as a suffix, we can disambiguate the references. To accomplish this, we modify Python syntax by using the \$ delimiter to append the cell tag (e.g. puppies\$info). While cell tags are more meaningful to the user, the notebook also stores a reference with the cell's persistent identifier (puppies\$a73bd0) to ensure that even if a user changes the tag, the dependency is preserved.

We do not require users to use cell identifiers unless they wish to reference an "older version" and resist appending cell identifiers unless the reference is ambiguous. By default, an untagged variable references the output that was generated most recently, which is often the one a user recalls and wishes to reference. Also, only when two cells both have an output with the same name do we need to show the identifier of the cell to disambiguate the references. However, the notebook metadata will always persist the references because if another version of a variable is created, we need to disambiguate all existing references, but the user does not need to see them until the ambiguity exists. While not implemented, there are other opportunities to facilitate other interactions with these grounded variable names, including adding methods to re-execute a cell with variable references updated to the most recent versions of those variables.

All of these changes allow the system to build a directed graph of dependencies based on the non-ambiguous references between cells. By disallowing cycles in the graph, we can allow the user to execute any cell, and the system will recursively evaluate upstream cells to ensure that the execution is consistent. Thus, for a notebook where every cell is deterministic and has been executed (to define dependencies), we know that another user executing any cell in the notebook will receive the same result. In this context, dataflow notebooks are reproducible. This does assume that cell executions are side-effect free, and while closures help, if there is any global memory that can be accessed between cells, we can still face repeatability issues. However, the same issues exist in the standard Jupyter environment, and we have reduces the difficulty of reproducing notebooks.

## 5 USAGE SCENARIO

Consider a scenario where a data analyst is exploring information about the top 100 soccer players. Here, the data is initially loaded from a JSON, which is a hierarchical format, and each record is stored as a associative array. The information can be printed but may quickly consume the notebook screen space. Using the compact, interactive outputs in the output-oriented notebook (OON), the analyst can expand and examine one or two records to understand the attributes that are available for analysis. This allows them to filter attributes that are not relevant before loading it into a dataframe. Without the ability to interactively expand the hierarchy, this would involve a number of programmatic calls to distinguish between JSON arrays and objects and identify the property names.

After loading and identifying relevant attributes, the analyst converts the data to a dataframe and begins to clean and transform it. A common convention is to refer to a dataframe with the identifier df and perform data cleaning operations and transformations by mutating the dataframe in place or reassigning the df identifier to the updated version. In the classic notebook (Figure 5a), one problem with such mutations is that one cannot return to the state of the dataframe before a particular transform without reexecuting the analysis and running cells until reaching the desired point. Note that using a series of identifiers df1, df2, . . . to refer to the identifiers requires the analyst must still attach meaning to each identifier and remember them. With dataflow notebooks, they can retain the df identifier for all transformations and let the system keep track of the different versions of the dataframe and how they

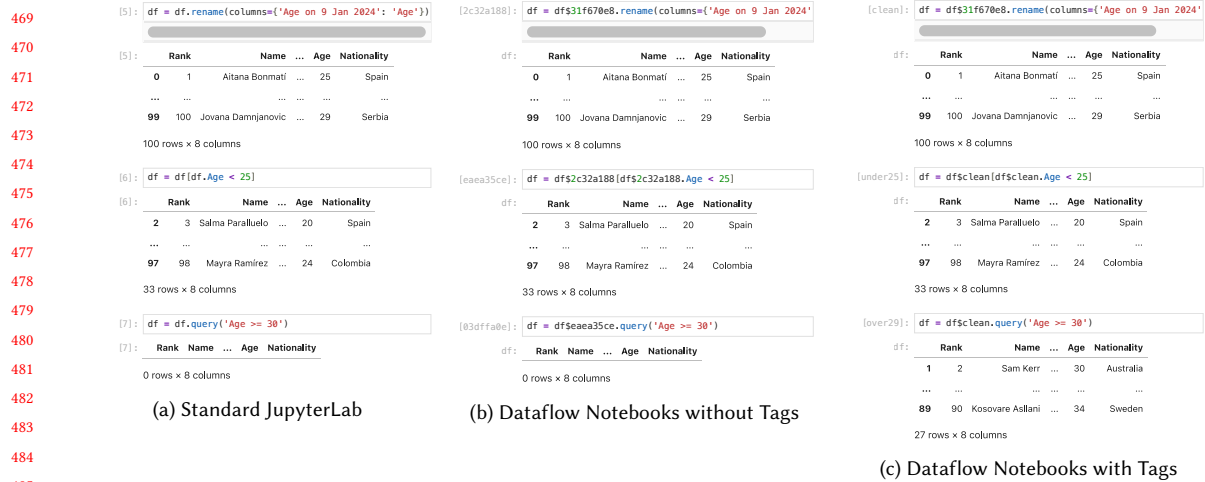


Fig. 5. In JupyterLab, variable reuse can lead to confusing outcomes that require extra computation to undo (a). While dataflow notebooks clarify the dependencies between cells, they require users to remember meaningless identifiers (b). This is remedied through the use of cell tags that allow users to differentiate named outputs, even after their initial execution (c).

are related (Figure 5b). Note that the random hexadecimal identifiers are added *by the system* to map to the most recent output with the specified name.

Suppose the analyst wishes to understand the characteristics of the top players by age. They might filter the dataframe to examine only those players 30 and older, and by convention reassign the output to the same `df`. After some investigation, they decide to examine the younger players, but realize they need to return to the dataframe before the filtering to older players occurred. With OONs, this can be a *visual search* on the *outputs* with no need to read the code. When they find the output they wish to access, they might note the hexadecimal identifier and copy it to directly reference the output (`df$2c32a188`), but since they plan to reference this output again, they instead choose to *tag* this cell as `clean`. This allows them to create a new cell with code that references `df$clean` which the system will tie to the cell currently identified by that tag (Figure 5c).

At a later point, the analyst might realize that more data cleaning needs to be done (e.g. the ages were computed with an incorrect date), and they may add new steps off of the cell previously identified with the `clean` tag, and then *move* the tag to the updated output. By default, existing cells retain references to the cells when they were originally executed, but users can choose to re-execute them to use the updated, *clean* dataframe. At all points, the dataflow notebook keeps track of the dependency graph induced by the references in one cell to the named outputs from other cells. This allows the analyst to execute any cell and know that all upstream code changes will be incorporated in the updated output.

## 6 DISCUSSION

One critique of work improving the reproducibility of notebooks has been that the majority of notebooks are used for scratchwork, and thus, adding extra constraints to notebooks to ensure reproducibility may hinder the types of exploration that users like to use notebooks for. While notebooks can also be used for publishing work for readers to interactively explore (e.g., [17, 25]), it is common for users to write such notebooks after they understand a problem, which is useful as they refine ideas and presentation. This is akin to drafting a paper and then performing a significant

rewrite to better present the material. We agree that such rewrites can be useful, but we argue that the extra constraints are not significant changes to how notebooks are currently used and provide many benefits. Currently, when users wish to examine a variable, most will write a line of code—sometimes a separate cell—to output that variable. The change to expose only specific variables as outputs is, therefore, unlikely to be a burden to users.

We believe there is an opportunity to take advantage of more interactive nature of outputs and lazily transfer data from the kernel. While we currently pass back complete outputs modulo the limits IPython imposes, there is no need to do so until a user requests it. For example, a list of hundreds of data frames need not be all transferred until the user acts to inspect them, and then only a subset needs to be transferred. The ability to access and render information on-demand can be useful to allow a user to see output faster, in addition to minimizing the complexity of seeing all of it at once. At the same time, the ability to indicate the need to see more information allows progressive refinement of the output. For example, given a data frame with hundreds of thousands of rows and tens of columns, the display might show a subset of the rows and columns but allow the user to pull in more of each on-demand. With libraries that support lazy evaluation (i.e., only evaluating a transformation on the first few rows), this can also improve the efficiency of obtaining the results.

One complication introduced by making outputs more interactive is that the data is transformed to support hierarchical exploration and thus needs to be rendered in a Web context. To publish notebooks to other mediums (e.g., transforming to a LaTeX document), the mechanisms fall back to other non-encoded outputs. Even if we could output in a more what-you-see-is-what-you-get manner, the notebooks may lose information due to the compressed state of the output. If a viewer wants to see more outputs or more detailed output, there is no kernel running to allow such inspection. The viewer would need to recompute the output, ensuring their environment is set up correctly and gaining access to the input data. Thus, storing a significant amount of output as a historical record can be useful even if it is not all immediately visible.

## 7 CONCLUSION & FUTURE DIRECTIONS

Output-Oriented Notebooks modify the ways that users build, execute, navigate, and read notebooks by focusing on the displaying of intermediate outputs that can be referenced by expressions in new cells. We believe these changes will aid users in understanding the dependencies between cells while simplifying execution. This provides a more structured manner for notebooks to evolve, but it still lacks the complete history of exploration, something versions of the notebook would preserve [26]. While solutions exist to capture and revisit this information [11], it may be possible to better compare and integrate past revisions with dataflow notebooks because of the structure of the notebook. In addition, improvements to outputs can enhance differencing tools (e.g. [32]) by making changes clearer to users. Finally, we understand that users can be frustrated by changes to common patterns, so we are exploring strategies to provide backward compatibility with standard notebooks.

## REFERENCES

- [1] Hirohisa Aman, Tomoyuki Yokogawa, Sousuke Amasaki, and Minoru Kawahara. 2019. Empirical Study of Fault Introduction Focusing on the Similarity among Local Variable Names. In *7th International Workshop on Quantitative Approaches to Software Quality*. CEUR-WS, Aachen, DE, 9.
- [2] Apache Software Foundation. 2022. Apache Zeppelin. <http://zeppelin.apache.org>.
- [3] D. Binkley and M. Harman. 2005. Locating Dependence Clusters and Dependence Pollution. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*. IEEE, Budapest, Hungary, 177–186. <https://doi.org/10.1109/ICSM.2005.58>
- [4] Roei Cates, Nadav Yunik, and Dror G. Feitelson. 2021. Does Code Structure Affect Comprehension? On Using and Naming Intermediate Variables. arXiv:2103.11008 [cs]

- [5] 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*. ACM, Honolulu HI USA, 1–12. <https://doi.org/10.1145/3313831.3376729>
- [6] Ian Drosos, Titus Barik, Philip J. Guo, Robert DeLine, and Sumit Gulwani. 2020. Wrex: A Unified Programming-by-Example Interaction for Synthesizing Readable Code for Data Scientists. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. ACM, Honolulu HI USA, 1–12. <https://doi.org/10.1145/3313831.3376442>
- [7] Google. 2022. Colab. <https://colab.research.google.com>.
- [8] Joel Grus. 2018. I Don't Like Notebooks. <https://www.oreilly.com/library/view/jupytercon-new-york/9781492025818/video322524.html>.
- [9] Andrew Head, Fred Hohman, Titus Barik, Steven M. Drucker, and Robert DeLine. 2019. Managing Messes in Computational Notebooks. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. ACM, Glasgow Scotland Uk, 1–12. <https://doi.org/10.1145/3290605.3300500>
- [10] Derek Jones. 2004. Memory for a Short Sequence of Assignment Statements. *CVU* 16, 6 (2004), 1–15.
- [11] Mary Beth Kery, Bonnie E. John, Patrick O'Flaherty, Amber Horvath, and Brad A. Myers. 2019. Towards Effective Foraging by Data Scientists to Find Past Analysis Choices. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. ACM, Glasgow Scotland Uk, 1–13. <https://doi.org/10.1145/3290605.3300322>
- [12] 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, 147–155. <https://doi.org/10.1109/VLHCC.2018.8506576>
- [13] 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>
- [14] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, and Carol Willing. 2016. Jupyter Notebooks – a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, F. Loizides and B. Schmidt (Eds.). IOS Press, Amsterdam, NL, 87 – 90.
- [15] David Koop. 2021. Notebook Archaeology: Inferring Provenance from Computational Notebooks. In *Provenance and Annotation of Data and Processes*, Boris Glavic, Vanessa Braganholo, and David Koop (Eds.). LNCS, Vol. 12839. Springer International Publishing, Online, 109–126. [https://doi.org/10.1007/978-3-030-80960-7\\_7](https://doi.org/10.1007/978-3-030-80960-7_7)
- [16] 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*. USENIX Association, USA, 1.
- [17] Laser Interferometer Gravitational-Wave Observatory (LIGO). 2016. Signal Processing with GW150914 Open Data. [https://lsc.ligo.org/s/events/GW150914/GW150914\\_tutorial.html](https://lsc.ligo.org/s/events/GW150914/GW150914_tutorial.html).
- [18] Doris Jung-Lin Lee, Dixin Tang, Kunal Agarwal, Thyne Boonmark, Caitlyn Chen, Jake Kang, Ujjaini Mukhopadhyay, Jerry Song, Micah Yong, Marti A. Hearst, and Aditya G. Parameswaran. 2021. Lux: Always-on Visualization Recommendations for Exploratory Dataframe Workflows. *Proceedings of the VLDB Endowment* 15, 3 (Nov. 2021), 727–738. <https://doi.org/10.14778/3494124.3494151>
- [19] Yang Liu, Tim Althoff, and Jeffrey Heer. 2020. Paths Explored, Paths Omitted, Paths Obscured: Decision Points & Selective Reporting in End-to-End Data Analysis. <https://doi.org/10.1145/3313831.3376533> arXiv:1910.13602 [cs]
- [20] Stephen Macke, Hongpu Gong, Doris Jung-Lin Lee, Andrew Head, Doris Xin, and Aditya Parameswaran. 2021. Fine-Grained Lineage for Safer Notebook Interactions. *Proceedings of the VLDB Endowment* 14, 6 (Feb. 2021), 1093–1101. <https://doi.org/10.14778/3447689.3447712>
- [21] Stephen North, Carlos Scheidegger, Simon Urbanek, and Gordon Woodhull. 2015. Collaborative visual analysis with RCloud. In *2015 IEEE Conference on Visual Analytics Science and Technology (VAST)*. IEEE, USA, 25–32. <https://doi.org/10.1109/VAST.2015.7347627>
- [22] ObservableHQ. 2022. Observable. <https://observablehq.com>.
- [23] Joao 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>
- [24] 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 (July 2021), 65. <https://doi.org/10.1007/s10664-021-09961-9>
- [25] Min Ragan-Kelley. 2017. Binder for LIGO Tutorial Notebook. <https://github.com/minrk/ligo-binder>.
- [26] Deepthi Raghunandan, Aayushi Roy, Shenzhi Shi, Niklas Elmqvist, and Leilani Battle. 2022. Code Code Evolution: Understanding How People Change Data Science Notebooks Over Time. arXiv:2209.02851 [cs]
- [27] 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>
- [28] 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), 1–12. <https://doi.org/10.1145/3274419>
- [29] Janet Siegmund. 2016. Program Comprehension: Past, Present, and Future. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, Suita, Osaka, Japan, 13–20. <https://doi.org/10.1109/SANER.2016.35>
- [30] StichFix. 2018. Nodebook. <https://github.com/stitchfix/nodebook>.

- [31] Richard H. Styron and Eric A. Hetland. 2014. Estimated likelihood of observing a large earthquake on a continental low-angle normal fault and implications for low-angle normal fault activity. *Geophysical Research Letters* 41, 7 (2014), 2342–2350. <https://doi.org/10.1002/2014GL059335>  
Notebook version: [https://github.com/cossatot/lanf\\_earthquake\\_likelihood/blob/master/notebooks/lanf\\_manuscript\\_notebook.ipynb](https://github.com/cossatot/lanf_earthquake_likelihood/blob/master/notebooks/lanf_manuscript_notebook.ipynb).
- [32] Jupyter Development Team. 2022. nbdime: Jupyter Notebook Diff and Merge tools. <https://github.com/jupyter/nbdime>.
- [33] The Sage Developers. 2022. *SageMath, the Sage Mathematics Software System*. The Sage Developers. <https://www.sagemath.org> DOI 10.5281/zenodo.6259615.
- [34] Two Sigma Open Source. 2022. BeakerX. <http://beakerx.com>.
- [35] California Polytechnic State University. 2019. Reactivepy. <https://github.com/jupytercalpoly/reactivepy>.
- [36] April Yi Wang, Anant Mittal, Christopher Brooks, and Steve Oney. 2019. How Data Scientists Use Computational Notebooks for Real-Time Collaboration. *Proceedings of the ACM on Human-Computer Interaction* 3, CSCW (Nov. 2019), 1–30. <https://doi.org/10.1145/3359141>
- [37] 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>
- [38] 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*. ACM, Seoul South Korea, 53–56. <https://doi.org/10.1145/3377816.3381724>
- [39] Wolfram Research, Inc. 2022. Mathematica. <https://www.wolfram.com/mathematica/>.
- [40] 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*. ACM, Virtual Event USA, 152–165. <https://doi.org/10.1145/3379337.3415851>