The Future of Notebook Programming Is Fluid

Mary Beth Kery  
School of Computer Science  
Carnegie Mellon University  
Pittsburgh, PA USA  
mkery@cs.cmu.edu

Donghao Ren  
Apple Inc.  
Seattle, WA USA  
donghao@apple.com

Kanit Wongsuphasawat  
Apple Inc.  
Seattle, WA USA  
kanitw@apple.com

Fred Hohman  
College of Computing  
Georgia Institute of Technology  
Atlanta, Georgia USA  
fredhohman@gatech.edu

Kayur Patel  
Apple Inc.  
Seattle, WA USA  
kayur@apple.com

Abstract

A new kind of widget has begun appearing in the data science notebook programming community that can fluidly switch its own appearance between two representations: a graphical user interface (GUI) tool and plain textual code. Data scientists of all expertise levels routinely work in both visual GUIs (data visualizations or spreadsheets) and plain-text code (numerical, data manipulation, or machine learning libraries). These work tools have typically been separate. Here, we argue for the unique role and potential of fluid GUI/text programming to serve data work practices. We contribute a generalized method and API for robust fluid GUI/text coding in notebooks that addresses key questions in code generation and user interactions. Finally, we demonstrate the potential of our method in two notebook tool examples and a usability study with professional data science and machine learning practitioners.

Author Keywords

Data Science Programming; Machine Learning Programming; Handoff; Computational Notebooks;

CCS Concepts

- Human-centered computing → Human computer interaction (HCI);
Introduction & Background

Data scientists coordinate between different tools and tasks in an rapid iterative fashion to experiment with data [5, 1]. Common tasks include data cleaning, visualization, transformation, and modeling [5, 1]. Common tools include spreadsheets, chart authoring tools, terminals, metric dashboards, code editors, and many code libraries [6, 8, 9].

To create a workflow that another person can sensibly replicate, notebook programming has quickly become a popular choice for anyone from students to professionals experimenting with data [13]. A notebook combines cells of formatted text/image notes, executable code, and rendered results in a single interactive document [15]. In this way, a notebook operates at a higher meta-level than any single form of work or programming language. It pulls together into a single page what data scientists otherwise commonly work with across separate tools: terminal shells, scripts, temporary output windows, output files, etc. [10]. Notebook programming has been highly lauded by the scientific computing community, who say that the format makes data work much easier to share and replicate [11, 17, 13, 12].

We observe that notebook programming, with its relatively recent rise to popularity, is still actively developing as a paradigm. This can be seen within the large active online ecosystem of communities focused on data topics, where publicly shared notebooks are common [15]. Here, we focus specifically on how the humble output cell faces an expanding role. As shown in Figure 1A, an output cell displays the result of executing the code cell directly above it. In the traditional sense of interactive programming with a read-evaluate-print-loop (REPL), output is a view-only final result. Finality is important here in the notebook’s design. Consider a notebook’s state holds the current value of each variable accross the entire notebook. State is only changed by running the users’ code cells (likely for good security reasons). An output, on the other hand, is a final endpoint. It cannot go back and update state. It doesn’t have access.¹

Newer widgets built by the community for notebooks tend to clash against this constraint. They imagine a much more expansive role for output than a REPL definition provides. It is common to see output cells, augmented with community-created tools, contain sophisticated interactive visualizations², elaborate ipyWidgets for interactive input, or even spreadsheets editors³. “Output” is reappropriated as a space for fully functional graphical user interface (GUI) tools where data scientists can continue performing useful work.

To illustrate, consider two aligned scenarios:

(A) A data scientist Rey is working in a notebook to analyze census data [4]. Rey starts by previewing the datatable (Figure 1A), which shows a standard view-only table. Rey now writes code to start cleaning the datatable.

(B) Rey sees the same table, but as a fully functional spreadsheet editor (Figure 1B). Rey quickly begins directly manipulating the table to re-arrange and rename columns so that the census data is easier to read (Figure 1B).

Both (A) and (B) are completely valid forms of the same data work. However (B) gives Rey the option to pick or combine between code or spreadsheet, whichever mode is easiest to them to achieve their task. We strongly believe that repurposing output for GUI tool work is fully within the spirit and ethos of notebooks to combine different forms of

¹An exception is ipyWidgets, an influential widget library that breaks some of these rules in a carefully controlled way: It allows output widgets to change the value of specific variables pre-chosen by the user.

²Good examples are visualization platforms plot.ly or bokeh, which both have interactive notebook widgets.

³See qgrid for an example of widget that approximates a spreadsheet.
Use case with %mage API

```python
df = df.reindex(columns=column_names)

# -- generated code —
summon table df
```

<table>
<thead>
<tr>
<th>age</th>
<th>workclass</th>
<th>fnlwgt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 90</td>
<td>Private</td>
<td>77053</td>
</tr>
<tr>
<td>1 82</td>
<td>Exec-managerial</td>
<td>132870</td>
</tr>
<tr>
<td>2 66</td>
<td>Private</td>
<td>186061</td>
</tr>
</tbody>
</table>

Figure 2: In (1) the table becomes a fully interactive spreadsheet. The user drags and drops a column to reposition it. In (2) the user’s action is reflected both in an updated table rendering and in code.

Data work. GUI tools, like the spreadsheet, are essential parts of the data scientist’s toolbox. However to actually achieve a notebook that fluidly combines code with GUI work requires dealing with some fundamental challenges around notebook state and user experience.

When Rey rearranges columns of \( df \) in the spreadsheet GUI (Figure 1B), how did this affect the value of \( df \) for the rest of the notebook? Since state is protected from output, Rey could work all day in the spreadsheet without ever effecting the value of \( df \) at all. Although multiple community-created spreadsheet widgets exist, this state barrier plagues all of them to various degrees. As long as GUI tools operate apart from the rest of the notebook state, all GUI work Rey does is easily lost between sessions.

Rey’s GUI work also loses replicability. It cannot simply be re-run alongside the rest of the notebook. Some implementations circumvent these issues by also writing code alongside spreadsheet actions\(^4\). Our goal is to build off of these early examples towards a generalized method. By carefully investigating the interaction needs for generalization, we hope to enable a future where all forms of GUIs, from interactive ML tools\(^3\) to complex visualization tools\(^14\), can provide data scientists with useful work in the notebook.

To inform our goal, dual code and GUI representations of work have extensive legacy in other domains. The ability to edit content in either GUI or code form is pervasive in editors for graphics\(^5\) and web design\(^6\). This idea has also appeared in HCI research tools like Juxtapose\(^7\).

Drawn from lessons-learned in prior work, the key of our approach is for each GUI action that should affect state, it is paired with an equivalent code action. Shown in Figure 2, when Rey moves the “occupation” column in (1), the equivalent move in pandas Python code is auto-generated and run in (2). The paired code run ensures notebook state is fully updated and Rey’s actions are recorded. Rey can go ahead and edit in the GUI or the code however they wish.

Code generation can be highly complex in theory, but here we rely on a simple code templating trick, discussed below. We built a small extension for Jupyter notebooks %mage, which acts as an application programming interface (API) to allow any GUI tool like table (Figure 2) to seamlessly generate code and share state with the notebook in a scaffolded way. %mage takes care of program analysis and notebook state concerns, while a GUI tool provides its own user interactions and code templates for any state-effecting actions. We discuss the design and tradeoffs of this approach in detail. Our contributions in this paper are:

1. Discussion of %mage API and design considerations to make this approach practical to tool builders of any GUI widget for doing active work in the notebook
2. An implementation of two example GUI tools that use %mage API: table and plot
3. New kinds of selection and drag-drop interactions between GUI and code in the notebook.
4. An initial study of our approach, testing the usability of plot and table with professional data scientists.

%mage API

The %mage API works as an extension to an unmodified Jupyter Notebook\(^11\). That said, %mage has more permissions than the average extension. %mage accesses the

\(^{4}\)See [bamboolib](http://bamboolib.8080labs.com) or [qgrid](https://github.com/quantopian/qgrid) which both generate some form of code.

\(^{5}\)See graphics environment [Blender](http://www.blender.org/).

Jupyter Notebook base application object directly to analyze, write, and run code. To invoke a GUI tool, as shown in Figure 2, the user writes a magics syntax `%%summon`, then the name of the GUI tool, then any parameters for that tool. In the case of the spreadsheet, this is `%%summon table df`. Magics syntax is a special kind of meta-command in notebooks that starts with `%%`. We chose to create a magics syntax so that it would more clearly stand-out to the user that the output produced will behave differently than normal notebook output. Any tool that uses `%mage` can be "summoned" into the notebook environment, much like a library import statement. However, anyone replicating `%mage`'s approach could choose to use an alternative syntax.

Figure 3 shows the general workflow of `%mage`. Upon invocation, `%mage` finds the correct tool based on its name (table for the spreadsheet), and calculates the value of each parameter by consulting notebook state. Required parameters are set by the individual tool's creator. For instance, table requires a variable that has the appropriate type such that it can be displayed in a spreadsheet. Next, `%mage` instantiates the GUI widget with its parameter values, and renders the GUI in an HTML box in the output.

When the user makes an action, such as adding a filter on table in Figure 3, there's a question of whether this action should affect notebook state. If the designers of the table decide that the action should only affect the tool display, no API call to `%mage` is needed. Here for a filter, however, an updated value of the variable `df` is needed to show a filtered table. So, `table` makes a call to the `%mage` API to figure out what that new value `df` is.

If we zoom out from `table` to any GUI tool that might use the `%mage` API, we run into a technical challenge. How does `%mage` know what a filter is and how to compute it? We initially considered hard-coding tabular data operations into `%mage`, but then, what if the GUI is a color picker? Or an image editor? If `%mage` needs update notebook state based on a GUI action, and a GUI could do just about anything, we ultimately decided that a GUI author will need to precisely define what their actions mean. To fill this need, we next discuss our adoption of code templates.

**Tempating Actions from GUI to Code**

To translate actions from GUI into some kind of computation that can affect state, we start from the intuition that programmers today routinely grab prefabricated code snippets from various resources online, and adjust those snippets to fit their scenario [2]. Thus it may not be too burdensome for a GUI tool creator to author a code snippet that should accompany a specific GUI action. For instance, say a user drops a column from their data in the `table` tool. In Python with the pandas library, this is written as:

```python
myData = myData.drop(columns=["dogs"])
```

Of course, in an interactive tool, we won’t know which column the user is dropping until the action occurs. Their `datatable` may also not be named `myData`. Re-writing this code to turn unknown values into a template it becomes:

```python
$DF = $DF.drop(columns=["$COL"])
```

Though we use Python examples, note that templates are not limited to Python. By writing templates with different language or library bindings for the same action, a GUI tool creator can support multiple languages and libraries. `%mage` uses type-matching to ensure the correct template is used.

---

1 magics start with a token unused by the source language. So this is `%` in Python, which we use here, but may be different in other languages.

2 pandas data manipulation library https://pandas.pydata.org/
The Full Update Cycle
To pull together the entire update cycle, we return to the point in Figure 3 where the user filters their data in the table. As soon as this action occurs, the table makes an API call to %mage with its code template for filtering. Additionally, since the user selected “age” and “< 65”, the table can send this known information to %mage as well. Thus %mage receives:

\[
\text{template } \text{DF} = \text{DF}[[\text{DF}[[\text{COL}]] = \text{EXPR}]
\]
\[
\text{where } \text{COL} = \text{“age” and EXPR} = \text{“< 65”}
\]

By consulting the notebook state, %mage identifies the name of $DF$ as $df$ and thus resolves the template as:

\[
df = df[df[“age”] < 65]
\]

Now this code is ready to run, %mage inserts the new code just above the invocation %summon table df and requests the notebook to run this cell again. When the %summon code line is re-run this time, %mage does not create a new table widget. Instead it shows the existing table and passes the updated value of $df$. By displaying the updated $df$, the table is now showing a properly filtered datatable for the user, and the update cycle is complete.

User Experience Design Challenges
Having walked through a simple use case, there are many more details that come into play when we consider serious usage between code/GUI work over time. Here we highlight some of the most challenging design considerations:

Challenge: Interrupted GUI Tool Session
Imagine our user Rey is working on the variable $df$ in the table tool. Now Rey goes and manually deletes all previously auto-generated code from the cell, leaving just the filter code. When Rey now runs the cell, the question is: Does $table$ still know that $df$ is filtered (i.e. show a filter indicator on “age”)?

This scenario is the classic round trip problem. Although $table$ and code were perfectly aligned in the initial session, as soon as Rey edited the code, $table$ no longer has a reliable list of what actions occurred—since some of them may have been deleted and effectively undone. Naively, $table$ will display $df$ as if it had never seen this data before, with no filter indicator. To make an effective “round trip” would require %mage to be able to read the user’s code and translate back code into GUI action. To a limited extent, %mage can do this, by turning table’s code templates into regular expressions to locate and pass possible table actions. However, given this approach is limited, it is unclear how to ensure a smooth user experience.

Challenge: Code Clutter
In early feedback on %mage, practitioners expressed concern that GUI spitting out a line of code for each action...
would quickly pile up a mess of code. This is illustrated in Figure 4. To combat this issue, we take the approach (Figure 4B) to compose operations into a smaller set. This is most easily achieved if it is possible to compare the start state of the GUI tool with the current state. For instance, a spreadsheet only has a finite set of columns. By comparing which columns were present at the start of the session versus now, we can combine column-related actions. If a user dropped four columns, all at different points in time, that might be compactly written in a single line of code.

A key limitation of this current approach is that it places the burden on GUI tool authors to create a composed action list, which becomes complicated when actions have order dependencies. Once the GUI tool has composed a list of templates, %mage fully replaces all previously auto-generated code from this session with these new templates. Finally, %mage takes a clean initial copy of state (df before any GUI work was done on it), and runs the notebook code with that to ensure table receives the correct value of df. While the result looks much more like human-authored code (Figure 4B), composition remains a open issue.

Drag-Drop Between Multiple Cells

Although creating a fluid environment between code and GUI certainly holds challenges, it also holds interesting opportunities. One of these is the ability for data scientists to select data in a visual form, like a table or plot, and seamlessly retrieve that selection in code. An example of this is shown in Figure 5. A user selects a region of data point from a plot, and then can drag and drop their selection into code, to perform further analysis, or into a table, to view the data points in detail. This drag and drop interaction we included in our implementation for both table and plot and is included as part of the %mage API since it concerns transferring state between one or more tools.

Usability Study

To test the usability of these ideas, we asked data science and machine learning practitioners from within Apple Inc. to try out table and plot in a series of predefined data analysis tasks on a simple census dataset [4]. Nine data workers participated in the study, with an average age of 30 and gender split 3 female and 6 male. Prior experience working with data ranged from a few months to 24 years.

All participants were able to complete all analysis tasks using table and plot. However, participants reacted differently to code being live generated as they worked. Some participants were very enthusiastic: “[pandas] is a very dense language, even for filters, if you don’t remember how to write it . . . with this simple thing you’ve got the whole power of pandas.” Another participant wanted a way to hide the code altogether unless they needed it. In a post-task survey all participants “Agreed” or “Strongly Agreed” on a 5-point Likert scale to the statements “These new interactions made me more efficient on the tasks I just did” and “It is pleasant to use”. While eight participants also agreed with the question “I learned to use it quickly”, one participant who had difficulty with plot felt just “Neutral”.

Conclusions

In this work we have illustrated both an opportunity space and the interaction challenges that come with involving GUI work in notebooks. We believe that a future notebook that fluidly incorporates diverse forms of data work is well worth the continued research, both from the research and practitioner communities, to achieve this dream.

Acknowledgements

We thank all our study participants, as well as all our colleagues at Apple Inc. who have provided invaluable feedback, brainstorming, and encouragement on this work.
REFERENCES


