Notes on Notebooks: Is Jupyter the Bringer of Jollity?

Jeremy Singer
University of Glasgow
Glasgow, UK
jeremy.singer@glasgow.ac.uk

Abstract

As the interactive computational notebook becomes a more prominent code development medium, we examine advantages and disadvantages of this particular source code format. We specify the structure of a coding notebook layout. We describe complexities in notebook programming; some of these are incidental whereas others may be inherent complexities. We outline how we envisage research and development might proceed to advance the cause of notebook programming.

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

Keywords: computational notebooks, Jupyter

ACM Reference Format:
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! '20), November 18–20, 2020, Virtual, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3426428.3426924

1 Introduction

Over the past decade, we have witnessed a quiet revolution in the way that software developers interact with source code. If mainstream software development practice in the nineties and noughties was dominated by the Integrated Development Environment (IDE), then the teens and twenties appear to be the era of the Computational Notebook. The growth in popularity of notebook programming coincides largely with the democratization of software development, considering concerted advances in school-age computer science as well as increasing emphasis on end-user software engineering. It seems that novices and non-specialist developers are the principal audience for computational notebooks. One still meets some die-hard, traditionalist, software developers who have 'never heard of Jupyter' and scoff at the idea of 'coding in an HTML textbox.' However, a trivial crawl of GitHub shows around 10 million notebooks have been checked in [19] over the past decade.

Given that a large number of people are engaged in notebook programming, many of whom may have never experienced other source code development modalities, These people may not know very much about software engineering and have never seen better source code preparation systems. How can we characterize the 'state of the notebook' and consider ways to improve it? These are the key motivations underlying this paper.

Computational notebook programming has its origins in Knuth's notion of literate programming [12] in which documentation and source code are seamlessly interleaved in a single textual entity that may have multiple views or interpretations.

Software development with interactive notebooks differs significantly from standard coding practice. Notebooks reside in a browser-based coding environment with no complex installation steps required. Typical notebook frameworks feature direct integration with popular languages and libraries, avoiding the overheads of user package management and dependency resolution.

An interactive notebook provides rich inline textual and graphical commentary on the code, interwoven with the source code itself. Further, the output of source code execution is immediately visible in the same inline representation, by means of live code and interactive interpretation.

1.1 Autoethnography Disclaimer

Where did this paper come from? The research method is largely autoethnographic [2] based on three kinds of personal experience.

1. I have a range of individual interactions with notebook programming in various incarnations of the Jupyter framework over the past few years, particularly in the context of university learning and teaching.
2. I was involved in the development and deployment of a beginner-friendly hosted notebook development environment called *ErysNotes*, used for practical exercises in a massive open online course 'Getting Started with Teaching Data Science in Schools' which commenced in April 2020[1].
3. I have initiated a series of useful workshop discussions titled 'Notebook Programming Considered Harmful?'

[1] https://www.futurelearn.com/courses/teaching-data-science

Onward! '20, November 18–20, 2020, Virtual, USA

This conversation commenced with colleagues within my institution, and later spread to national level, about the relative merits of notebook programming.

Drawing on these activities, this paper encapsulates the arguments and points out some potential directions for improvement.

1.2 Contributions

This paper makes the following original contributions in the field of computational notebook research and development:

1. It provides a syntactic definition of notebooks, viewed from both client and server perspectives.
2. It identifies a number of issues with the current state-of-the-art in notebook programming.
3. It outlines promising avenues for future enhancements in the notebook programming model, including support for modularity, versioning, distribution, and introspection.

2 Popularity of Notebooks

Notebooks are universally popular. There is incontrovertible evidence for this. For example, the number of notebook files stored in the GitHub centralized repository is around 10 million (as of Oct 2020) and growing exponentially [19].

In this section, I examine various potential reasons for this growth in notebook usage.

2.1 Learner Testimony

The appeal of notebooks springs from their apparent simplicity and accessibility [29]. The following verbatim quotes are all taken from online learners participating in our Teaching Data Science MOOC. These feedback comments were captured immediately after their initial exposure to interactive notebook programming in the first week of the course.

In terms of the exploratory, interactive nature of the live code blocks with instant feedback, learners felt they were 'playing around' (two people said this) and 'having a go.' Others commented that 'it was fun' and 'made sense to me.'

In terms of the highly structured notebooks with small code blocks interspersed by small text blocks, one learner said they 'do really like the simplicity.' Another felt the exercise was 'very interactive yet supportive at the same time.' A further participant stated, 'I like the way you explained what would happen in each section.'

2.2 Channelling Zeitgeist

Notebook programming appropriately fits contemporary coding practice. Programmers hunt for useful code snippets via highly specialized online search. This 'stack overflow mentality' which leads to 'cut-n-paste coding' is increasingly common [3, 28]. It is a natural tendancy in the era of post-modern programming [18].

The notebook format—small cells containing inline code, output, and explanatory rich text scaffolding—lends itself to code borrowing and sharing in an exploratory, interactive manner. Further, notebooks are a highly shareable format, particularly with the *nbviewer* tool enabling a notebook to be exported as a HTML document.

Notebook programming involves browser-based interactions. In that sense, notebook development is platform agnostic, tapping into the Bring Your Own Device trend for the different kinds of developers. The next section identifies where these developers come from.

2.3 Diverse Audience Appeal

While computational notebook usage is widespread, there appear to be three concentrated domains where notebooks are particularly prevalent.

2.3.1 Novice Developers. Inexperienced coders generally appreciate the exploratory nature of interactive interpreters or read-eval-print-loop (REPL) systems. The notebook is one step up from that in terms of its structure, but it retains the immediate feedback of a REPL. Well-designed notebooks are effectively interactive textbooks for learners.

Typical notebook environments for beginners require minimal client-side toolchain installation or they use a hosted solution with no client-side requirements apart from a web browser. The familiar browser interface also helps reassure novices.

One learner from our Teaching Data Science MOOC, who had never encountered notebooks before, commented: 'What a wonderful intro into data science and Jupyter Notebook. I still love these independent cells. Smart ;)'

2.3.2 Scientific End-user Developers. End-user software developers [13] who work in scientific research [27] are the original intended audience for Jupyter [11]. For such scientists, their code is not the primary output. They care more about the results of the analysis. However code is necessary so a computational notebook acts as a reporting tool, much like a lab notebook [26].

Reproducibility is a key goal in scientific research. Computational notebooks facilitate reproducible experiments [21, 23].

2.3.3 Data Scientists. Professional data scientists are intensive 'power users' of notebooks [10, 20]. The interactive nature, coupled with convenient integration to standard libraries, enables efficient exploratory data analysis.

3 History of Notebooks

The importance of lab notebooks as a permanent scientific record is increasingly recognized [7]. There is a slow progression towards electronic lab notebooks in traditional sciences [8]. This trend is more rapid in information sciences [26].

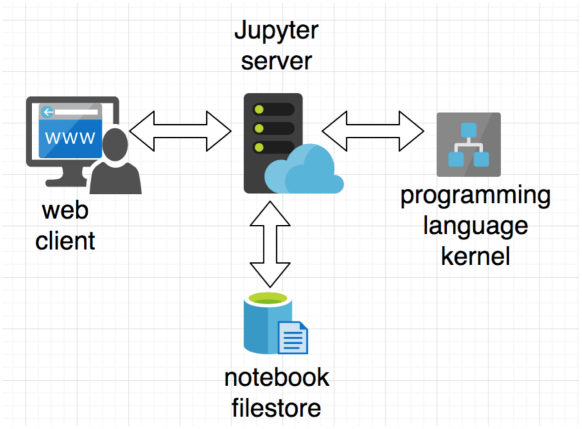

Figure 1. Architecture of the Jupyter Notebooks system

The concept of *literate programming* was introduced by Knuth to combine source code and rich textual commentary into a single coherent document [12]. The *REPL* facility, for interactive interpretive execution with inline execution feedback, was pioneered in LISP [16] with the `eval` function. Computational notebooks blend both these notions by providing interactive code execution with interleaved output, in context of a richly annotated text document.

The proprietary Mathematica package introduced the computational notebook in 1988; this has since evolved into the online Wolfram Computational Notebooks system[2].

Notebook-based data analysis is supported in R, with systems like Sweave [15] and knitr [30].

Currently, the most popular notebook framework is Jupyter [11]. This system supports a massive range of languages, using a pluggable kernel framework for language interpretation. Jupyter is a distributed system:

- the **client** operates in a web browser, rendering the notebook and capturing user interaction
- the **server** is integrated as a web server host, serving data to one or more clients
- the **kernel** is responsible for interactive program execution, it runs on the server and accumulates state for each client (e.g. defined variables and dynamically allocated in-memory data structures)
- the **filestore** saves notebook files as persistent JSON records, which are stored on the server but can be downloaded by the client

Figure 1 gives a schematic overview of the Jupyter system.

Jupyter is available for users to install locally, in which case the client and the server are the same machine. Commercially hosted versions of Jupyter are commonplace: these include Binder[3], Google's Colab[4] and the Noteable system [5].

Our notebook programming course, Getting Started with Teaching Data Science in Schools, was aimed primarily at novice coders. We felt that the meandering menu system of Jupyter was unsuitable (cf. Figure 2) as were some of its code execution features. We chose to implement a cut-down hosted notebook execution framework called ErysNotes[6].

4 Abstract Description of Notebooks

In this section, we intend to capture the inherent characteristics of interactive notebooks. Section 4.1 describes a characterization of the structure of a static notebook as a sequence of consecutive cells, visible on the client-side. Section 4.2 outlines the dynamic view of a notebook from the server-side.

4.1 Notebook Structure

We consider a notebook to be a linear sequence of cells. The order of cells is specified, so we cannot denote a notebook by a set of cells, instead we need to use a list. Alternative notebook geometries are being explored, such as a spreadsheet-style, two-dimensional grid of cells [17]. However, we restrict attention to one-dimensional notebooks for now.

Each cell has a specific type, which is either code or markdown. A code cell contains source code in a specified programming language. A code cell optionally has output associated with it, if it has been executed and the execution caused a visible side-effect (such as a print statement). A markdown cell contains text that can be rendered by a markdown formatter. Figure 3 presents a context-free grammar to express this linear sequence of cells in a notebook.

A Jupyter notebook is encoded as a JSON data structure. Further cell metadata may be captured, such as the number of times each code cell has been executed. Notebook metadata including the source language and version number are stored as key/value pairs.

This is the static, client-side view of a notebook. The user may manually edit the structure of the notebook, adding new cells, rearranging cell order or changing cell type between code and markdown.

4.2 Dynamic View

When a notebook is executing, it dynamically accumulates state. A *kernel* process runs on the server, acting as an interactive interpreter for the client. Execution state builds up over a sequence of code cell invocations from the notebook. However this ordering of code cell invocations is arbitrary,

[2] https://www.wolfram.com/notebooks/

[3] https://mybinder.org

[4] https://colab.research.google.com

[5] https://noteable.edina.ac.uk/

[6] https://github.com/citizendatascience/ErysNotes

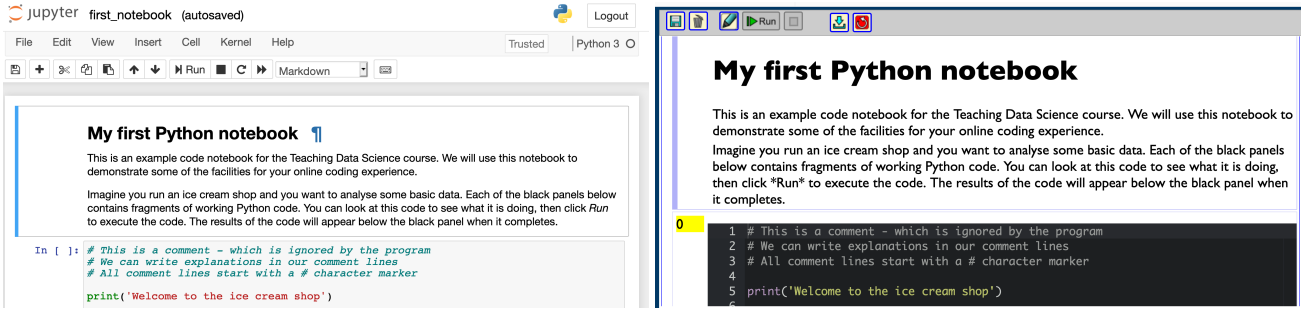

Figure 2. Comparison of interface on Jupyter Notebook (left) and ErysNotes (right)

$$
\begin{aligned}
\langle \text{notebook} \rangle &::= \langle \text{cell} \rangle^* \\
\langle \text{cell} \rangle &::= \langle \text{codecell} \rangle \mid \langle \text{textcell} \rangle \\
\langle \text{codecell} \rangle &::= \langle \text{codeblock} \rangle \ \langle \text{outputblock} \rangle \\
\langle \text{outputblock} \rangle &::= \texttt{output} \mid \epsilon \\
\langle \text{codeblock} \rangle &::= \texttt{source} \\
\langle \text{textcell} \rangle &::= \texttt{markdown}
\end{aligned}
$$

Figure 3. BNF grammar for static notebook structure as a sequence of cells

as far as the notebook structure is concerned, since the user selects cells to execute.

Notebook execution appears to be a sequence of code block executions, which is simply a series of lines of source code interpreted consecutively to accumulate program state, consisting of variables, data structures, etc. Observable outputs such as graphics and print statement results are transmitted back from the kernel to the server using the standard IPython messaging protocol[7] and then relayed from the server to the client as HTML for display in output blocks.

When the client is 'attached' to a server (more precisely, to a kernel) then state can be queried by and communicated to the executing notebook on the client. However this program state is transient, so when the notebook instance is disconnected from the server (perhaps due to a network issue) then all the underlying runtime state is lost. The notebook is still visually complete, in terms of output cells previously computed. This output may be persisted by saving the notebook locally. However once the notebook is disconnected, no further incremental execution is possible. When the notebook is reconnected to the kernel, or reloaded in a new Jupyter context, execution recommences with an empty runtime state.

5 Notebook Complexity

This section examines some of the difficulties associated with notebook programming, particularly in the context of the Jupyter Notebook framework. I divide complexities into incidental (Section 5.1) and intrinsic (Section 5.2) and describe them separately below.

5.1 Incidental Complexities of Notebooks

These presentational worries are mostly caused by the Jupyter Notebook user interface. Our ErysNotes system bypassed these complexities in a refactored interface, while preserving the underlying JSON-based notebook file format.

5.1.1 Cell Execution Order. A key problem with computational notebooks where source code is split across multiple cells is that these cells may be executed by the user in an arbitrary order. A non-linear execution sequence is possible, including multiple executions of the same, non-idempotent cell. Although the order of cell execution can sometimes be inferred from Jupyter (see sequencing labels on left of executed cells), in the case where multiple cells are executed multiple times then the execution order is no longer apparent.

Generally, notebook developers follow standard conventions like expecting cells should be executed in sequential (top-to-bottom) order. In our online course for Teaching Data Science, we respected this sequential cell execution convention in all template notebooks for learners, and also ensured that that all cells had idempotent code so multiple executions of the same code block would not change the notebook behaviour.

It seems that other educators adopt similar self-imposed constraints on their notebooks. For instance, one colleague told me: 'I put all the code in a single block to avoid sequencing issues.' Another said: 'I click 'Run all' when first open a notebook.'

There is an open issue[8] on GitHub for the Jupyter Notebook project to 'enforce a top-down order of execution'

[7] https://jupyter-client.readthedocs.io/en/latest/messaging.html

[8] https://github.com/jupyter/notebook/issues/3229

which imposes the behaviour educators are trying to encourage.

Some extensions to Jupyter Notebook support an explicit dependence graph for cell execution encoded directly into the notebook structure, providing users with a clear view of inter-cell dependences [14].

An alternative solution might involve a script management system like Proof General [4]. This system handles interactive execution of scripts for theorem proving, distinguishing between code already executed and code remaining to be executed. It manages the dialogue between a user and an interactive text-based shell interface, which is entirely unconstrained in Jupyter at present.

5.1.2 Decoupled Persistence. As outlined in Section 4.2, it is possible for a notebook to be disconnected from a server kernel, at which point that notebook loses its runtime state. This occurs when a client is disconnected from a server or when a notebook file is saved and reloaded in a different server context.

The confusion arises because outputs from executed code cells are retained, so a user intuitively (but incorrectly) assumes that any state which generated these outputs is still available.

Our ErysNotes system incrementally serializes the Python interpreter kernel state on the server immediately after each code block execution. Thus the notebook state and the backend kernel state are both persisted and synchronized. This requires unique user identification (we used LTI[9]) to serve each user with their own saved notebook and interpreter state. In the worst case, it means there is lots of duplicated data on the server. In practice we did not find this to be a problem: for our beginner course, we had 500 learners and only 2.3MB of pickled Python state in total across all learner profiles. Admittedly, all our course notebooks were short and simple to suit the novice audience.

5.1.3 Schrödinger's Notebook. The simple operation of loading and inspecting a notebook has the potential to modify it. At the very least, it can change OS timestamps on a file's access times. More significantly, the notebook metadata might be updated. Once the user begins to make changes to the notebook cells, then any rolling back of edits is subject to the vagaries of the web browser undo facility or checkpointed saves of the notebook state.

In our ErysNotes system, we implemented a readonly cell feature, to prevent user editing of fixed cells (instructional text cells or essential library imports, for instance). Further, we provided a 'reset' button on the toolbar to allow users to roll back the notebook state to a default 'clean' notebook, which is predefined by the course educator for each notebook activity.

[9] http://www.imsglobal.org/activity/learning-tools-interoperability

5.2 Intrinsic Complexities of Notebooks

In this section, I examine some intrinsic notebook complexities. Solving these challenges will require more significant engineering effort. Possibly this is a subjective assessment on my part, but no satisfactory solutions for these problems exist to date.

Mature programming language systems must facilitate appropriate encapsulation, efficient software process integration and reflective programmatic interaction.

5.2.1 No Modularity. Notebooks support basic scripting activity, but it is unclear how to grow notebooks to handle large-scale software engineering projects. Each program is contained in a single notebook; it is not straightforward to construct notebooks that contain other notebooks.

The standard programming language approach to building larger scale systems is modularity. Notebooks might support modularity, e.g. it is possible to import Python modules into a Jupyter notebook; however, notebooks cannot be directly mapped onto Python modules themselves. How can we import one notebook's code into another notebook? The most common approach appears to be a copy-and-paste operation [22]. It is impossible to inherit a notebook, to use object-oriented parlance, and simply override one part. Instead we must clone the notebook and modify the required cells in place.

The key issue is that the notebook is designed to be the top-level orchestration script for a computation; anything underneath should be developed and deployed using traditional code preparation techniques. It's not fractally recurring notebooks, or 'notebooks all the way down,' in terms of the source code. The notebook contains the top-level script; currently it is only practical for this purpose.

There are complex workarounds like the ipynb module[10] and registering callbacks for Python import hooks[11]. Neither is a particularly satisfactory solution.

5.2.2 No Concurrency. The idea that there is a single thread of execution, with code executing in one place at one time, is ideal for beginner programmers using notebooks.

In a Jupyter notebook, it is only possible to execute a single cell at once, or to queue a sequence of cells for consecutive execution.

What would a distributed system look like when written as a notebook, or a set of notebooks? Could it be developed and coordinated easily in a notebook framework?

There are some specialized systems for data parallel computing. For example, ipyparallel[12] coordinates a cluster of indexed kernels to compute Python code on a set of server

[10] https://ipynb.readthedocs.io/en/latest/

[11] https://jupyter-notebook.readthedocs.io/en/stable/examples/Notebook/ImportingNotebooks.html

[12] https://ipyparallel.readthedocs.io/en/latest/

nodes. However, this maps awkwardly onto the single notebook model.

5.2.3 No Versioning. Since notebooks are structured JSON documents, it is difficult to use standard text-based source code management tools like diff and patch. This difficulty extends to version control systems like git that operate principally on text-based source code artifacts.

Typical advice on git version control for Jupyter [25] seems simplistic. It involves scrubbing outputs, reverting to HTML or Python, then saving text files. This is a 'lowest common denominator' approach. There is no version control information for Jupyter encoded in notebook metadata.

Specialist tools are coming into play, such as nbdime[13]. This supports 'intelligent' diffing of notebooks, since it can show differences in code blocks and highlight differences in outputs. Is it necessary for every source code processing tool (e.g. diff, patch) to be 'ported' to a version that parses JSON format explicitly? If this is the case, the tools will be closely coupled with the notebook format.

A fundamental problem is that text-based check-in of versions does not fit well with the interactive nature of notebook development. Users might prefer some kind of sophisticated checkpointing and rollback, like the time travel feature of Cocalc[14]. While there are plugin modules to supply this facility for Jupyter, none are integrated by default and they are complex to configure.

5.2.4 No Introspection. In some senses, a HTML document with its associated Document Object Model (DOM), is like a computational notebook. Both HTML documents and computational notebooks contain interwoven marked up text and executable source code. The distinguishing feature of HTML is that Javascript, when executed, can modify the DOM dynamically. In that sense, the document structure is exposed to the Javascript, and therefore can be subject to live programmatic updates.

On the other hand, notebook code—although more visible than Javascript in a web page—does not have any direct links between the documentation (markdown cells) and the source code. There is no dynamic interaction between them. For instance, there is no way Python code in a notebook can determine which cell block it belongs to. There is some limited introspection capability with the `inspect` module in Python and the `get_ipython` method. However these relate to Python code that has been executed by the kernel, and do not provide meaningful handles back to the client-side notebook structure.

Each code block effectively exists in independent isolation. The kernel on the server is unaware of the overall notebook structure. Unfortunately this makes the notebooks somehow static and unresponsive. The notebook is not reified at runtime, so cannot be introspected or updated programmatically. In the same way as Smalltalk popularized the concept of reflective programming [9] we need a meta notebook protocol to enable reflective programming for Notebooks.

The last resort is to use Jupyter `magic` directives. This is a kludge to support specific hard-coded kernel runtime behaviour. However end-users should never need to resort to magic; they should be able to accomplish elegant, rich reflective programming in their source code language of choice.

6 Related Work

Grus [6] identified some of these shortcomings before me: notably problems with out-of-order code block execution, lack of modularity, and lack of version control. These pitfalls are documented elsewhere [1] but without proposals for solutions. There are many online blog articles with titles like 'Why I don't like Jupyter Notebooks' containing similar arguments.

Pimentel et al [22] report on a study of over 1 million Jupyter notebook files downloaded from GitHub; they discover that only 4% of these notebooks generated reproducible results.

Rule et al [24] report on a different study of over 1 million Jupyter notebook files downloaded from GitHub; they discover that only 25% of these notebooks contain explanatory text cells—the remainder consist entirely of source code and saved outputs.

Chattopadhyay et al [5] highlight nine 'pain points' with contemporary computational notebook platforms, based on a rigorous mixed-methods user study. Their user base consists of expert data scientists with established workflow techniques. The problems identified by the study overlap with some of our issues, including preservation of state, versioning, modularity and distribution of notebook code.

7 Conclusions

We have outlined the 'state of the notebook' at present, particularly focusing on the Jupyter Notebook ecosystem.

We have examined a range of challenges facing computational notebook users and have pointed out promising solutions, some of which have commenced in development already. Note there is no extant solution to the problem of notebook reflection; there is no meaningful notion of meta notebook programming. This could be a rewarding area for future investigation.

Acknowledgments

The online course development associated with this work was funded by The Data Lab in Scotland. The ErysNotes system was created by Niall Barr at the University of Glasgow.

[13] https://nbdime.readthedocs.io/en/stable/
[14] https://cocalc.com/doc/jupyter-notebook.html

I gratefully acknowledge these and other collaborators who helped to shape my thoughts on this topic.

References

[1] Aalto Science. 2020. Pitfalls of Jupyter Notebooks. https://scicomp.aalto.fi/scicomp/jupyter-pitfalls.

[2] Tony E. Adams, Stacy Holman Jones, and Carolyn Ellis. 2015. *Autoethnography (Understanding Qualitative Research)*. Oxford University Press.

[3] L. An, O. Mlouki, F. Khomh, and G. Antoniol. 2017. Stack Overflow: A code laundering platform?. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 283–293. https://doi.org/10.1109/SANER.2017.7884629

[4] David Aspinall. 2000. Proof General: A generic tool for proof development. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 38–43. https://doi.org/10.1007/3-540-46419-0_3

[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*. 1–12. https://doi.org/10.1145/3313831.3376729

[6] Joel Grus. 2018. I don't like notebooks. Presentation at JupyterCon, slides at https://t.co/30peBFwTbv?amp=1.

[7] Frederic Lawrence Holmes, Jürgen Renn, and Hans-Jörg Rheinberger. 2006. *Reworking the bench: Research notebooks in the history of science*. Springer.

[8] Samantha Kanza, Cerys Willoughby, Nicholas Gibbins, Richard Whitby, Jeremy Graham Frey, Jana Erjavec, Klemen Zupančič, Matjaž Hren, and Katarina Kovač. 2017. Electronic lab notebooks: can they replace paper? *Journal of Cheminformatics* 9, 1 (2017), 31. https://doi.org/10.1186/s13321-017-0221-3

[9] Alan C. Kay. 1996. The Early History of Smalltalk. In *History of Programming Languages—II*. 511–598. https://doi.org/10.1145/234286.1057828

[10] Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E. John, and Brad A. Myers. 2018. The Story in the Notebook: Exploratory Data Science Using a Literate Programming Tool. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. 1–11. https://doi.org/10.1145/3173574.3173748

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

[12] Donald Ervin Knuth. 1984. Literate programming. *Comput. J.* 27, 2 (1984), 97–111. https://doi.org/10.1093/comjnl/27.2.97

[13] Andrew J. Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, Mary Beth Rosson, Gregg Rothermel, Mary Shaw, and Susan Wiedenbeck. 2011. The State of the Art in End-User Software Engineering. *ACM Comput. Surv.* 43, 3, Article 21 (2011), 44 pages. https://doi.org/10.1145/1922649.1922658

[14] David Koop and Jay Patel. 2017. Dataflow notebooks: encoding and tracking dependencies of cells. In *9th USENIX Workshop on the Theory and Practice of Provenance (TaPP 2017)*.

[15] Friedrich Leisch. 2002. Sweave: Dynamic generation of statistical reports using literate data analysis. In *Compstat*. 575–580. https://doi.org/10.1007/978-3-642-57489-4_89

[16] John McCarthy. 1978. History of LISP. *SIGPLAN Not.* 13, 8 (Aug. 1978), 217–223. https://doi.org/10.1145/960118.808387

[17] Hisham Muhammad. 2019. Userland: creating an integrated dataflow environment for end-users.

[18] James Noble and Robert Biddle. 2004. Notes on Notes on Postmodern Programming: Radio Edit. In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. 112–115. https://doi.org/10.1145/1028664.1028710

[19] Peter Parente. 2014. Estimate of Public Jupyter Notebooks on GitHub. https://github.com/parente/nbestimate.

[20] Jeffrey M Perkel. 2018. Why Jupyter is data scientists' computational notebook of choice. *Nature* 563, 7732 (2018), 145–147. https://doi.org/10.1038/d41586-018-07196-1

[21] Stephen R Piccolo and Michael B Frampton. 2016. Tools and techniques for computational reproducibility. *GigaScience* 5, 1 (07 2016). https://doi.org/10.1186/s13742-016-0135-4

[22] J. F. Pimentel, L. Murta, V. Braganholo, and J. 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)*. 507–517. https://doi.org/10.1109/MSR.2019.00077

[23] Adam Rule, Amanda Birmingham, Cristal Zuniga, Ilkay Altintas, Shih-Cheng Huang, Rob Knight, Niema Moshiri, Mai H Nguyen, Sara Brin Rosenthal, Fernando Pérez, et al. 2019. Ten simple rules for writing and sharing computational analyses in Jupyter Notebooks. *PLoS computational biology* 15, 7 (2019). https://doi.org/10.1371/journal.pcbi.1007007

[24] Adam Rule, Aurélien Tabard, and James D. Hollan. 2018. Exploration and Explanation in Computational Notebooks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. 1–12. https://doi.org/10.1145/3173574.3173606

[25] David Schmüdde. 2019. How to Version Control Jupyter Notebooks. https://nextjournal.com/schmudde/how-to-version-control-jupyter.

[26] Helen Shen. 2014. Interactive notebooks: Sharing the code. *Nature* 515, 7525 (2014), 151–152. https://doi.org/10.1038/515151a

[27] James Somers. 2018. The scientific paper is obsolete. *The Atlantic* (2018).

[28] B. Vasilescu, V. Filkov, and A. Serebrenik. 2013. StackOverflow and GitHub: Associations between Software Development and Crowdsourced Knowledge. In *2013 International Conference on Social Computing*. 188–195. https://doi.org/10.1109/SocialCom.2013.35

[29] Greg Wilson, Fernando Perez, and Peter Norvig. 2014. Teaching Computing with the IPython Notebook (Abstract Only). In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*. 740. https://doi.org/10.1145/2538862.2539011

[30] Yihui Xie. 2013. knitr: A general-purpose Tool for dynamic report generation in R.