Notes on the Code Quality Culture on Jupyter (Notebooks)

Daniel Speicher[*], Tiansi Dong[*], Olaf Cremers[*], Christian Bauckhage, Armin B. Cremers
{dsp, dongt, cremerso, bauckhage, abc}@bit.uni-bonn.de
Bonn-Aachen International Center for Information Technology, Universität Bonn

While we argued in [5, 4] that code quality needs to take context into account, there is now software that demands a *really different quality culture* like we would be entering another planet. Jupyter, to be precise: A "Jupyter Notebook is [a] web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text."[1] It consists of text and code cells. The content of code cells is sent on demand to a Python session, executed and the output inserted below the cell. We will approach the quality of notebooks from the perspective of *communicative code* and *design patterns*.

Communicative Code Let us discuss three aspects of how a developer may communicate through code that are special for notebooks or mathematical code: 1) *Imports should be at the beginning of a file* so that a reader can immediately recognize what the code requires to run and what the reader needs to know to understand the code. So, if a reader finds an import somewhere in-between, he has to readjust his expectations while reading. And, a missing dependency might only be found after long running calculations are already completed. But, couldn't there be a *message behind a late import*? We found notebooks covering a main topic, e.g. an involved calculation, and some minor side topics, e.g. a performance evaluation or a visualization. By postponing imports for the side topics to later sections the notebook emphasizes that these imports are not relevant for the main topic. 2) *Shorter identifier names take longer to comprehend* (see [2] and the discussion of related work therein) so that longer identifier names are preferable. The research indicates that the positive effect of longer names stems from their ability to facilitate the understanding of their meaning. *In mathematical contexts there are short identifiers that have well established meaning*, so that there is at least reason to believe that the established short identifiers outperform longer unfamiliar identifiers in terms of recognizability.[2] 3) Domain Driven Design has elaborated the benefits of *keeping model and code close to the terminology of the domain*. As long as developers and domain experts use the same language, they can efficiently collaborate to evolve and improve the system. Even within the mind of one developer it is preferable to have as little translation effort as possible. Approaching the domain of mathematics and statistics with this mindset, we recognize that there are implementation variants that are closer to the domain than others. Compare for example the following three lines to the formula for the *ordinary least squares solution* $(X^TX)^{-1}X^Ty$:

```
np.dot(np.dot(la.inv(np.dot(X.T, X)), (X.T)), y)
la.inv(X.T.dot(X)).dot(X.T).dot(y)
(X.T * X).I * X.T * y
```

All three variants are valid implementations. The third sticks out as it is almost a verbatim translation of the original formula. This quality might or might not outweigh performance or numerical criteria.[3]

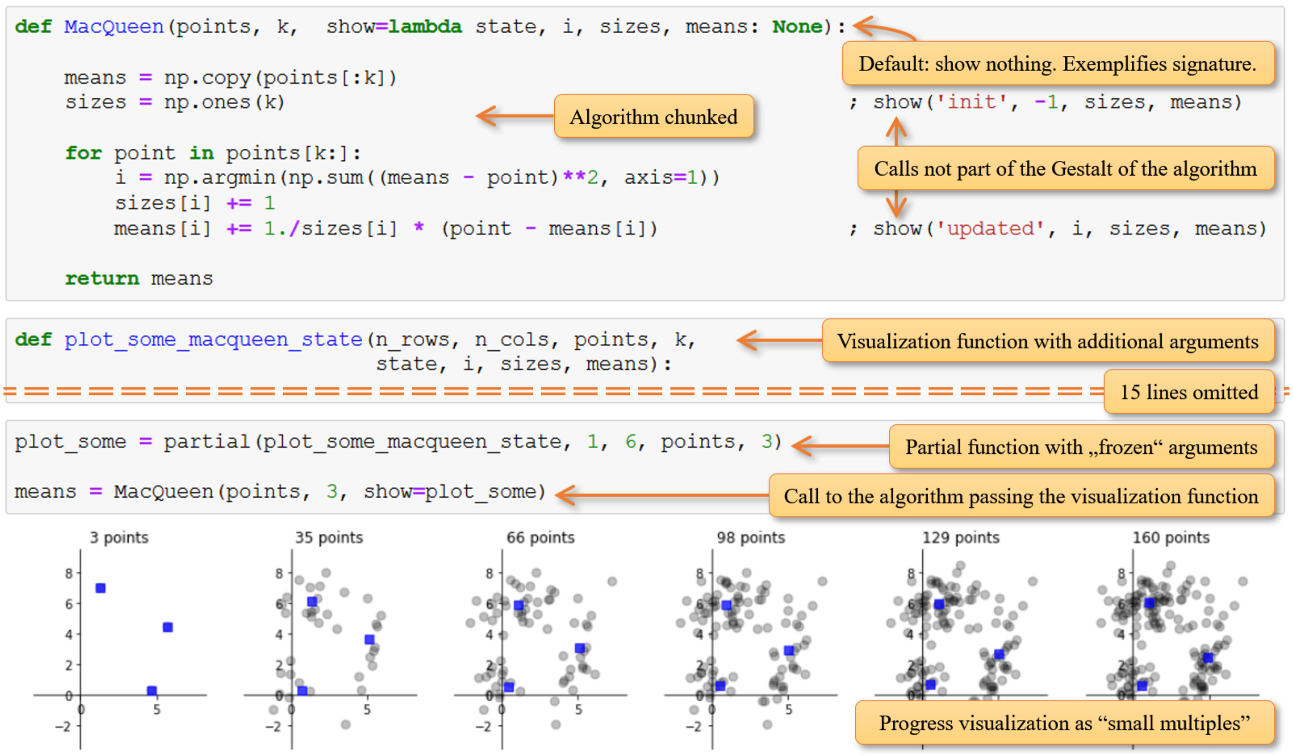
Design Patterns The notion of a *design pattern* being a *solution to conflicting forces in a context* (see e.g. [1, S. 1.1]) turns out to be helpful for the design of notebooks. The context of a calculation presented as a linear narrative leads to solutions that differ substantially from solutions for other kinds of software. 1) Function Exemplification: *Forces:* Notebooks present code and its result in a linear sequence, yet the result of a function definition is a defined function and no immediate output. Self defined functions (let alone objects) are therefore used much less frequently in notebooks than in other software. Still, they are helpful for internal reuse and to give structure to a longer calculation. *Solution:* For functions without side effects, short runtime and easy to provide parameters it is possible to illustrate the use of the func-

[*]This paper was prepared within the project P3ML which is funded by the Ministry of Education and Research of Germany (BMBF) under grant number 01/S17064. The authors gratefully acknowledge this support.

[1]`https://jupyter.org/`

[2]Interesting research questions arise from the constraints within the programming language compared to common mathematical notations: a) Should Greek variable names be used if they are common in maths? b) As statisticians use $\hat{y}$ for estimators for y, should a Unicode variable name be used to represent $\hat{y}$ in the code, or `y_hat` (representing "$\hat{y}$" representing "estimated y"), or `y_estimated` or `y_est` (immediately representing "estimated y" but ignoring the established notation)? Aspects of cognitive psychology as well as pragmatic aspects of editing effort play a role in this question.

[3]For a few more details we like to refer to our notebook "Ordinary Least Squares Optimization" at `https://p3ml.github.io/` and the technical report on which it builds. In the third variant `X` and `y` are matrices. Matrices are less often used than `numpy` arrays as in the other two cases.



```
def MacQueen(points, k,  show=lambda state, i, sizes, means: None):

    means = np.copy(points[:k])
    sizes = np.ones(k)                                              ; show('init', -1, sizes, means)

    for point in points[k:]:
        i = np.argmin(np.sum((means - point)**2, axis=1))
        sizes[i] += 1
        means[i] += 1./sizes[i] * (point - means[i])                ; show('updated', i, sizes, means)

    return means
```

```
def plot_some_macqueen_state(n_rows, n_cols, points, k,
                             state, i, sizes, means):
```

```
plot_some = partial(plot_some_macqueen_state, 1, 6, points, 3)

means = MacQueen(points, 3, show=plot_some)
```

Default: show nothing. Exemplifies signature.
Algorithm chunked
Calls not part of the Gestalt of the algorithm
Visualization function with additional arguments
15 lines omitted
Partial function with „frozen“ arguments
Call to the algorithm passing the visualization function
3 points
35 points
66 points
98 points
129 points
160 points
Progress visualization as “small multiples”

Figure 1: A Visualization Callback allows to keep the `MacQueen` algorithm unconcerned with visualization. There is only the function parameter and the calls, separated by the layout. The callback is still able to produce a visualization consisting of small multiples (see [6, p. 67]) illustrating that the means move mostly at the start and reach plausible positions.

tion in the next cell.[4] 2) Updated Progress Line: *Forces:* When executing the code while exploring own approaches or reproducing results of others, it is essential to get feedback about the progress of long running computations. But once the calculation is done, a larger part of the progress information in the notebook becomes uninteresting and distracting. *Solution:* Let the calculation repeatedly overwrite only temporarily interesting progress information in the same line.[5] 3) Visualization Callback: *Forces:* To illustrate an algorithm, its implementation should not be influenced by other concerns. In addition we often want to show intermediate state of the algorithm. The same implementation should be usable with or without visualization. If it is not visualized it should be fast. It is often interesting to visualize algorithms in varying detail and with respect to different aspects. *Solution:* We pass a function as a parameter to the function that implements the algorithm as illustrated in Fig. 1. As a default value this parameter gets an anonymous function doing nothing.[6] The algorithm function calls the parameter function passing all potentially interesting information in. Visualization functions that actually show something may have additional parameters that can be “frozen” by creating a partial function. If the functions were objects we would talk about a Strategy Pattern with a Null Object as default strategy. If the space in the cell allows, we might position the calls further to the right so that they are preattentively perceived as separated from the algorithm.

We exemplified the possibility to discuss the substantially different code quality of Jupyter notebooks. There are many further opportunities to apply Reverse Engineering to notebooks. For example, notebooks that have served to *explore* data and calculations often need thorough clean-up before they may be passed on to *explain* findings, see [3].

[4]There is no technical reason not to call the function in the same cell, yet we consider defining a function to be substantially different from illustrating it.

[5]`'\r'` positions the cursor at the beginning of the current line so that progress feedback can be given via `print('Progress: {} of {}.'.format(i, n), end='\r')`.

[6]The performance cost of invoking this function is below the cost of three integer additions.

References

[1] E. Gamma, R. Helm, R. E. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[2] J. C. Hofmeister, J. Siegmund, and D. V. Holt. Shorter identifier names take longer to comprehend. *Empirical Software Engineering*, 24(1):417–443, Feb 2019.

[3] A. Rule, A. Tabard, and J. D. Hollan. Exploration and Explanation in Computational Notebooks. *ACM CHI Conference on Human Factors in Computing Systems*, 2018.

[4] D. Speicher. Code Quality Cultivation. In *IC3K 2011, Revised Selected Papers*. Springer Berlin Heidelberg, 2013.

[5] D. Speicher and S. Jancke. Smell Detection in Context. In *WSR 2010, Softwaretechnik-Trends, Band 30, Heft 2*, 2010.

[6] E. Tufte. *Envisioning Information.* Graphics Press, Cheshire, CT, USA, 1990.