Empirical Software Engineering (2021) 26: 65
https://doi.org/10.1007/s10664-021-09961-9

Understanding and improving the quality and reproducibility of Jupyter notebooks

João Felipe Pimentel[1] · Leonardo Murta[1] · Vanessa Braganholo[1] · Juliana Freire[2]

Accepted: 19 March 2021 / Published online: 8 May 2021

Abstract

Jupyter Notebooks have been widely adopted by many different communities, both in science and industry. They support the creation of literate programming documents that combine code, text, and execution results with visualizations and other rich media. The self-documenting aspects and the ability to reproduce results have been touted as significant benefits of notebooks. At the same time, there has been growing criticism that the way in which notebooks are being used leads to unexpected behavior, encourages poor coding practices, and makes it hard to reproduce its results. To better understand good and bad practices used in the development of real notebooks, in prior work we studied 1.4 million notebooks from GitHub. We presented a detailed analysis of their characteristics that impact reproducibility, proposed best practices that can improve the reproducibility, and discussed open challenges that require further research and development. In this paper, we extended the analysis in four different ways to validate the hypothesis uncovered in our original study. First, we separated a group of popular notebooks to check whether notebooks that get more attention have more quality and reproducibility capabilities. Second, we sampled notebooks from the full dataset for an in-depth qualitative analysis of what constitutes the dataset and which features they have. Third, we conducted a more detailed analysis by isolating library dependencies and testing different execution orders. We report how these factors impact the reproducibility rates. Finally, we mined association rules from the notebooks. We discuss patterns we discovered, which provide additional insights into notebook reproducibility. Based on our findings and best practices we proposed, we designed Julynter, a Jupyter Lab extension that identifies potential issues in notebooks and suggests modifications that improve their reproducibility. We evaluate Julynter with a remote user experiment with the goal of assessing Julynter recommendations and usability.

This article belongs to the Topical Collection: *Mining Software Repositories (MSR)*

This work is partially supported by CAPES, CNPq, FAPERJ, and the NYU Moore-Sloan Data Science Environment

Communicated by: Bram Adams and Sonia Haiduc

✉ João Felipe Pimentel
jpimentel@ic.uff.br

Extended author information available on the last page of the article.

Keywords Jupyter notebook · GitHub · Reproducibility · Quality · Lint

1 Introduction

Jupyter Notebook is the most widely-used system for interactive literate programming (Shen 2014). It was designed to make data analysis easier to document, share, and reproduce. The system was released in 2013, and today there are over 9 million notebooks in GitHub (Parente 2020). Jupyter originated from IPython (Pérez and Granger 2007) and, in addition to Python, it supports a variety of programming languages, such as Julia, R, JavaScript, and C. It also allows the interleaving of not only code and text, but also different kinds of rich media, including image, video, and even interactive widgets combining HTML and JavaScript.

Kluyver et al. (2016) advocate the usage of notebooks for publishing reproducible research due to their ability to combine reporting text with the executable research code. However, the format has been increasingly criticized for encouraging bad habits that lead to unexpected behavior and are not conducive to reproducibility (Pomogajko 2015; Grus 2018; Mueller 2018; Pimentel et al. 2019b). Among the main criticisms are hidden states, unexpected execution order with fragmented code, and bad practices in naming, versioning, testing, and modularizing code. In addition, the notebook format does not encode library dependencies with pinned versions, making it difficult (and sometimes impossible) to reproduce the notebook. These criticisms reinforce prior work, which has emphasized the negative impact of the lack of best practices of Software Engineering in scientific computing software (Wilson et al. 2014), regarding separation of concerns (Hürsch and Lopes 1995), tests (Myers et al. 2004), and maintenance (Horwitz and Reps 1992).

While studies have been carried out to better understand how notebooks are used (Kery et al. 2018; Neglectos 2018; Rule et al. 2018), they did not attempt to execute the notebooks and assess characteristics related to reproducibility. Instead, they explored other aspects, including use cases (Kery et al. 2018), narrative (Kery et al. 2018; Rule et al. 2018), and structure (Neglectos 2018; Rule et al. 2018).

In recent work (Pimentel et al. 2019b), we analyzed a large corpus of notebooks to obtain insights into what contributes to their reproducibility or lack of thereof. We used the aforementioned criticisms as a guide to define metrics that reflect the extent of the adoption of both good and bad practices. We computed these metrics for a collection of 1,159,568 unique notebooks from 264,020 GitHub repositories. We found evidences of good and bad quality practices. As good practices, we found that (i) 69.07% of the notebooks have Markdown cells; (ii) 70.90% of the notebooks that have loops or condition structures also have function definitions; and (iii) most notebooks have descriptive filenames (i.e., only 1.99% start with "Untitled" and only 0.69% have "-Copy"in their names). As bad practices, we found that (i) only 1.54% of the Python notebooks import known test modules; (ii) 76.90% of the unambiguous execution order notebooks have at least one skip; (iii) 36.36% have out-of-order cells; and (iv) 21.11% have non-executed code.

Besides, to assess the reproducibility rate, we attempted to execute the notebooks. Out of 863,878 attempted executions of valid notebooks (i.e., notebooks with defined Python version and execution order), only 24.11% executed without errors, and only 4.03% produced the same results. Based on our findings, we proposed a set of best practices for the development of Jupyter Notebooks.

In this paper, we extend our previous work in several directions. To get more insight into the context in which good and bad practices are applied, we select a subset of popular notebooks (based on the number of stars and forks of the repositories they belong to) and compare their results with the overall results. We select this subset with the expectation that notebooks that receive more stars and forks are likely to be of higher quality.

Having the same goal of getting insight into the context of notebooks, we perform a systematic sampling of real notebooks. We perform an in-depth analysis of the notebooks in the sample, looking either for characteristics that are hard to extract automatically from the dataset or qualitative characteristics that are impracticable to manually analyze in a set of over a million notebooks. We also use notebooks from the samples to illustrate good and bad practices.

In the reproducibility study of our original work, we may have underestimated the reproducibility metrics by running all the notebooks in shared environments, following a single cell execution order, and comparing the results at the character-by-character level. We did not consider small acceptable deviations that occur during the execution of notebooks, such as dates referring to the execution moment, memory addresses of Python objects, and small variations of images. Hence, in this paper, we isolate dependencies, test different execution orders, and explore various strategies for comparing notebook results. We discuss how these changes affect the reproducibility rate. Additionally, to gain insights into quality metrics that influence reproducibility, we mine association rules (Agrawal et al. 1994) that relate specific notebook features to both success and failure of reproductions.

Based on our findings and the set of best practices we proposed before, we designed Julynter, a Jupyter Lab extension for linting notebooks. Julynter identifies potential problems and suggests fixes that aim at improving the notebook quality and reproducibility. We also evaluated Julynter with Jupyter users to assess Julynter recommendations, usability, and improved it accordingly.

This paper is organized as follows. Section 2 provides some background about literate programming and Jupyter Notebooks. Section 3 presents the extended analysis of the large corpus of notebooks. In Section 4, we propose a set of best practices for the development of Jupyter Notebooks. We describe and evaluate Julynter in Section 5. We present related work in Section 6. Finally, we conclude in Section 7, where we outline directions for future work.

2 Background

Knuth (1984) introduced the *literate programming* paradigm that, by combining code and natural language, allows programmers to document a program's logic. This paradigm enables the programmers themselves and others to more easily understand the code. The original system was designed for static documents and required two compilation processes (Knuth 1984): tangling and weaving. The tangling process executes the code snippets in the document and produces the results. Then, the weaving combines the text, code snippets, and results to deliver a human-readable document. Nowadays, literate programming is used in *interactive computational notebook environments* (Shen 2014). These environments allow parts of a notebook to be executed with immediate visualization of results and formatted text, avoiding the need for tangling and weaving.

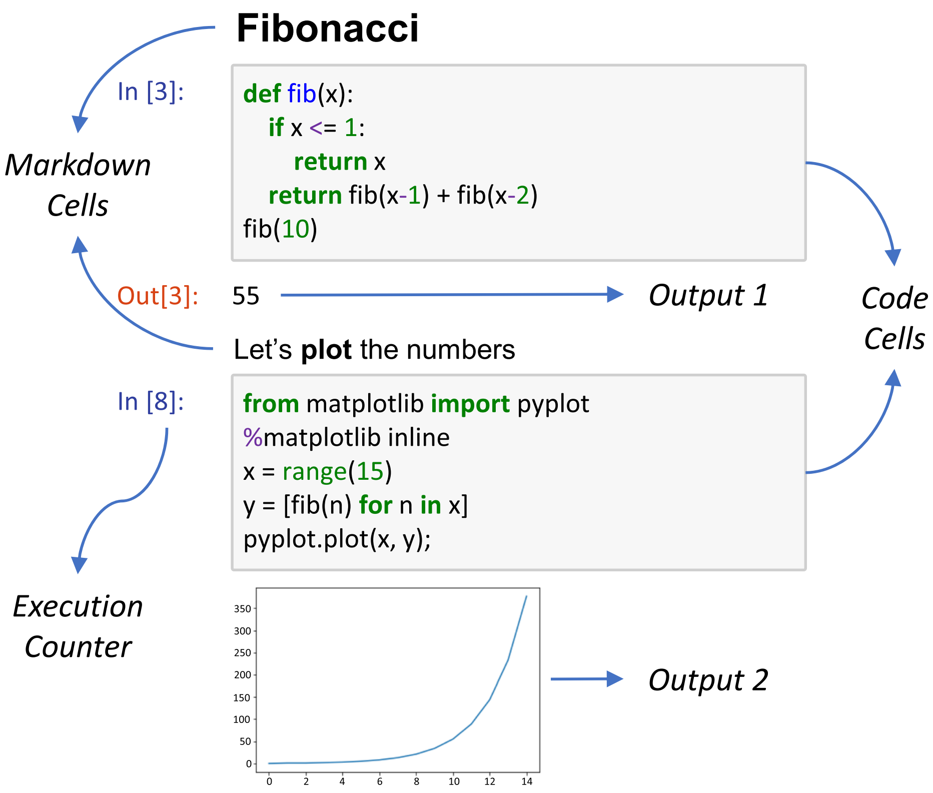
A *Jupyter Notebook* (Shen 2014) is both an interactive literate programming document and an application that executes the document. In this work, to avoid the ambiguity, we use the term *Jupyter* to refer to the application that executes notebooks, such as *Jupyter Notebook* and *Jupyter Lab*. We use the terms *Notebook* or *Jupyter Notebook* interchangeably to refer to the literate programming document.

A notebook is composed of *cells*, which can be of three types: code, Markdown, and raw. A *code* cell contains executable code used to produce results. A *Markdown* cell contains formatted text. Finally, a *raw* cell contains text that is neither code nor formatted text—tools that convert notebooks into other formats use raw cells for configuration.

Jupyter uses a *kernel* to execute code cells. During the execution of a cell, the kernel communicates with Jupyter to display partial and final results. By default, Jupyter displays text, images (PNG, JPG, and SVG), HTML with JavaScript, and Markdown. Additionally, it supports extensions to display other formats. The notebook format uses JSON to store all of its contents in ".ipynb" files. When Jupyter sends a code cell for execution, it marks the cell as *executing* by assigning "*" to the cell *execution counter*. After the execution, the kernel allocates a number to the counter, which indicates the execution order. Users can execute the cells in any order, and a given cell can be executed multiple times.

Storing either *executed* or *non-executed* notebooks is possible. A non-executed notebook contains only *prospective* data (Freire et al. 2008), i.e., the notebook title and definition of its cells. An executed notebook contains *prospective* data plus *retrospective* data (Freire et al. 2008) derived by the execution of the notebook cells—the output of code cells and their execution counters. The execution of a notebook does not require cleaning the outputs of previous executions. Thus, an executed notebook may contain retrospective data from *multiple* executions.

Figure 1 shows an executed Jupyter notebook, which contains two Markdown cells and two code cells. On the left of code cells, Jupyter displays an execution counter that indicates the order in which the cells were executed. Below the code cells, Jupyter displays their



Fibonacci

In [3]:
```
def fib(x):
    if x <= 1:
        return x
    return fib(x-1) + fib(x-2)
fib(10)
```
Out[3]:
```
55
```
Let's **plot** the numbers

In [8]:
```
from matplotlib import pyplot
%matplotlib inline
x = range(15)
y = [fib(n) for n in x]
pyplot.plot(x, y);
```

Fig. 1 An example of an executed notebook with Markdown, code, and output

outputs. Note that the first code cell returns a number, identified by `Out [3]`, and the second code cell displays an image without returning it. This figure also illustrates skips on the execution counters. A *skip* represents cell executions that do not have explicit definitions in the notebooks. In this case, the two executions before the execution counter 3 represent one skip, and the four executions between 3 and 8 represent another.

When initially released, IPython (Pérez and Granger 2007) notebooks supported only Python. The system has evolved into Jupyter, which is language agnostic. Today IPython is the kernel that Jupyter uses for executing Python code. IPython supports a superset of Python. In addition to all Python constructs, it supports *line magics* to execute IPython related commands; *cell magics* to modify the semantics of code cells; *bang expressions* to execute system commands; *cell referencing* to reference the code and output of other code cells; and *help queries* to access the documentation and source code of functions and classes. Note that the second code cell of Fig. 1 uses the line magic `%matplotlib inline` to enable the visualization of *matplotlib* figures.

3 Notebook Study

We analyzed a large corpus of notebooks to obtain insights into what contributes to their quality and reproducibility or lack thereof. This section is organized as follows: Section 3.1 presents the research questions and the definition of their analyses. Section 3.2 describes the collection and preprocessing of the GitHub data. Section 3.3 discusses the selection of a popular set of notebooks that we obtained to use as a baseline and compare it with the overall results. Section 3.4 presents the sampling process we used to gain more insights about the data. Section 6 discusses the corpus of this work. We present the analysis results in Section 3.6. Finally, Section 3.7 presents threats to the validity of our study.

3.1 Research Questions and Analyses

As discussed before, Jupyter has recently been the target of substantial criticism for encouraging bad coding habits and practices that hinder reproducibility (Grus 2018; Mueller 2018; Pomogajko 2015). In what follows, we discuss these criticisms and propose analyses to quantify their impact on notebooks available in GitHub. These criticisms relate to both prospective (i.e., code definition) and retrospective (i.e., code execution) components of notebooks (Freire et al. 2008). We thus frame our analyses in terms of seven research questions (RQ1, RQ2, RQ3, RQ4, RQ5, RQ6, and RQ7), which we organize into two categories: Analysis of Prospective Data, which covers RQ1–RQ4, and Analysis of Retrospective Data, which covers the remaining questions.

Note that these are the same research questions we use in our previous work (Pimentel et al. 2019b). However, as we discuss in the introduction, we answer them now in more detail, with a qualitative analysis of a sampled set of notebooks. We also use examples of real notebooks of the sample to illustrate each aspect of the research questions. Moreover, we compare the obtained results with a set of popular notebooks.

For RQ7, which investigates the reproducibility of the notebooks, we re-executed all the experiments using different execution orders, isolating the execution environments using *Docker*, and also applying normalizations to the notebook outputs to minimize small deviations that do not necessarily indicate a different result. Thus, the reproducibility study we present in this extended paper removes several threats to the validity of the experiments in our original paper.

Analysis of Prospective Data Notebooks store cell definitions and the notebook title as prospective data. In our analyses, we used this information to answer the following questions:

RQ1. *How do notebooks use literate programming features?* According to Wilson et al. (2014), scientists should write programs for people and not for computers. Being a literate programming tool, Jupyter can fulfill this goal. Jupyter allows users to write Markdown cells with text describing the logic behind their programs, followed by direct visualizations of the results. However, the ability to do it does not imply that users will write descriptions or whether these descriptions are meaningful. Grus (2018) pointed out that among the officially recommended tutorials written in Jupyter, there are tutorials with descriptive text that does not correctly explain what the code does. We analyze whether Jupyter is used as a literate programming tool by looking at the number of Markdown cells and their positions in the notebooks. Investigating the presence of linguistic anti-patterns (Arnaoudova et al. 2016) or whether the Markdown descriptions are meaningful for the notebooks is outside the scope of this work.

RQ2. *How are notebooks named?* By default, Jupyter creates notebooks titled "Untitled". It discourages users from choosing meaningful names (Grus 2018). Also, the title is used as the name of the file which stores the contents of the notebook. Using the title as a filename creates OS-based restrictions in the size of titles and the allowed characters (e.g., in Windows, it is impossible to create or use a notebook that has "?" in the title (Microsoft 2018)). Moreover, the choice of notebook title is restricted by the filename conventions adopted by different OS (e.g., not using space characters (Tim and Doorknob 2014)). We analyze the number of untitled notebooks, the number of notebooks with "-Copy" in the title, the size of notebook titles, and the presence of characters not recommended by the POSIX fully portable filenames guide (the guide recommends A-Z a-z 0-9 . _ -) (Lewine 1991).

RQ3. *How do notebooks use modules, functions, and classes?* In traditional programming languages, modules, functions, and classes are essential constructs to maintain the separation of concerns in software (Hürsch and Lopes 1995). In literate programming environments, Markdown cells could be used to separate the concerns. However, this would lead to the lack of referencing and reusability. Moreover, Python treats every script as a module and allows users to import functions and classes from them, which improves the reusability across scripts. However, importing notebooks is hard and unusual (Grus 2018). We extract the Python Abstract Syntax Tree (AST) from cells to analyze the presence of local module imports, and function and class definitions as evidence of separation of concerns.

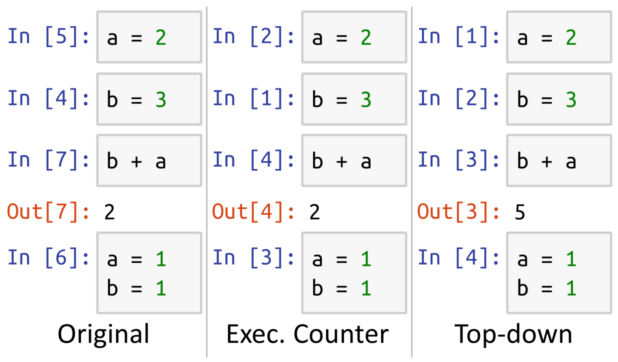
RQ4. *How are notebooks tested?* Testing is a good practice to verify that a given program meets its requirements and keeps working after changes are applied (Myers et al. 2004). Since notebooks are not modules, testing code in a notebook is challenging as it requires mixing test code with the notebook narrative code (Grus 2018; Mueller 2018). To search for evidence of testing in notebooks, we analyze the imported module names that contain "test", "Test", "TEST", "mock", "Mock", or "MOCK" as a sub-string. We also checked for known Python testing tools that do not have these sub-strings (i.e., antiparser, aspectlib, behave, doublex, fit, fudge, fusil, hypothesis, lettuce, ludibrio, mox, nose, peckcheck, pester, pry, pythoscope, reahl.tofu, reahl.stubble, sancho, subunit, taof, twisted.trial). We obtained this list of modules from the categories unit testing tools, mock testing tools, fuzz testing tools, and acceptance testing tools of the Python testing tools taxonomy page (Python-Wiki 2019).

Analysis of Retrospective Data Notebooks store cell outputs and execution counters as retrospective data. We use the following questions to explore the retrospective data.

RQ5. *Do users store notebooks with retrospective data?* Displaying execution results is part of the literate programming aspect of notebooks. The support for rich media enhances the narratives and the writing of programs for people. Moreover, having partial cell results helps in checking the reproduction of a notebook by allowing the comparison of the cell outputs upon re-execution. However, some advocate that the results of notebook execution should be removed before committing to avoid noise in diffs (Staley 2017). Furthermore, Jupyter is also used as an IDE for general-purpose software development with the goal of extracting the produced code to scripts afterwards (Kery et al. 2018). We analyze the number of notebooks that have retrospective data and whether Jupyter is used as a literate programming tool by looking at the output formats (i.e., MIME types of cells' outputs) in executed notebooks.

RQ6. *How are notebooks executed?* Jupyter allows users to execute cells in any order. While notebooks present the cells in a linear top-bottom narrative, a user may choose to execute the cells in a non-linear, arbitrary order. This ability departs from how most people expect to run code (Grus 2018; Mueller 2018; Pomogajko 2015). Moreover, cells that appear at the beginning of notebooks may depend on cells that appear later, leading to additional issues for users that run them in the default top-down order (Koop and Patel 2017). Figure 2 presents an unordered notebook and two re-executions of it following distinct execution orders: cell execution counter order and top-down order. In this example, the order that produces the same results is the one that follows the cell execution order. To quantify the prevalence of this practice, we identify notebooks that have cells in a non-linear order.

In addition to out-of-order cells, when Jupyter executes a code cell, the execution may change a state in the environment. It does not cause problems when users run cells only once and do not change previously executed cells. However, when the user runs the same



```
In [5]:  a = 2
In [4]:  b = 3
In [7]:  b + a
Out[7]: 2
In [6]:  a = 1
         b = 1
```

```
In [2]:  a = 2
In [1]:  b = 3
In [4]:  b + a
Out[4]: 2
In [3]:  a = 1
         b = 1
```

```
In [1]:  a = 2
In [2]:  b = 3
In [3]:  b + a
Out[3]: 5
In [4]:  a = 1
         b = 1
```

Fig. 2 Original notebook and two executions that follow different orders

cell multiple times, edits, or removes the cell code after executing it, the environment state may no longer represent the code definition, and this can lead to bugs and make debugging harder (Grus 2018; Pomogajko 2015).

Figure 3 presents three examples of hidden states caused by these situations. Having a hidden state may make it impossible to reproduce the same results upon the re-execution of the notebook. In fact, the re-execution of these notebooks would produce results that differ from the ones in the output cell. Note that hidden states caused by cell re-execution or removal make the notebooks skip numbers in the execution counter sequence. Thus, in our analyses, we count how many execution counters skips there exist in the notebooks. Also, note that a removed or re-executed cell that causes a skip number does not necessarily produce a hidden state when it has code that does not change the environment. Hence, our measurement states the susceptibility of notebooks to have hidden states rather than confirming that they have them. Additionally, our analysis does not consider hidden states caused by edited cells that were not executed.

We can only analyze the presence of skips and out-of-order cells in unambiguous execution order notebooks. We define *unambiguous execution order notebooks* as notebooks that have only one valid execution sequence. That is, they neither have cells with repeated execution counters, nor cells whose counter indicates that they are being executed. Note that this definition does not guarantee that the notebook outputs represent a single execution, but it is a close approximation with practical implications in our analyses.

Finally, the presence of non-executed code cells in the middle of the notebooks also hinders the reasoning about the execution. We analyze this issue by counting how many non-executed cells are in the notebooks and by comparing their positions with the position of executed ones.

RQ7. *How reproducible are notebooks?* Notebooks do not declare the versions of imported libraries (Grus 2018). The lack of version information may cause incompatibilities and prevent the execution of the notebook in environments that are different from the one in which the notebook was created. In Python, this issue can be addressed by defining dependencies in standard files: `setup.py`, `requirements.txt`, and `Pipfile`. We analyze how many notebooks belong to repositories with such files.

The existence of hidden states, out-of-order cells, hard-coded paths, and other bad practices also prevent the reproduction of notebooks. To assess the rate of reproducibility, we perform a reproducibility analysis of all unambiguous execution order Python notebooks. An unambiguous execution order notebook can have non-executed code cells in the middle. We ignore these cells since they do not have outputs.

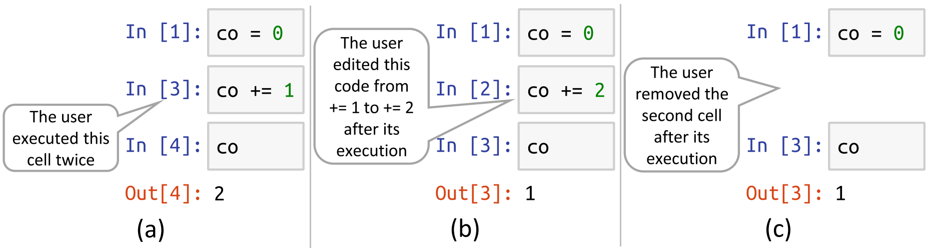

In [1]: co = 0
In [3]: co += 1
In [4]: co
Out[4]: 2
The user executed this cell twice
(a)
The user edited this code from += 1 to += 2 after its execution
In [1]: co = 0
In [2]: co += 2
In [3]: co
Out[3]: 1
(b)
The user removed the second cell after its execution
In [1]: co = 0
In [3]: co
Out[3]: 1
(c)

Fig. 3 Three types of Hidden States: **a** Re-execution; **b** edited cell; **c** removed cell

Execution Modes In this analysis, we try to execute notebooks in five different modes to assess their reproducibility rate, as summarized in Table 1. We assess the rate by identifying notebooks that, when executed, lead to results that are the same as the results stored within the notebooks.

The first execution mode is the one adopted in our previous work (Pimentel et al. 2019b), in which we executed the notebooks following the *cell execution counter order* in a *shared* OS environment with conda and anaconda environments to manage multiple Python installations and kernels. *Conda* is a package and environment management system that installs and manages the dependencies of packages. It allows multiple versions of Python to be installed with different dependencies. Conda was originally designed as part of *anaconda* (Anaconda 2018), which is a Python and R distribution that includes over 100 Scientific Packages, such as *numpy*, *scipy*, *matplotlib*, and other packages. Today, anaconda is a conda package that includes all these dependencies. In this work, we refer both to *conda environment* and *anaconda environment*. When we refer to conda environment, we consider an environment with only Python and Jupyter installed. When we refer to anaconda environment, we consider the environment that bundles the anaconda package and all of its dependencies. The decision on which environment to use was based on the availability of dependency declarations in the repositories. For repositories that declared dependencies, we used a conda environment and attempted to install the dependencies. For the other repositories, we used an anaconda environment with multiple pre-installed packages.

Since the first execution mode uses a shared OS environment, one execution can change system dependencies and affect the execution of other notebooks. Hence, we define four additional execution modes as an attempt to reduce the number of false negatives. In the second execution mode, we use docker containers to *isolate* the executions, and we run cells following the existing cell execution counter. In the third execution mode, we also use docker containers, but we run cells following the *top-down* order. In both these modes, we have anaconda environments installed in the containers. In the fourth and fifth execution modes, we also use docker containers, but we created *bloated* containers by attempting to install the maximum number of packages that we could install. The goal was to prevent notebooks from failing to reproduce due to the lack of dependencies. In the fourth execution mode, we executed notebooks following the existing cell execution counter, and in the fifth execution mode, we executed notebooks following the top-down order. Table 1 presents

Table 1 Execution modes for the reproducibility experiments

#	Mode	Environment	Execution order
1	Shared + Exec. Counter	Shared OS with conda and anaconda environments	Cell Execution Counter
2	Isolated + Exec. Counter	Isolated docker container with an anaconda environment	Cell Execution Counter
3	Isolated + Top-Down	Isolated docker container with an anaconda environment	Top-Down
4	Bloated + Exec. Counter	Bloated docker container with many dependencies installed	Cell Execution Counter
5	Bloated + Top-Down	Bloated docker container with many dependencies installed	Top-Down

all the execution modes that we use in this work. We set a time limit of 5 minutes for the execution of each notebook.

Normalizations In the first execution mode, we performed a direct character-by-character comparison of the output of a re-execution with the saved result to check the reproducibility results. However, this comparison can lead to false negatives due to small differences. For instance, the cell execution counter is part of the cell output. In the first analysis, every notebook with a skip would lead to a non-reproducible notebook. Similarly, insignificant deviations in number, date, and other object formats would lead to a non-reproducible notebook despite the notebook producing a similar result that is semantically the same.

To avoid this problem, we normalize the notebook outputs in the latter four execution modes. The normalization operations we applied are presented in Table 2. We apply these operations in the same sequence they appear in Table 2. Hence, before we apply the stream normalization, we apply both the encode normalization and the execution counter normalization. Thus, having skips in the execution counter or insignificant deviations in formats does not lead to non-reproducible notebooks, reducing the number of false negatives.

We indicate the reproducibility rate for each normalization. Note that the Image normalization is expected to have the highest rate, as it includes all the previous normalization operations. On the other hand, the image normalization is expected to cause false positives, as it strips images out of the notebooks.

Mining Relationships Between Notebook Features and Reproducibility To gain deeper insights into factors that influence reproducibility, we mined association rules (Agrawal et al. 1994) that relate specific notebook features to both success and failure to reproduce notebooks. Association rules have the goal of finding probabilistic associations or correlations. They are expressed as X → Y, where X is the antecedent set, and Y is the consequence set. Their interpretation is based on the amount of evidence determined by three metrics: *support*, *confidence*, and *lift* (Agrawal et al. 1994; Han et al. 2011). These metrics can be calculated as follows:

$$support(X \rightarrow Y) = P(X \cup Y) \tag{1}$$

$$confidence(X \rightarrow Y) = P(Y|X) = \frac{P(X \cap Y)}{P(X)} \tag{2}$$

$$lift(X \rightarrow Y) = \frac{P(X \cap Y)}{P(X) \times P(Y)} \tag{3}$$

The *lift* metric indicates how much the occurrence of X increases the probability of Y occurring. When $lift > 1$, X increases the probability of Y; when $lift = 1$, X does not interfere with Y; and when $lift < 1$, X decreases the occurrence of Y (Han et al. 2011).

Our data mining strategy was conservative by nature—we used a low absolute support threshold of 100 transactions and did not use a confidence threshold. This is important to avoid hindering infrequent but relevant rules. Then, we mined for rules with size two (one feature in the antecedent and one in the consequent) using the Apriori algorithm (Agrawal et al. 1994) provided by package *arules* in R. Next, we sorted the obtained rules descending by lift and fixed features related to reproducibility in the consequent. Finally, we observed what features appeared in the antecedent with lift significantly higher or lower than one.

3.2 Data Acquisition and Preprocessing

We used the GitHub API to find repositories created between January 1st, 2013 and April 16th, 2018 that had a file with "Jupyter Notebook" as identified language. This query

Table 2 Normalization operations for comparing execution results

Operation	How	Reason
Encode	Encodes outputs into UTF-8.	Some notebooks were stored in a different encoding, leading to mismatches.
Execution Counter	Removes the execution counter from outputs.	Skips in the execution counter lead to mismatches.
Stream	Combines print sequences into a single output element.	Different versions of Jupyter/IPython behave differently, leading to mismatches on print statements.
Dictionary	Alphabetically sorts dictionary keys and set elements.	The order of these elements does not matter, but the textual comparison fails for unordered objects.
Dataframe	Removes HTML representation of *pandas* dataframes, keeping only textual representations.	*Pandas* outputs both text representations and HTML representations of the same dataframes, but the HTML representation has changed over time for styling reasons, leading to mismatches.
Exception Path	Removes paths from exceptions.	Python exceptions show the file path, leading to mismatches in different machines.
Deprecation	Removes deprecation warnings.	Deprecation warnings in new versions of libraries lead to mismatches.
White space	Transforms all kinds of white spaces into a single space.	The representation of line breaks and other white space characters changes from system to system, leading to mismatches.
Decimal	Cuts numbers at the second decimal place.	Small variations in float precision lead to mismatches.
Date	Replaces dates by 1970-01-01T.	Running a notebook that outputs the current date at two different dates would result in a mismatch.
Time	Replaces time by 00:00:00.	Running a notebook that outputs the current time at two different times would result in a mismatch.
Memory	Replaces numbers that start with 0x by 0x0000000.	Python objects often indicate their position in the memory on print statements. Since every execution puts the object in a different position, keeping the original number leads to a mismatch.
Image	Removes images from outputs.	It is hard to compare images, and very small changes in image generation lead to different results.

returned 265,888 repositories with 1,450,071 notebooks. We did not collect checkpoint notebooks stored in `.ipynb_checkpoints` directories. Most repositories (60.09%) have 2 or fewer notebooks. Only 12.42% of the repositories have 10 or more notebooks. However, 61.45% of the notebooks belong to repositories with 10 or more notebooks.

On July 22nd, 2020, we queried GitHub again to obtain the number of stars and forks of these repositories. Some repositories were removed from GitHub in this interval. Thus, we removed them from our analyses as well, resulting in 1,274,872 notebooks from 235,643 repositories.

After collecting the repositories, we excluded invalid notebook files, empty notebooks, empty repositories, and repositories that we lost access between the query moment and the analysis moment, resulting in 1,251,074 valid notebooks from 234,729 repositories. From this result, we also excluded 226,805 (18.13%) duplicated notebooks. The goal was to reduce the bias towards forks and notebook copies (Kalliamvakou et al. 2014). We detected these notebooks by calculating the SHA1 hashes from cell sources and output formats. We did not use the output results or other metadata when we calculated the hashes to be able to detect notebooks that only had distinct prospective data as duplicates. This resulted in 1,024,269 unique notebooks for the analyses.

3.3 Popular Notebooks Selection

From the set of unique notebooks, we extracted a set of popular notebooks representing mature notebooks to use it as a baseline for the analyses. This selection reasoning is that those mature notebooks should condense the most quality and reproducibility characteristics as they have received the most attention from users. In fact, we observed that popular notebooks attain more quality features, such as having more Markdown cells (see Section 3.6.1), fewer issues with titles (see Section 3.6.2), and less out-of-order cells and skips (see Section 3.6.6). As we discuss in Section 3.6.7, popular notebooks are more reproducible than the overall group as well. They are 31.04% more likely to execute until the end, 84.83% more likely to reproduce the same results without normalizations, and 41.34% more likely to reproduce the same results after all normalizations.

For selecting the set of popular notebooks, we approximated the popularity of the notebooks based on the popularity of their repositories. Hence, we first assigned to each notebook the number of stars and forks of the repositories containing them. We then computed a popularity score (s) that consists of the harmonic mean of stars and forks of the notebooks. From this, we removed notebooks with zero as the popularity score and obtained top outliers of the remaining notebooks as follows:

$$\begin{aligned} s &\geq 1.5 \times IQR \\ &\geq 1.5 \times (Q3 - Q1) \\ &\geq 33.331 \end{aligned} \tag{4}$$

This selection resulted in a popular set of 38,063 notebooks, which corresponds to 3.72% of the unique notebooks.

3.4 Sampling

To get more insight into the context in which good and bad practices are applied, we systematically extracted a sample of real notebooks from the complete set of unique notebooks. With this sample, we can observe characteristics that we could not extract automatically from the notebooks and use real notebooks as examples. We used Cochran's sample size formula (Israel 1992) to calculate our population's sample size. Assuming a maximum variability ($p = 0.5$), and desiring a confidence level of 90% and ±10% precision range, we calculated the sample size as 68.05 notebooks. Hence, we randomly selected 69 notebooks as our sample.

As a sanity check, we compared 75 metrics with percentages related to the broad set of unique notebooks that we report throughout this work with their sample counterparts. We found that only 11 metrics deviate from the ±10% range when using sub-group selections. However, all of them are calculated based on sub-groups that are not representative enough in the sample (e.g., a percentage over the number of "Untitled" notebooks, which corresponds to only 1.93% of the notebooks). When we do not consider sub-group selections (i.e., we compare the percentages over the total number of unique notebooks or sampled notebooks), all analyzed metrics are within the ±10% range.

In the sample, we found 31 notebooks associated with courses, such as tutorials, class assignments, or course exercises. We also found ten academic notebooks related to papers, dissertations, theses, and capstone projects. Ten notebooks had analysis for tasks not related to education—although some are related to writing blog posts. Nine notebooks were related to personal practicing, such as solving book exercises or exploring new things. Five notebooks described how to use other tools and libraries. Three notebooks are related to books. A notebook is part of a presentation. Given the number of notebooks related to education, understanding and supporting notebooks may greatly impact educational projects.

Given that most notebooks belong to educational projects, we analyzed to which area they belong the most. We counted 28 notebooks related to data exploration, using simple *pandas* functions and plots. Ten notebooks were related to machine learning, with libraries such as *keras*, *sklearn*, and *lasagne*. Additionally, six notebooks were related to data mining, with algorithms for clustering and building decision trees. Five notebooks were related to data cleaning, to transform a data format into another. While most notebooks were data-centric, we also found notebooks related to other areas: programming (four notebooks), databases (three notebooks), math problems (three notebooks), algorithms (three notebooks), computer vision (two notebooks), games (one notebook), computer graphics (one notebook), and physics (one notebook). The last two notebooks only had markdown cells with tasks for students. These results indicate that most notebooks are data-centric analyses.

Despite most notebooks sharing areas and being data-centric, most of them use different datasets for their analyses. Some of them use toy datasets available in existing libraries—others use real data from many different contexts.

3.5 Corpus

We analyzed the declared programming languages of all unique notebooks. Figure 4 presents, in the log scale, the 15 most declared programming languages we found. Python is by far the most used programming language, corresponding to 93.11% of the notebooks. It is followed by R (1.33%) and Julia (0.96%). The popular group has slightly more Julia notebooks (1.05%) than R notebooks (0.99%), but Python still represents most of the notebooks. All notebooks in the sample declare Python as their programming language. However, two notebooks could declare any or no programming language, as they do not have code cells. Moreover, two notebooks are composed mostly of *cell magics* with SQL queries. We also found five Python notebooks in the sample that used bang expressions to invoke shell commands.

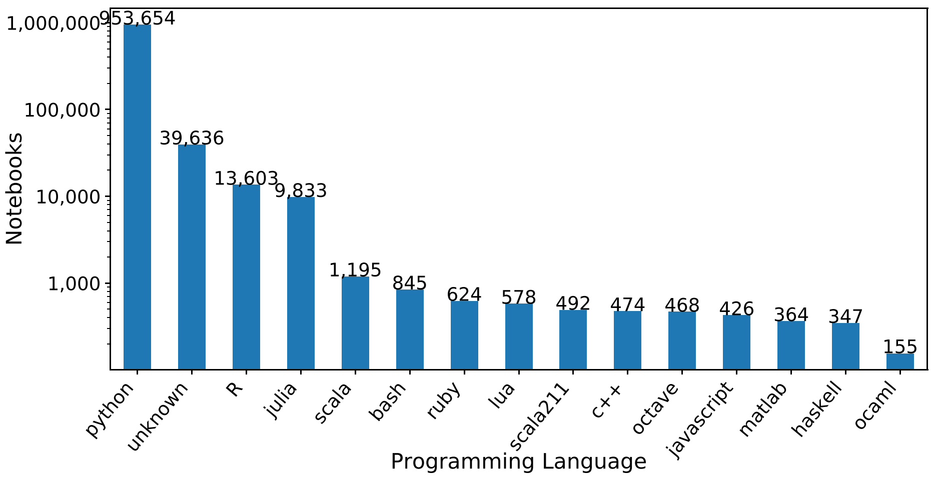
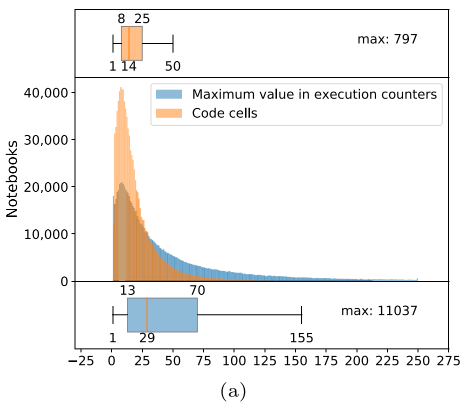


Fig. 4 Top 15 most declared programming languages. Notebooks axis in logarithmic scale

Due to the interactive nature of notebooks, most programming languages are scripting languages. Nonetheless, Jupyter is also used for compiled languages such as C++ and Haskell. A total of 39,636 unique notebooks do not declare a programming language, and 30,953 of them use *nbformat* lower than 4, which predates the release of the language-agnostic Jupyter. Although this is a strong indication that these notebooks also use Python, we opted for removing them from Python-specific analyses.

Since most notebooks contain Python code (953,654) and questions RQ3, RQ4, and RQ7 require language-specific analyses, we focus on Python notebooks to answer these questions. We extracted declared versions and cells with metadata from Python notebooks, and we used the Python AST to extract Python constructs and imported modules. The most used version is Python 2.7, which corresponds to 36.38% of the Python notebooks. However, by combining minor releases, Python 3 surpassed Python 2. In fact, 63.53% of the Python notebooks use Python 3. The remaining did not declare a version. For RQ3 and RQ4, we used only valid Python notebooks (i.e., notebooks with a valid Python syntax in all code cells). Valid Python notebooks correspond to 886,668 (92.98%) notebooks. For RQ7, we did not have this restriction, because we ran only executed cells of Python notebooks with unambiguous execution order, which correspond to 753,405 (79.00%) notebooks.

In addition to these restrictions, we analyzed only executed notebooks for RQ5 and RQ6, corresponding to 932,382 (91.03%) notebooks. Figure 5 presents the distributions of code cells and maximum execution counter value by executed notebooks for the overall group (a) and popular group (b). Both distributions concentrate at the beginning of their histograms, indicating that notebooks are relatively small both in the number of code cells and in the number of maximum execution counter, compared to the size they can get. However, the mismatch between the number of code cells and the maximum execution counter number—which can be observed both by comparing the median or the visual representation—indicate that notebooks have more executions than cells (i.e., they necessarily have skips). Popular notebooks are even smaller (shorter medians), and the difference between the maximum execution counter and the number of code cells is smaller than that of the overall group as well. It indicates that they have fewer skips. Their execution counter is closer to the code definition.



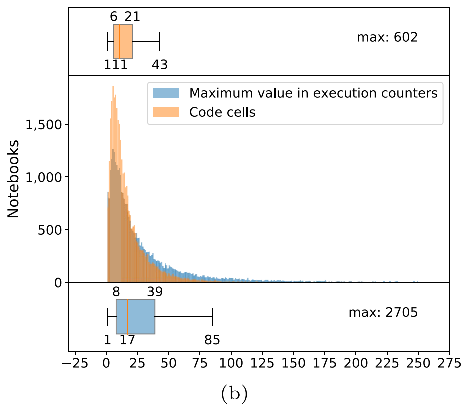


Fig. 5 Distribution of code cells and maximum execution counter for overall group (**a**) and popular group (**b**)

Figure 6 summarizes the corpus of this work. The percentages reported in each research question's analysis refer to the number presented in this corpus unless stated otherwise. While we indicate the number of samples in each group in this figure, we still discuss all the samples in the analyses, since some restrictions do not hold for manual qualitative analyses. For instance, to assess the modularization (RQ3) and tests (RQ4) of notebooks, we do not

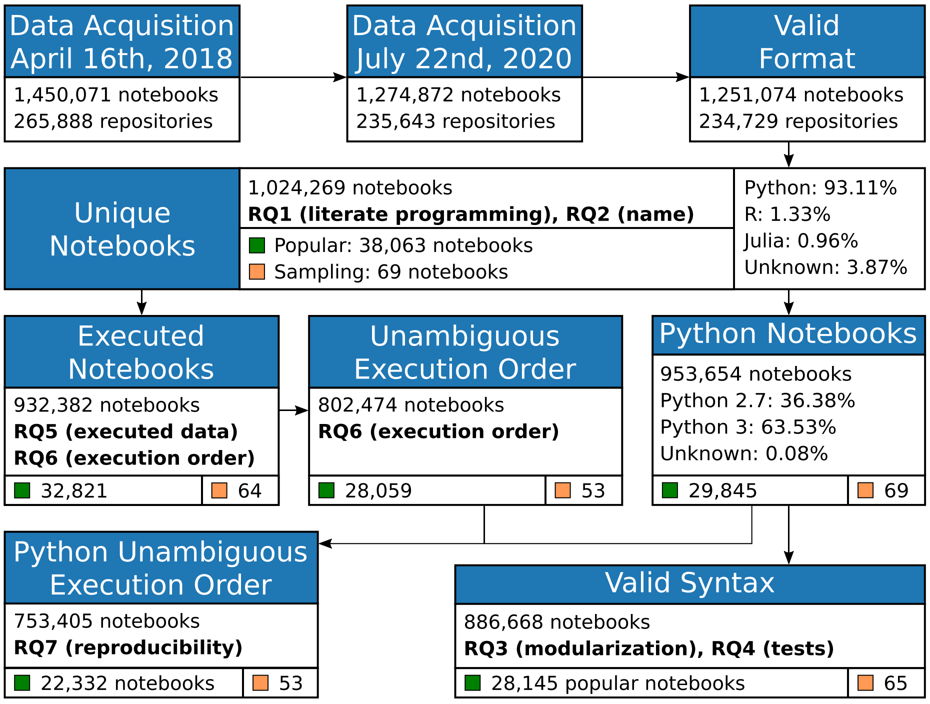


Fig. 6 Notebook corpus and its partitions used in the analyses

need to attain to Python notebooks with valid syntax, since we are not using automatic AST analysis tools.

3.6 Results and Discussion

In this section, we present the results we collected to answer each of the research questions of our study.

3.6.1 RQ1. How do Notebooks Use Literate Programming Features?

Markdown Use An important aspect of literate programming is using natural language for describing the code. In the collected data, notebooks have a median of 4 Markdown cells. As a comparison, they have a median of 13 code cells. Note, however, that 30.84% of the notebooks have no Markdown cell at all.

Popular notebooks have many more Markdown cells (median of 8), and fewer code cells (median of 11). Additionally, they are 59.59% less likely to have no Markdown cells, suggesting that either they are better documented or they have more Markdown cells directly associated with code cells.

Overall, in notebooks with Markdown cells, these cells concentrate at the beginning of the notebooks, as presented in Fig. 7a. The same thing occurs with the popular group, but popular notebooks have a higher percentage of Markdown cells in the middle of the notebook, as presented in Fig. 7b.

Among the 708,393 notebooks that have Markdown cells, 50% have at least 26 non-symbol Markdown lines. Additionally, 50% of the notebooks have at least 168 non-symbol words. We consider non-symbol words all the words that are not part of the Markdown syntax. Similarly, non-symbol Markdown lines are lines that have at least one non-symbol word. We also count stopwords as non-symbol words. Stopwords correspond to a median of 44 words in the notebooks for which we could detect the language. Despite popular notebooks having more Markdown cells, their sizes are smaller: median of 25 non-symbol lines, 118 non-symbol words, and 23 stopwords. It suggests that they are not necessarily better documented. Instead, they use Markdown cells closer to code cells for more direct documentation.

Finally, the most common Markdown elements are headers (H1, H2, and H3), and paragraphs. These elements appear respectively in 90.72% and 79.50% of notebooks with Markdown. Notebooks have a median of 18 words in all headers and 89 words in all paragraphs.

Language We used the *langdetect* library (Danilak 2016) to identify the language of Markdown cells. English appears in the Markdown cells of 87.38% (619,001) of notebooks with Markdown cells. However, only 275,380 are solely in English. We detected other languages in cells of 46.29% (327,908) of the notebooks with Markdown. Besides English, the most popular languages are French, Italian, German, Romanian, Indonesian, Spanish, Norwegian, Portuguese, and Danish, in this order. Additionally, we could not detect the language of cells in 37.88% of notebooks with Markdown. Popular notebooks are 10.67% less likely to use only English.

Sampled Notebooks Among the 69 sampled notebooks, we noted that 68.12% of them use Markdown. As expected, this is close to the percentage in the full dataset (69.16%). Most of these notebooks are written in English, but we found four notebooks in distinct languages:

Chinese, Japanese, Spanish, and Portuguese. Nonetheless, two of these notebooks had code written in English, and the other two did not have any code cell. Instead, they only had Markdown cells defining the problem. We used an external service to translate the languages that we could not understand for analyzing the content of the markdown cells.

Regarding the content, 36 notebooks have a Markdown cell with a meaningful title for the notebook. While we only found this type of title in header elements, not all header elements represent a notebook title. Some headers only separate sub-subjects of the notebook, and others only describe the executed code. In the sample, 38 notebooks use headers and paragraphs for describing the notebook code. Some descriptions in the Markdown have a goal that is different from describing what the code does: 29 notebooks use the Markdown to describe a problem and/or goal; 16 notebooks present tasks for students, reinforcing the educational aspect of notebooks; and 13 notebooks have a markdown cell concluding the notebook with findings, directions for the next steps, or limitation discussion.

Figure 8 presents a snippet of a sampled notebook. This notebook starts with a Markdown cell with the title and a brief explanation of the notebook subject. It has sub-headers (e.g., "Training with k-Fold Cross-Validation") separating sections of the notebook, and Markdown cells explaining the code cells. In addition to Markdown cells, this notebook also uses code comments to describe the code.

RQ1. *How do notebooks use literate programming features?*
Answer: Most notebooks have Markdown cells, which is a literate programming characteristic. Moreover, Markdown cells correspond to almost one-fourth of the cells. On the other hand, the text is often short, and the most used elements are simple headers and paragraphs, despite the possibility of displaying lists, images, links, and other formatted elements. In the sample, we observed that notebooks use headers to separate sections and describe code cells. In addition to describing code cells, notebooks also use Markdown to describe the problem/goal they are tackling, describe tasks, and conclude the document.
Possible implications: Markdown plays a considerable role in notebooks, but their usage may not properly reach the literate programming goal of producing well-described narratives, given their sizes and elements. It potentially compromises the understandability of the notebook. Their position indicates that users give more attention to the beginning of notebooks. Additionally, Markdown could provide descriptions on how to reproduce the notebook, such as indicating libraries to install or the execution order. Hence, reproducing the last cells of an average notebook that does not have Markdown cells may represent a challenge. The linting tool we propose in Section 5 attempts to support the goal of producing well-described narratives by suggesting the writing of Markdown cells at the beginning and ending of notebooks for describing and concluding them, respectively.

3.6.2 RQ2. How are Notebooks Named?

Anti-patterns Only 1.93% of the notebooks start with "Untitled", and only 0.68% of the notebooks have "-Copy" in their names. A considerable number of notebooks (26.86%) have characters not recommended by the POSIX fully portable filenames guide. Many of these names do not cause problems for most systems, but 0.08% of the notebooks would not work on Windows. Since we used Linux to clone the repositories, we do not know how

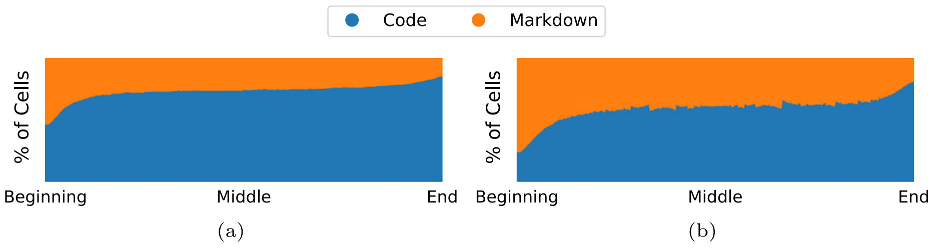
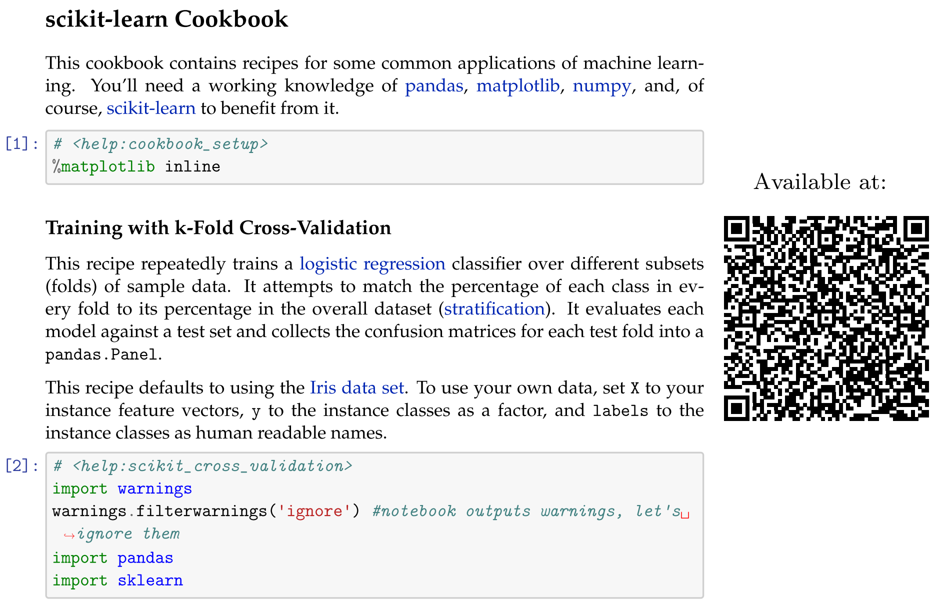


Fig. 7 Distribution of cell types among all notebooks with Markdown (**a**), and popular notebooks (**b**)

many titles Linux does not support, if any. All these anti-patterns are less prevalent in the popular group: notebooks with “Untitled”, “-Copy”, non-recommended POSIX characters, and invalid Windows characters correspond respectively to 0.40%, 0.15%, 20.52%, and 0.03% of the popular notebooks.

We further investigated the repositories of notebooks with invalid names. We observed that 56.49% of the untitled notebooks belong to repositories with two or more untitled notebooks, and 50.79% of them belong to repositories with ten or more notebooks. Similarly, 59.99% of the “-Copy” notebooks belong to repositories with two or more “-Copy” notebooks, and 65.49% of them belong to repositories with ten or more notebooks. This suggests that these anti-patterns are more likely to appear and repeat in repositories with many notebooks. Users can better handle smaller repositories with few notebooks.

Name Length Figure 9 presents the distribution of filename lengths. Note that all notebook names have the extension “.ipynb”. We found 8 notebooks without a title (i.e., their



scikit-learn Cookbook

This cookbook contains recipes for some common applications of machine learning. You’ll need a working knowledge of pandas, matplotlib, numpy, and, of course, scikit-learn to benefit from it.

```
[1]: # <help:cookbook_setup>
     %matplotlib inline
```

Training with k-Fold Cross-Validation

This recipe repeatedly trains a logistic regression classifier over different subsets (folds) of sample data. It attempts to match the percentage of each class in every fold to its percentage in the overall dataset (stratification). It evaluates each model against a test set and collects the confusion matrices for each test fold into a `pandas.Panel`.

This recipe defaults to using the Iris data set. To use your own data, set `X` to your instance feature vectors, `y` to the instance classes as a factor, and `labels` to the instance classes as human readable names.

```
[2]: # <help:scikit_cross_validation>
     import warnings
     warnings.filterwarnings('ignore') #notebook outputs warnings, let's␣
      ↪ignore them
     import pandas
     import sklearn
```

Available at:

Fig. 8 Snippet of IBM/Science/sklearn_cookbook.ipynb from the GitHub repository WatPro/binder-workspace

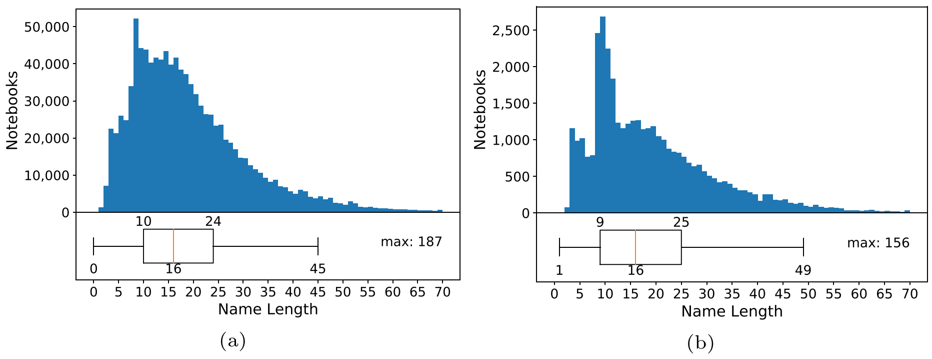


Fig. 9 Distributions of filename lengths in the overall group (**a**) and popular group (**b**)

filename was just ".ipynb"). Excluding the extension, 50% of the notebooks—and popular notebooks—have 16 letters or less. This corresponds to an average of 2 to 3 English words.

Sampled Notebooks In our sample of 69 notebooks, we observed the semantics of the names. We found that most sampled notebooks have a meaningful name (84.06%). We noticed that 11 notebooks have meaningless names, such as "project", "main", "exercise", or a variation with numbers and letters. Additionally, two notebooks had "-Copy", which removes some meaning by allowing multiple notebooks to share the same name. Observing the notebook's context in the repository, we also noticed 20 notebooks (28.99%) with numbered names. These numbers most likely define the order of execution of the notebooks.

RQ2. *How are notebooks named?*
Answer: Most users seem to change the default name in the titles of their committed notebooks and use meaningful but short names. On the other hand, many users do not seem to be concerned about OS-based restrictions and conventions in naming files. In the sample, we observed that some repositories define a sequence of execution for the notebooks.
Possible implications: Although the title is important for the narrative, disregarding OS-based naming constraints may hamper the reproducibility when using other operating systems. Julynter—proposed in Section 5—has seven linting suggestions that aim at supporting both the narrative and OS-based naming constraints. In addition to these constraints, the sequencing in the naming scheme is important both for the reproducibility (e.g., executing a notebook that depends on data generated by a previous notebook in the sequence) and for the literate aspect of the notebooks (e.g., executing a notebook that deepens on the explanation of a concept that was introduced in previous notebooks).

3.6.3 RQ3. How do Notebooks Use Modules, Functions, and Classes?

Imports To answer this question, we analyzed the AST of all 886,668 valid Python notebooks. While 91.43% of them had imports, only 10.41% of them had local imports (i.e., imports of modules defined in the repository directory). The popular group percentages are close: 90.19% of notebooks with imports and 10.93% with local imports. Figure 10 presents

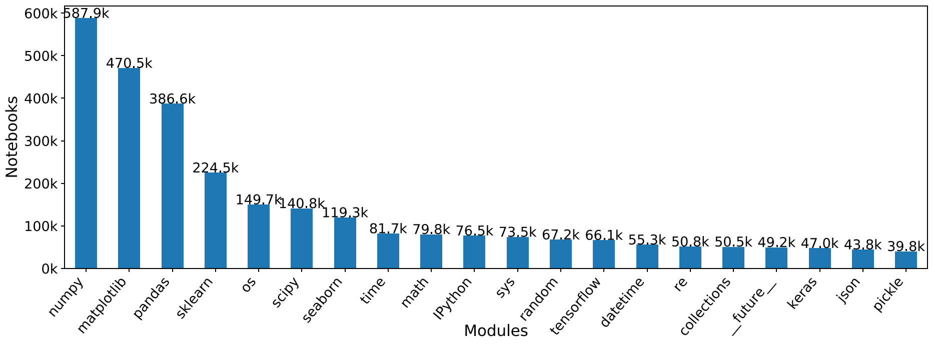


Fig. 10 Top 20 most imported modules

the top 20 most imported modules. The most used modules are *numpy*, *matplotlib*, and *pandas*, which are modules related to scientific software and data analytics. Built-in Python modules also appear among the top 20, but in a much lower number of notebooks. Figure 11 presents the distribution of cells with imports in notebooks. Note that most imports occur at the beginning of the notebooks. In Python scripts, the official Python style guide (PEP 8) recommends writing imports at the top of the files (van Rossum et al. 2001).

Constructs Next, we analyzed the AST constructs from notebooks to understand if they define functions and classes. Figure 12 presents the used AST constructs from valid Python notebooks. Note that only 54.28% of valid Python Notebooks define functions, and only 8.55% define classes. The small number of classes is expected: Python is a multi-paradigm programming language that encourages starting the code as a simple functional or imperative script and evolving object-oriented code from it (Vavrová and Zaytsev 2017). While the percentage of notebooks with functions may indicate that notebooks discourage writing functions, we found that 71.02% of notebooks with loops or conditional structures also have function definitions. Hence, notebooks without functions may be simple enough, not requiring this abstraction. We did not observe a big difference in the distribution of constructs in the popular group.

Sampled Notebooks Regarding the 69 sampled notebooks, we observed that 86.96% of them have imports in the first cells. Despite condensing most imports in the beginning, 42.03% of the notebooks also have imports in the middle, indicating that the users did not have a strict concern about the position of their imports. Moreover, 7.25% only have imports in the middle. In the sample, only 44.93% of the notebooks concentrated all imports in the

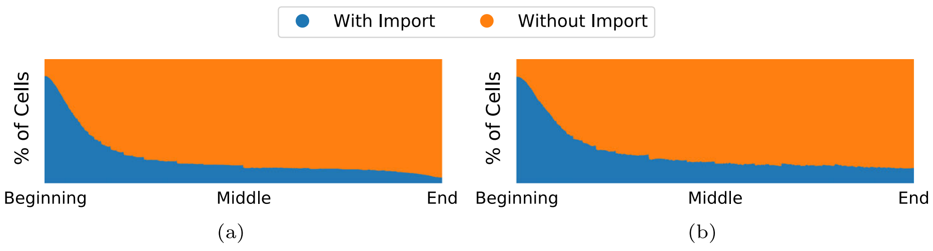


Fig. 11 Distribution of cells with imports in valid Python notebooks (**a**) and popular notebooks (**b**)

beginning, but even in this case, some of them did not use only the first cell: 10.14% have imports up to the second one, and 1.45% have imports up to the third cell. Some of these notebooks spread imports through the first cells. Others use the first cell for initialization code, such as constant definitions, bang expressions to install external libraries, and IPython magics to load extensions.

We also found that 79.71% of the notebooks are mainly designed to orchestrate library functions, which themselves perform heavy processing. It is expected, as Python is not a fast language for processing-intensive tasks. These notebooks often use loops to convert a data format into another compatible with the tools they are using. Only 12 notebooks of the sample (17.39%) implement their own intensive processing. While 50.72% of the notebooks in the sample had function definitions, the only pattern we could identify is that the function definitions usually appear near the cells that use them.

Figure 13 presents a snippet of one of the sampled notebooks. This notebook loads the data into a *pandas DataFrame*, and most of its operations are an orchestration of *pandas* and the other imported libraries (not visible in the snippet). The snippet shows the cell `In [13]` with loops. However, these loops do not constitute the heavy processing operations of the notebook. Instead, the main goal of these loops in the cell is to convert the data from a format to another and add it back to the *pandas DataFrame* for other orchestration operations. In our sampled notebooks, most loops had a similar goal of transforming the data format or feeding the orchestrated libraries with data. Few notebooks used loops for heavy processing.

RQ3. *How do notebooks use modules, functions, and classes?*
Answer: On the one hand, users seem to create functions in notebooks that have more complex code with control flow constructs. On the other hand, users do not seem to extract functions to local modules, given the fewer number of notebooks with local modules. Class definitions are indeed rare, but it may be a consequence of the multi-paradigm design of Python.
Possible implications: While defining functions and classes inside notebooks achieves the benefits of reusability and abstraction, these benefits are limited to internal use of the notebook. Local modules could be better explored to extend the reusability to other notebooks and scripts, and reduce the size of code cells in notebooks. However, keeping the code inside the notebook can be good for reproducibility, as it allows users to share only the notebook file with all code.

3.6.4 RQ4. How are Notebooks Tested?

Tests Only 13,894 (1.57%) valid Python notebooks import known testing modules or modules that have “test”, “Test”, “TEST”, “mock”, “Mock”, “MOCK” as a sub-string of their names. The most imported testing module is *problem_unittests*, which is a local module from a deep-learning course that has been forked 4,293 times at the time of this writing (Udacity 2017). Note that we excluded duplicated notebooks. Thus, all the notebooks that import this file have a distinct source code. The second most imported testing module is *unittest*, which is the built-in Python module for unit testing. The percentage of valid popular Python notebooks that import testing modules is very close: 1.62%.

While some tools were designed to support tests on Jupyter (Burns and Ward 2013; Pimentel 2016), we could not detect many uses of these tools. Moreover, they require modifying the notebook code in a way that may break the narrative.

The reason why we found very few notebooks with tests may not be the notebook environment. Instead, it may be the domain of the applications that people develop with Jupyter. As observed in the sample in Section 3.4, and discussed in Section 3.6.3, Jupyter Notebooks are mainly used for scientific software and data analytics. Testing this kind of software is hard due to the lack of oracles, and the difficulty of judging the number of tests required to guarantee the correctness (Hook and Kelly 2009). Nonetheless, another reason that explains the lack of tests in notebooks is their usage in exploratory tasks, which are often one-offs. If people are deploying code that will be run multiple times, they may prefer to do so in Python scripts.

Sampled Notebooks Our sample did not have any notebooks with tests. However, we did find tests in eight repositories associated with the notebooks of our sample. Only one of them had tests inside other notebooks but in a subject not related to the sampled notebook. The other repositories had tests in Python files. Three repositories had tests unrelated to the sampled notebook. Two tool repositories that use notebooks to show how to use the tool had tests related to the tool. A repository had a test file that attempted to run the notebook and check if it executes successfully, but it did not check the outputs. Finally, a repository had tests in the same subject of the sampled notebook, but it did not test the notebook's code.

RQ4. *How are notebooks tested?*
Answer: Very few notebooks import testing modules. However, we observed in the sampled notebooks that some repositories attempt to test code related to the notebook outside the notebook environment.
Possible implications: There is an opportunity for improving tests on notebooks. As presented in Section 3.6.3, users already tend to create functions in notebooks that have a more complex code. These are probably the most appropriate abstractions for testing with default testing tools, such as the Python *unittest*. An appropriate test suite is important for assuring the reproducibility in other environments. However, for notebook code that is based on data exploration, the existing tools are not sufficient and too intrusive. It also opens the opportunity for proposing testing approaches for notebooks.

3.6.5 RQ5. Do Users Store Notebooks with Retrospective data?

Outputs As stated in Section 3.2, we collected 932,382 executed notebooks, which corresponds to 91.03% of the unique notebooks. These notebooks have retrospective data.

Among the executed notebooks, 54.31% of the code cells had an output, and 96.20% of the notebooks had at least one cell with an output. Despite having fewer code cells, popular notebooks have more code cells with an output, proportionally (59.66%).

Table 3 presents the percentage of cells and notebooks with each output format for both the set of executed notebooks and executed popular notebooks. Note that a cell can have multiple output formats. Thus, the percentages add up to more than 100%. The same happens for notebooks. In this table, *Text* represents the textual output of cells, and *Stream* represents the output of print statements and exceptions. *Image* represents PNG, JPEG, and

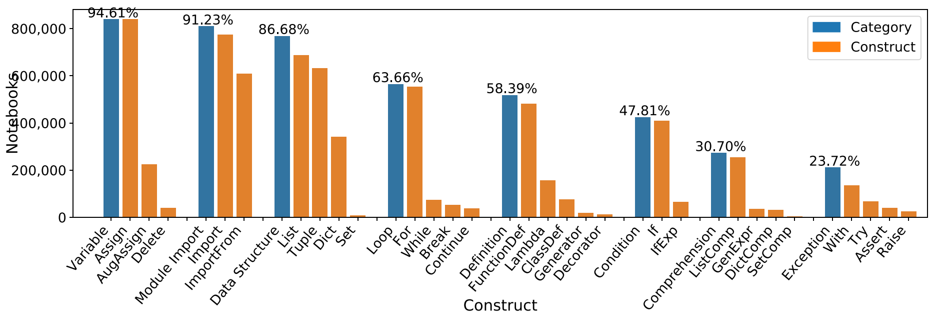
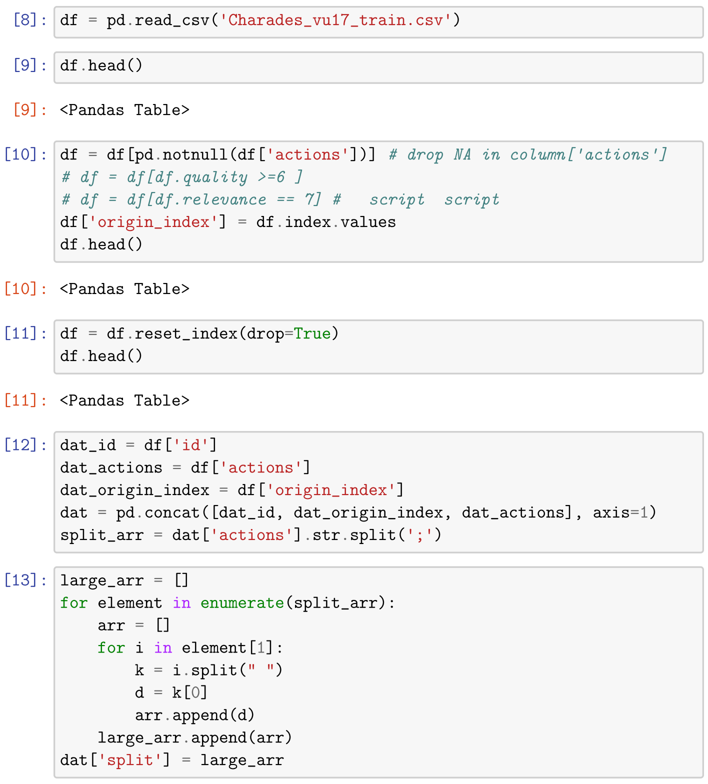


Fig. 12 Distribution of Python constructs in notebooks. This figure groups constructs into categories. The constructs of a category appear on the right of the category bar. A category corresponds to the union of its constructs

SVG formats, which are the default image formats supported by Jupyter. *Formatted* represents *Markdown* and LaTeX formats. Finally, *Extension* combines all extension-specific formats. The most common extension formats are Jupyter Widgets, *plotly*, and *bokeh* formats. Very few notebooks use the extension formats. Note in this table that most executed



```
[8]:  df = pd.read_csv('Charades_vu17_train.csv')

[9]:  df.head()

[9]:  <Pandas Table>

[10]: df = df[pd.notnull(df['actions'])] # drop NA in column['actions']
      # df = df[df.quality >=6 ]
      # df = df[df.relevance == 7] #   script  script
      df['origin_index'] = df.index.values
      df.head()

[10]: <Pandas Table>

[11]: df = df.reset_index(drop=True)
      df.head()

[11]: <Pandas Table>

[12]: dat_id = df['id']
      dat_actions = df['actions']
      dat_origin_index = df['origin_index']
      dat = pd.concat([dat_id, dat_origin_index, dat_actions], axis=1)
      split_arr = dat['actions'].str.split(';')

[13]: large_arr = []
      for element in enumerate(split_arr):
          arr = []
          for i in element[1]:
              k = i.split(" ")
              d = k[0]
              arr.append(d)
          large_arr.append(arr)
      dat['split'] = large_arr
```

Available at:

Fig. 13 Snippet of train_actions_csv.ipynb from the GitHub repository AdrianHsu/charades-parser

Table 3 Output formats in cells and notebooks. Note that a cell can have multiple output formats, thus, the percentages add up to more than 100%

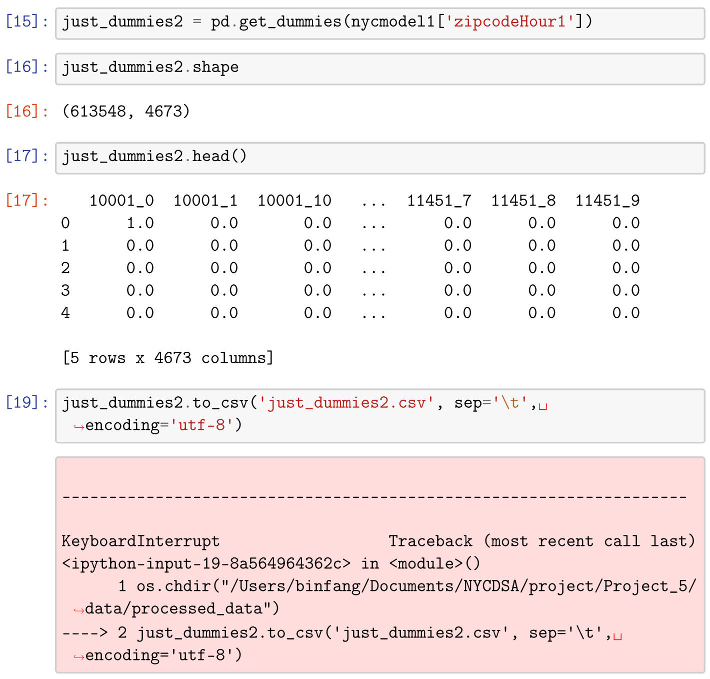
Format	Overall		Popular	
	% of cells with output	% of executed notebooks	% of cells with output	% of executed notebooks
Text	68.11%	82.00%	58.48%	67.21%
Stream	36.08%	70.47%	44.32%	65.28%
Image	22.67%	51.69%	19.84%	43.61%
HTML/JS	16.23%	36.92%	12.08%	26.98%
Error	2.28%	14.86%	1.15%	6.92%
Formatted	1.22%	1.91%	1.31%	1.75%
Extension	0.44%	1.56%	0.32%	0.97%
PDF	0.08%	0.12%	0.08%	0.11%

notebooks have outputs in cells. Note also that despite having proportionally more executed cells with outputs, the popular group has a smaller percentage of all output formats. It might indicate that cells in the overall group tend to have multiple outputs at once, while cells in the popular group tend to be more focused on a smaller number of outputs.

Sampled Notebooks In the 69 sampled notebooks, only five notebooks (7.25%) do not have execution data. Two of them are the notebooks mentioned before that only have tasks descriptions in the Markdown. In the notebooks with retrospective data, we found eight types of output data: 53.62% of the notebooks with results on cell outputs, 5.80% with accidental results on cell outputs (e.g., functions that configure *matplotlib* plots returning objects at the end of cells whose main goal is to display plots), 72.46% with stream outputs, 46.38% with images, 43.48% with tables, 21.74% with warnings, 14.49% with exceptions, and 7.25% with interactive components that did not load without re-executing the notebooks. Some of these components use HTML and JS in the output, while others use extensions. Finally, we also identified that 27.54% of the notebooks write files in addition to the usual notebook output.

Note that these categories are somewhat different than the ones we reported in Table 3. This mainly happens because we analyzed the outputs in the sampled notebooks using the Jupyter Lab interface instead of reading their JSON representations. It leads to two major consequences. First, as humans, it is easier for us to visually identify that a cell has a table than to identify that a cell outputs HTML to display the table—everything is HTML in the Jupyter Lab interface. Similarly, we can easily identify whether a stream output is a warning message or just the result of a print statement. Second, in some situations, we are only able to identify one output type, despite a given cell generating at the same time multiple outputs (e.g., a *pandas* table has both an HTML representation and a text representation). This happens because Jupyter only displays the most appropriate for the application in such situations but stores both results in the notebook file.

Figure 14 presents a snippet of one of the sampled notebooks that has a cell with no output (`In [15]`), and cells that output a text string at `Out[16]`, an HTML table at



```
[15]: just_dummies2 = pd.get_dummies(nycmodel1['zipcodeHour1'])

[16]: just_dummies2.shape

[16]: (613548, 4673)

[17]: just_dummies2.head()

[17]:    10001_0  10001_1  10001_10  ...  11451_7  11451_8  11451_9
      0      1.0      0.0       0.0  ...      0.0      0.0      0.0
      1      0.0      0.0       0.0  ...      0.0      0.0      0.0
      2      0.0      0.0       0.0  ...      0.0      0.0      0.0
      3      0.0      0.0       0.0  ...      0.0      0.0      0.0
      4      0.0      0.0       0.0  ...      0.0      0.0      0.0

      [5 rows x 4673 columns]

[19]: just_dummies2.to_csv('just_dummies2.csv', sep='\t',␣
       ↪encoding='utf-8')

      ---------------------------------------------------------------------

      KeyboardInterrupt                  Traceback (most recent call last)
      <ipython-input-19-8a564964362c> in <module>()
            1 os.chdir("/Users/binfang/Documents/NYCDSA/project/Project_5/
       ↪data/processed_data")
      ----> 2 just_dummies2.to_csv('just_dummies2.csv', sep='\t',␣
       ↪encoding='utf-8')
```

Available at:

Fig. 14 Snippet of pythoncode/improvedlm.ipynb from the GitHub repository poorbaby/Predict-New-York-Taxi-Demand

`Out [17]`, and an error at `Out [19]`. It is common to find notebooks containing multiple output formats.

RQ5. *Do users store notebooks with retrospective data?*
Answer: Most notebooks store cells with outputs.
Possible implications: This result fosters reproducibility. Knowing the expected output allows users to re-run notebooks and check if they reproduce the results. However, notebooks may store results from distinct executions, and the lack of session identification may hamper the reproducibility check. Julynter (see Section 5) suggests re-executing cells with results from previous sessions during the newest one to avoid this issue.

3.6.6 RQ6. How are Notebooks Executed?

Executed Notebooks Among the 932,382 executed notebooks, 21.11% had non-executed code cells, and 62.14% had empty cells. Figure 15 presents the distribution of code cells in the notebooks. Note that the percentage of executed code cells drops towards the bottom of notebooks, while the percentage of non-executed and empty cells grows. While 59.15% of executed notebooks finish with empty cells, only 11.35% of executed notebooks have empty cells among non-empty ones. Popular notebooks are 38.43% less likely to have non-executed cells, 38.80% less likely to have empty cells in the end, and 56.14% less likely to have empty cells among non-empty ones.

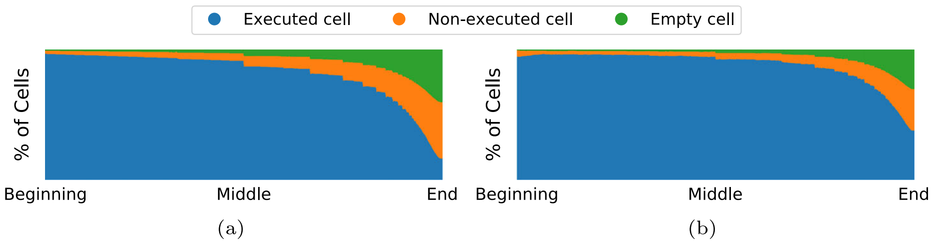


Fig. 15 Distribution of code cells in executed notebooks (**a**) and popular notebooks (**b**)

Unambiguous Execution Order We collected 802,474 notebooks with *unambiguous execution order* (i.e., the ones that neither have repeated values in execution counters nor executing cells, marked with an asterisk). This number corresponds to 86.07% of the executed notebooks. Among the notebooks with unambiguous execution order, 36.45% have out-of-order cells. The percentage of unambiguous notebooks in the popular group is very close (85.49%), but only 22.37% of them have out-of-order cells.

By following the execution counters' sequence in unambiguous execution order notebooks, we counted how many skips occurred. Since skips represent cell executions without explicit definitions, they may indicate the presence of hidden states. Figure 16 presents the distribution of skips by notebooks. 76.88% of unambiguous execution order notebooks have at least one skip. A skip contains 12.83 executions on average. By considering only skips in the middle (i.e., excluding skips in the first cell), the percentage of notebooks with skips drops to 66.15%. Additionally, the average of skipped executions drops to 10.33. As expected, all these numbers drop as well for popular notebooks: 57.54% of them have skips, and 47.84% of them have skips in the middle. A skip contains 9.84 executions on average, or 8.31 executions when we only consider skips in the middle.

Sampled Notebooks Among the 69 sampled notebooks, we found 28 unordered notebooks for exploratory reasons (40.58%), such as updating plots, reloading data, or changing the algorithm. Among them, 19 notebooks had cells defining names (i.e., variables and functions) executed after the cells that use them, leading to non-reproducible notebooks. Moreover, 15.94% of the notebooks had ambiguous execution order due to the repetition of cell numbers, making it hard to execute them. In some cases, the repetition occurred following the top-down order, indicating an attempt to re-execute the notebook in a new session

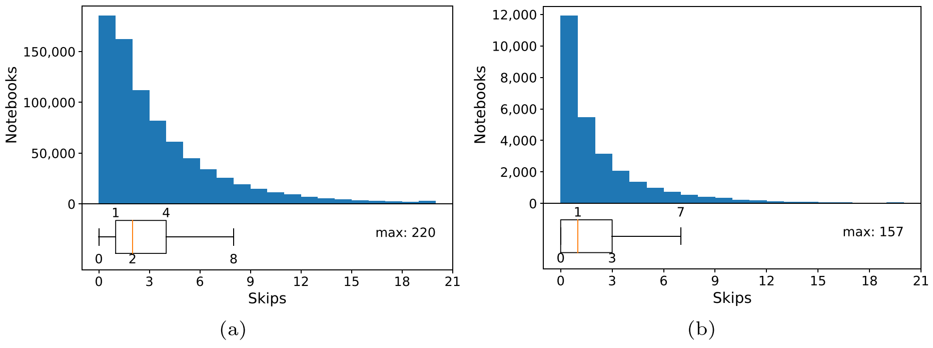
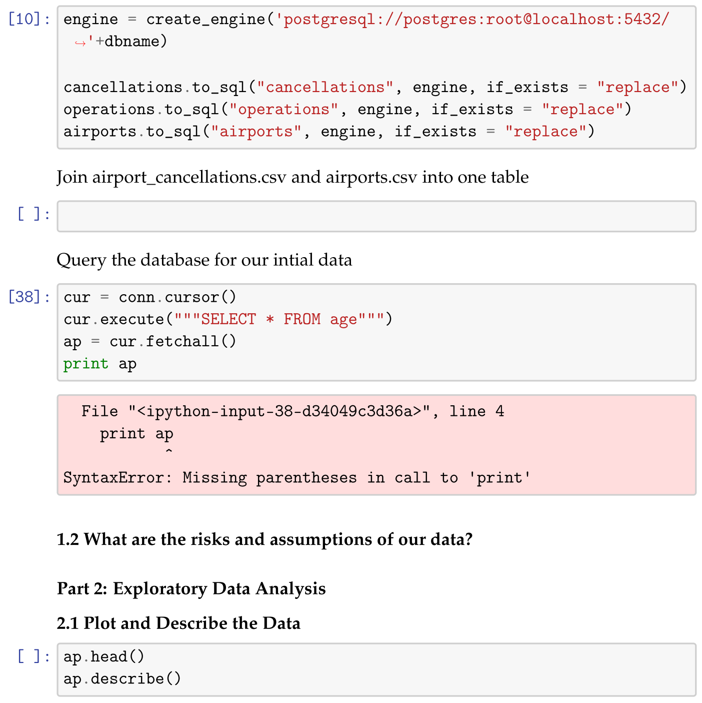


Fig. 16 Distribution of skips in notebooks with unambiguous execution order (**a**) and popular notebooks (**b**)

that did not run all cells. In other cases, the notebook had results from two separate sessions, with one of them executing cells at the beginning and the other executing cells at the end. Some unordered notebooks had repeated cell numbers spread throughout the notebook, making it hard to understand the desired execution intention.

We also observed that 8.70% of the sampled notebooks have cells that were noticeably edited after their execution (e.g., cells with output that could not be generated by that cell code), and 36.23% of the notebooks have non-executed code cells, making it hard to decide which cells should be executed. Nonetheless, 32.00% of these notebooks with non-executed cells had them at the end of the notebook, indicating that the users stopped executing the cells at a given moment. Additionally, 16.00% of these notebooks have non-executed code cells with only code comments. We also found that a notebook had an incomplete code cell that was not executed, and a notebook had non-executed cells after an exception, which might have prevented the user from running the end of the notebook using the *Run all cells* option.

Figure 17 presents a snippet of one of the sampled notebooks. This notebook has an empty cell between two Markdown cells, a skip in the execution count, and non-executed code cells in the end. While the skip in the snippet is from `In [10]` to `In [38]`, the actual skip in the notebook is from `In [17]`, as the notebook has the cells in the wrong order. Having cells in the wrong order is also a source of hidden states in this case: the cell `In [12]` appears at the beginning of the notebook, but it redefines the variables `cancellations`, `operations`, and `airports` used in the `In [10]`, presented in the snippet. Hence, executing this notebook following the cell execution counter would fail.



```
[10]: engine = create_engine('postgresql://postgres:root@localhost:5432/
       ↪'+dbname)

      cancellations.to_sql("cancellations", engine, if_exists = "replace")
      operations.to_sql("operations", engine, if_exists = "replace")
      airports.to_sql("airports", engine, if_exists = "replace")
```

Join airport_cancellations.csv and airports.csv into one table

```
[ ]:
```

Query the database for our intial data

```
[38]: cur = conn.cursor()
      cur.execute("""SELECT * FROM age""")
      ap = cur.fetchall()
      print ap

        File "<ipython-input-38-d34049c3d36a>", line 4
          print ap
                 ^
      SyntaxError: Missing parentheses in call to 'print'
```

1.2 What are the risks and assumptions of our data?

Part 2: Exploratory Data Analysis

2.1 Plot and Describe the Data

```
[ ]: ap.head()
     ap.describe()
```

Available at:

Fig. 17 Snippet of pparker-roach/project_7-SANDBOX.ipynb from the GitHub repository mohsseha/DSI-BOS-students

RQ6. *How are notebooks executed?*
Answer: Many unambiguous execution order notebooks have non-executed code cells, out-of-order cells, and skips in the execution counter. All these characteristics hinder the reasoning about execution states. The number of notebooks with skips and the average size of skips drop when we exclude skips at the beginning of the notebooks. A possible cause for these skips happening only at the beginning of a notebook is the re-execution of all of its cells without restarting the kernel.
Possible implications: There is an opportunity for proposing approaches that measure non-executed code cell, out-of-order cells, and skips as code smells in notebooks, i.e., structures in the code that violate design principles and can negatively impact quality (Garousi and Küçük 2018). Fortunately, most of these code smells are easily fixable by restarting the kernel and executing all cells again before committing. Nonetheless, such an approach could detect out-of-order cells by looking not only to cell numbers but also to variable usages occurring before their definition. In Section 5, we propose Julynter, a linting tool that checks these code smells and suggests fixes.

3.6.7 RQ7. How Reproducible are Notebooks?

Handling Dependencies To answer RQ7, we conducted a reproducibility study in which we attempted to execute all 753,405 Python notebooks with unambiguous execution order. Among these, 94,183 (12.50%) belong to repositories that declared module dependencies (which corresponds to 8.78% of the repositories that have Python notebooks with unambiguous execution order). Proportionally, a higher percentage of popular notebooks belong to repositories that declared dependencies (21.77%), suggesting that popular notebooks have more intention of providing directions for their reproducibility. These repositories correspond to 24.63% of the popular repositories that have Python notebooks with unambiguous execution order.

Among repositories with dependencies, 79.85% use `requirements.txt`, while 45.62% use `setup.py`. Many of these repositories (26.09%) have both `setup.py` files and `requirements.txt` files. Moreover, some repositories even have more than one of these files. In addition to these files, we found 865 notebooks that belong to repositories with `Pipfile`.

Popular notebooks are 7.80% less likely to have `requirements.txt` files, 22.20% more likely to have `setup.py` files, 13.71% more likely to have both, and 50.77% less likely to use `Pipfile`. Using more `setup.py` and less `requirements.txt` may indicate that popular notebooks are part of repositories meant to be redistributed and used together with other projects (e.g., libraries)—in opposite to repositories that define the complete Python environment for their standalone execution. The reason they use less `Pipfile` may be related to their age and the time they needed to become popular, as `Pipfile` is a much more recent system.

Not all dependency declarations are valid. In the first execution mode (shared + execution counter), we attempted to install the dependencies for these notebooks in conda environments. However, the dependencies of 59.30% of the notebooks failed to install. To install the dependencies, we first installed all the `setup.py` files in the repository. Then, we installed the `requirements.txt` files. Finally, we installed the `Pipfile` files. The failure rates for these files were 65.53%, 57.21%, and 60.69%, respectively.

The failure rate for the installation of `requirements.txt` was lower than the other formats. While the `requirements.txt` is a declarative format in which the module version is pinned, the `setup.py` is a generic Python script that supports any flexible installation code. Thus, `setup.py` is more susceptible to errors. In comparison to `Pipfile`, `requirements.txt` is a well-established format that has been used for many years. `Pipfile`, on the other hand, was introduced less than four years ago, and its specification still goes through constant revisions.

The failure rate of `setup.py` was about the same for popular notebooks (64.92%). However, `requirements.txt` and `Pipfile` were less likely to fail: 47.42%, and 40.91%, respectively. The reason it happened may be related to the intention of the users when creating these files. Usually, Python developers create `setup.py` to install libraries and command-line tools (PyPA 2020). This intention does not change according to the repository popularity. However, `requirements.txt` and `Pipfile` have the intention of either describing the dependencies of a complete Python environment or describing the dependencies of an application. We suppose popular repositories may design these files describing only the project dependencies to allow other users to use it, while non-popular repositories may use the `pip freeze` command to describe all the environment dependencies for a `requirements.txt` file and not face issues from other users.

Among the reasons for installation errors, we identified that 29.17% have files that require other unavailable files (e.g., sub-requirements and downloads from unavailable servers), 29.17% have malformed files (i.e., wrong syntax or conflicting dependencies), 25.59% have files that require a previous installation of Python packages (e.g., a `setup.py` that requires Cython to compile and build a package), 19.69% have files that require external tools (e.g., compilers and libraries), 8.43% have files designed for other systems (e.g., Raspberry Pi and Windows), and 0.98% have dependencies that do not support the declared Python version (e.g., the repository has a Python 2 notebook, but the `setup.py` requires a module that dropped support to Python 2 and did not pin the module version). Popular notebooks have fewer errors related to unavailable files (20.28%), malformed files (20.28%), being designed for other systems (5.94%), or having dependencies that do not support the Python version (0.10%), but more errors related to requiring a previous installation of a Python dependency (33.87%) or an external tool (25.09%). It is expected since there was no standard way to define installation dependencies during the development of most of these notebooks. The specification for Python build system requirements was proposed in May 2016 and implemented in March 2017 (Cannon et al. 2016).

We were able to install the dependencies for 40.70% of the notebooks. In addition to these notebooks, we prepared anaconda environments for the notebooks that did not declare dependencies (87.48% of them). Unlike previous conda environments, an anaconda environment comes with a comprehensive set of scientific Python packages, such as *numpy*, *matplotlib*, and *pandas*. Combining both the set of notebooks for which we were able to install the dependencies and the set of notebooks that did not declare dependencies, we had 697,398 notebooks on our original reproducibility study (Pimentel et al. 2019b). The installation success rate for popular notebooks was very close: 41.38%.

Isolating Executions and Exploring Different Execution Orders While the first execution mode used conda environments to isolate the dependency installation, it did not isolate the interactions between notebooks and the OS. Consequently, one execution could interfere with another by changing state in the *shared* OS. To address this limitation, we run notebooks inside *isolated* docker containers. Additionally, the first execution mode was inherited from the original study (Pimentel et al. 2019b), which only executed notebooks by following the order of cell execution counters, which can lead to false negatives with respect to reproducibility assessment. In the current study, we follow both the cell execution counter and the top-down order.

We also prepared *bloated* docker images in which we attempted to install all modules imported by all Python notebooks that we collected. As expected, many installations failed and we left them out of the image. Nonetheless, most popular modules, which were imported by more than 2,000 notebooks in our corpus, were successfully installed. The only exceptions were *GraphLab* (imported by 11,092 notebooks), *PyTorch* (imported as `torch` by 7,745 notebooks and as `torchvision` by 3,433 notebooks), *gensim* (imported by 7,174), and *GeoPandas* (imported by 3,350). *GraphLab* requires a license to use. *PyTorch* did not work in our environment. *GeoPandas* and *gensim* had binary dependencies that we could not install. In addition to these dependencies, we also installed common tools, compilers, and interpreters in all docker containers to reduce the number of failures due to the absence of external tools.

Notebook Executions We report five execution modes by alternating both the execution order and the environment. The first one (shared + execution counter) is the reproducibility study of the first paper, in which we executed notebooks following the cell execution counter in conda and anaconda environments installed in a shared system. The second one (isolated + execution counter) uses the anaconda docker image and runs notebooks following the cell execution counter. The third one (isolated + top-down) also uses the anaconda docker image, but we execute the cells following the top-down execution order. In the fourth one (bloated + execution counter), we use the bloated docker image to execute notebooks following the cell execution counter. Finally, in the fifth mode (bloated + top-down), we also use the bloated docker image to execute in the top-down order. Due to time constraints, we could not install the dependencies from the dependency files in the docker containers. Thus, we restricted the executions in the isolated modes only to notebooks in repositories that did not have dependency files. Additionally, we executed all notebooks in bloated modes without installing specific packages from dependency files of each repository.

In our experiments, many notebooks failed to execute all cells. Some failed because their execution exceeded a time limit of 5 minutes, while others failed due to an exception. Figure 18 presents the percentage of notebooks that failed due to timeout, in addition to the 10 most common exceptions the notebooks presented in each execution mode of our assessment.

By mining association rules related to timeout on the executions of the isolated modes, we found that importing *unittests*, defining “raise” and “while” statements increases the frequency of timeouts by at least 9.81, 2.37 and 1.63 times, respectively, as presented in Table 4.

The bloated docker environments failed much less due to *ImportError* and *ModuleNotFoundError* than the other environments, since these exceptions are related to missing dependencies. On the other hand, these environments failed much more due to *AttributeError*, *KeyError*, and *RuntimeError*. The former two exceptions are related to updates on libraries that deprecate and change APIs, and the latter exception may be related to conflicting versions of libraries. Hence, simply installing dependencies from imports does not solve all dependency problems. In fact, with the growth of these other types of exceptions, the percentage of notebooks that run all cells did not improve much. These issues would be better addressed through the proper definition of dependencies with their versions in dependency files.

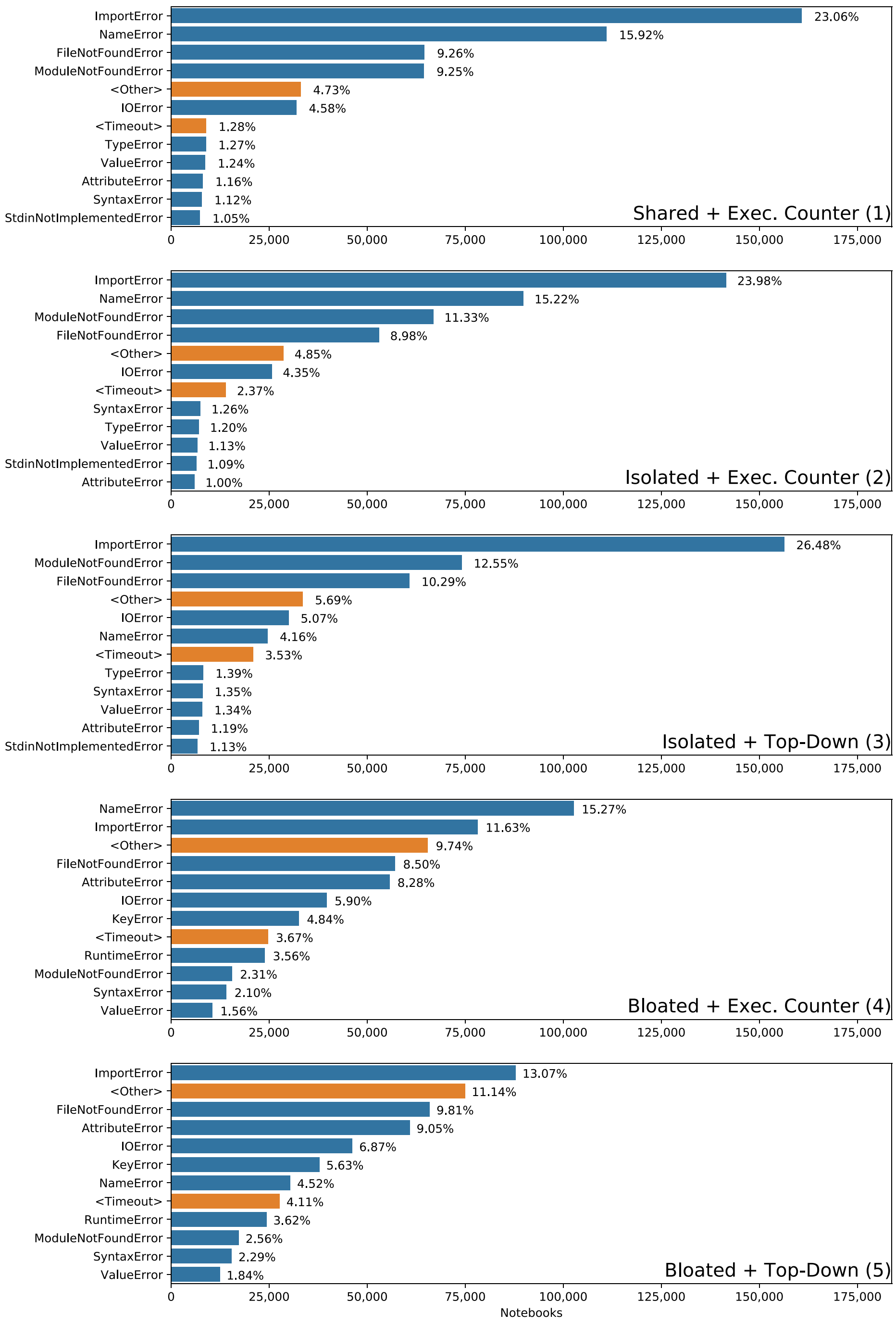


Fig. 18 Failure reasons for the executions in each execution mode. The blue bars represent the Top 10 exceptions. The "Timeout" orange bar represents executions that we stopped when they took 5 minutes to run. The "Other" orange bar groups all the other exceptions that are not part of the Top 10

Table 4 Association rules related to timeout

Mode	Antecedent	Consequent	Support	Confidence	Lift
Isolated + Exec.Counter	unittests	timeout	0.09%	28.80%	11.56
Isolated + Exec. Counter	raise	timeout	0.17%	6.79%	2.73
Isolated + Exec. Counter	while	timeout	0.32%	4.21%	1.69
Isolated + Top-Down	unittests	timeout	0.11%	35.26%	9.81
Isolated + Top-Down	raise	timeout	0.21%	8.50%	2.37
Isolated + Top-Down	while	timeout	0.45%	5.86%	1.63

Surprisingly, in the first execution mode (in which we tried to install declared dependencies), 44.18% of the notebooks from repositories with declared dependencies failed with one of these errors. In contrast, only 31.61% of the notebooks from repositories without declared dependencies failed with these errors. It probably happened because we used anaconda environments with more pre-installed dependencies for the latter ones. Still, it indicates that many dependency files do not declare all the notebook dependencies. Popular notebooks were about 13% more likely to fail in both situations.

Another very common exception in the executions that followed the cell execution counter was *NameError*. This exception occurs when Python tries to access a variable that was not defined. This exception is related to hidden states and out-of-order cells. By mining association rules over features from executions of the isolated modes, we found that skips raise the chance of *NameError* exceptions, as presented in Table 5. Skips increase more the likelihood of exceptions when we run notebooks following the cell execution counter than when we run them in top-down order (1.27 times vs. 1.19 times). Moreover, having skips in the middle raises even more the likelihood of exception in the execution counter order (1.39 times). Since this exception occurred much more on executions that followed the execution counter than on executions that followed the top-down order, this suggests that even though

Table 5 Association rules related to skips and *NameError*

Mode	Antecedent	Consequent	Support	Confidence	Lift
Isolated + Exec. Counter	Skips in the middle	NameError	13.41%	20.32%	1.39
Isolated + Exec. Counter	Skips	NameError	14.21%	18.50%	1.27
Isolated + Top-Down	Skips in the middle	NameError	3.05%	4.62%	1.18
Isolated + Top-Down	Skips	NameError	3.56%	4.63%	1.19

users can execute the notebook at any other and define and redefine variables, they still tend to define variables in the top-down order.

Finally, the other very common exceptions were *FileNotFoundError* and *IOError*. These errors occur when users use absolute paths to access data files or do not include the data in the repositories.

Reproducibility Results. Table 6 presents the reproducibility results for each execution mode, considering each normalization described in Tables 2 and 7 presents the reproducibility results for the popular notebooks. The normalization columns of the first execution mode (shared + execution counter) are empty because we originally did not apply any normalization. In these tables, the percentages refer to the number of notebooks that we attempt to run (i.e., we exclude failed installations in the first execution mode and notebooks with dependency descriptors in isolated modes). Also, we considered notebooks that resulted in a timeout and notebooks with exceptions as non-reproducible. This last assumption may not be true since reproducible notebooks could have exceptions in them.

In Table 6, we calculated that about 11% of the executed notebooks (or 6% in Table 7 for the popular group) originally had exceptions, and about 7% of the notebooks (popular group: 4% in Table 7) had cells with outputs after the original exception. While we counted them as non-reproducible, these exceptions are likely the expected behavior of these notebooks. However, as we did not compare the exceptions, we cannot indicate whether they are reproducible.

The percentage of executions that run all cells ranged from 22.57% to 26.09%. These results are very close to the reproducibility rate of 24.9% that Collberg et al. (2014) achieved in their study of reproducibility in general computer systems research. Their study only attempted to compile the source code and did not check the execution results. In our case, the shared + execution counter mode was able to run more notebooks (26.09%) than the other modes, probably due to the installation of dependency files. Additionally, the bloated + execution counter mode had the smallest percentage of notebooks that run all cells (22.57%). However, these results do not reflect the number of notebooks that produce the same results.

The worst reproducibility rate was observed for the first execution mode (shared + execution counter) with no normalization (4.90%) and the best rate occurred for the bloated + top-down mode after the image normalization (15.04%), as expected. The image normalization not only applies all the previous normalizations shown in Table 2, but it is also the most susceptible to false positives, as it ignores differences in image results by considering that these differences could be caused by small changes in module updates. The boated + execution counter mode had the second-best results (14.40%, after the image normalization).

Popular notebooks were more reproducible in all situations. The percentage of executions that run all cells ranged from 31.84% to 36.27%. In this case, the isolated + top-down mode could run more notebooks than the other environments, proportionally (36.27%). However, in terms of producing the same results, the bloated + top-down mode dominated all the others, reaching a reproducibility rate of 21.32% after the image normalization.

While the normalizations did not affect the ability to run notebooks, they almost doubled the reproducibility rate (compared to the scenario when no normalization was applied). However, not all normalizations were equally effective. The most effective normalizations were image, execution counter, dataframe, deprecation, and stream. The deprecation normalization had more effect than the dataframe one on the environments with all dependencies. This is probably due to the fact that we installed the most recent version of packages in the bloated environments, increasing the chance of having deprecations in them.

Table 6 Reproducibility results for all notebooks

	Shared + Exec. Counter (1)	Isolated + Exec. Counter (2)	Isolated + Top Down (3)	Bloated + Exec. Counter (4)	Bloated+ Top Down (5)
Attempted executions	697,398	590,354	590,358	672,232	672,235
Run all cells	181,955 (26.09%)	137,208 (23.24%)	152,555 (25.84%)	151,730 (22.57%)	170,949 (25.43%)
Stopped by timeout	8,903 (1.28%)	13,969 (2.37%)	20,847 (3.53%)	24,690 (3.67%)	27,652 (4.11%)
Stopped by exception	506,539 (72.63%)	439,177 (74.39%)	416,956 (70.63%)	495,296 (73.68%)	473,324 (70.41%)
Had exception originally	80,520 (11.55%)	67,762 (11.48%)	66,226 (11.22%)	76,151 (11.33%)	74,816 (11.13%)
Output after exception	55,138 (7.91%)	46,165 (7.82%)	43,459 (7.36%)	52,042 (7.74%)	49,287 (7.33%)
Same results					
No normalization	34,148 (4.90%)	29,927 (5.07%)	33,555 (5.68%)	58,365 (8.68%)	58,910 (8.76%)
Encode		29,927 (5.07%)	33,555 (5.68%)	58,365 (8.68%)	58,910 (8.76%)
Execution counter		39,976 (6.77%)	44,618 (7.56%)	70,050 (10.42%)	71,873 (10.69%)
Stream		42,306 (7.17%)	47,274 (8.01%)	72,449 (10.78%)	74,410 (11.07%)
Dictionary		42,306 (7.17%)	47,426 (8.03%)	72,449 (10.78%)	74,410 (11.07%)
Dataframe		45,773 (7.75%)	51,357 (8.70%)	75,708 (11.26%)	77,990 (11.60%)
Exception path		45,773 (7.75%)	51,415 (8.71%)	75,708 (11.26%)	77,990 (11.60%)
Deprecation		48,138 (8.15%)	54,058 (9.16%)	79,763 (11.87%)	82,370 (12.25%)
White space		48,138 (8.15%)	54,593 (9.25%)	79,763 (11.87%)	82,370 (12.25%)
Decimal		48,138 (8.15%)	55,161 (9.34%)	79,763 (11.87%)	82,370 (12.25%)
Date		48,138 (8.15%)	55,174 (9.35%)	79,763 (11.87%)	82,370 (12.25%)
Time		48,138 (8.15%)	55,183 (9.35%)	79,763 (11.87%)	82,370 (12.25%)
Memory		48,138 (8.15%)	55,404 (9.38%)	79,763 (11.87%)	82,370 (12.25%)
Image		64,214 (10.88%)	76,745 (13.00%)	96,783 (14.40%)	101,078 (15.04%)

Table 7 Reproducibility results for the popular group

	Shared + Exec. Counter (1)	Isolated + Exec. Counter (2)	Isolated + Top Down (3)	Bloated + Exec. Counter (4)	Bloated + Top Down (5)
Attempted executions	19,473	13,842	13,842	17,411	17,411
Run all cells	6,864 (35.25%)	4,806 (34.72%)	5,021 (36.27%)	5,543 (31.84%)	5,858 (33.65%)
Stopped by timeout	172 (0.88%)	280 (2.02%)	363 (2.62%)	639 (3.67%)	679 (3.90%)
Stopped by exception	12,437 (63.87%)	8,756 (63.26%)	8,458 (61.10%)	11,224 (64.46%)	10,869 (62.43%)
Had exception originally	1,201 (6.17%)	861 (6.22%)	843 (6.09%)	1,046 (6.01%)	1,032 (5.93%)
Output after exception	873 (4.48%)	616 (4.45%)	584 (4.22%)	758 (4.35%)	727 (4.18%)
Same results					
No normalization	2,135 (10.96%)	1,610 (11.63%)	1,652 (11.93%)	2,590 (14.88%)	2,609 (14.98%)
Encode		1,610 (11.63%)	1,652 (11.93%)	2,590 (14.88%)	2,609 (14.98%)
Execution counter		1,769 (12.78%)	1,821 (13.16%)	2,769 (15.90%)	2,801 (16.09%)
Stream		2,097 (15.15%)	2,142 (15.47%)	2,907 (16.70%)	2,935 (16.86%)
Dictionary		2,097 (15.15%)	2,146 (15.50%)	2,907 (16.70%)	2,935 (16.86%)
Dataframe		2,189 (15.81%)	2,243 (16.20%)	2,999 (17.22%)	3,029 (17.40%)
Exception path		2,189 (15.81%)	2,243 (16.20%)	2,999 (17.22%)	3,029 (17.40%)
Deprecation		2,249 (16.25%)	2,310 (16.69%)	3,098 (17.79%)	3,134 (18.00%)
White space		2,249 (16.25%)	2,332 (16.85%)	3,098 (17.79%)	3,134 (18.00%)
Decimal		2,249 (16.25%)	2,343 (16.93%)	3,098 (17.79%)	3,134 (18.00%)
Date		2,249 (16.25%)	2,343 (16.93%)	3,098 (17.79%)	3,134 (18.00%)
Time		2,249 (16.25%)	2,343 (16.93%)	3,098 (17.79%)	3,134 (18.00%)
Memory		2,249 (16.25%)	2,347 (16.96%)	3,098 (17.79%)	3,134 (18.00%)
Image		2,678 (19.35%)	2,846 (20.56%)	3,641 (20.91%)	3,712 (21.32%)

Since the execution counter normalization is among the ones that affected the most the reproducibility rates without adding false positives, we decided to use it when mining association rules. The association rules indicate that small notebooks, "while" definitions, and "class" definitions raise the probability of obtaining the same results by 122%, 83%, and 67% when following the execution counter order, and 107%, 81%, and 65% when following the top-down order, respectively. On the other hand, we found that skips in the middle, imports, unordered cells, and big notebooks decrease the chance of obtaining the same results by 36%, 37%, 66%, and 70% when following the execution counter order, and 27%, 34%, 41%, and 64% when following the top-down order, respectively, as presented in Table 8.

RQ7. *How reproducible are notebooks?*
Answer: We were able to successfully run between 22.57% and 26.09% of the notebooks that we attempted to run. This number is close to the results of a previous reproducibility study (Collberg et al. 2014) about general computer systems research (24.9%). However, the rates are way smaller (4.90%–15.04%) when we count only notebooks that produce the same results. The most common causes of failures were related to missing dependencies, the presence of hidden states and out-of-order executions, and data accessibility in all execution sets. In the experiments that we used docker environments with most pre-installed dependencies, many executions also failed due to incompatible versions of dependencies and conflicts.
Possible implications: While the reproducibility rate is comparable to the rate in general computer systems research (Collberg et al. 2014), it is far from ideal. The identification of the root causes suggests that there is an opportunity to improve the reproducibility rate in notebooks by devising approaches that address these problems. More specifically, managing the dependencies of notebooks and guaranteeing the linear (top-down) execution order could improve the reproducibility rate. It is worthy of noting that dependency resolution problems are also common in other contexts, such as building past snapshots of software (Tufano et al. 2017). Additionally, tools such as ReproZip (Chirigati et al. 2016) can automatically capture dependencies (both libraries and data) and create packages including these dependencies, thus ensuring reproducibility. ReproZip has a plugin for Jupyter (ReproZip 2017). Similarly, Julynter—proposed in Section 5—detects imports of modules that are not declared in `requirements.txt` files and suggests their inclusion. Additionally, Julynter also detects out-of-order cells and suggests reordering or re-executing the notebook to guarantee a linear execution order.

3.7 Threats to Validity

This first study attempts to obtain a picture of quality and reproducibility practices used in the design of Jupyter Notebooks. As presented in Section 3.1, we have designed measures that capture different aspects of notebooks that impact their reproducibility. These measures, however, have some threats to validity that we discuss below.

Internal While we used clean conda environments in the first execution mode (shared + execution counter), we did not isolate the executions in the system. It means that a notebook execution or dependency installation could install or modify system dependencies before

Table 8 Association rules related to executions that generate the same results after the execution counter normalization

Mode	Antecedent	Consequent	Support	Confidence	Lift
Isolated + Exec. Counter	9 or less cells (1st quartile)	Same Results	3.84%	14.47%	2.22
Isolated + Exec. Counter	while	Same Results	0.91%	11.95%	1.83
Isolated + Exec. Counter	class	Same Results	0.88%	10.94%	1.67
Isolated + Exec. Counter	Skips in the middle	Same Results	2.77%	4.19%	0.64
Isolated + Exec. Counter	Imports	Same Results	3.66%	4.11%	0.63
Isolated + Exec. Counter	Unordered	Same Results	0.80%	2.23%	0.34
Isolated + Exec. Counter	37 or more cells (4th quartile)	Same Results	0.44%	1.98%	0.30
Isolated + Top-Down	9 or less cells (1st quartile)	Same Results	4.00%	15.09%	2.07
Isolated + Top-Down	while	Same Results	1.00%	13.16%	1.81
Isolated + Top-Down	class	Same Results	0.96%	12.02%	1.65
Isolated + Top-Down	Skips in the middle	Same Results	3.48%	5.28%	0.73
Isolated + Top-Down	Imports	Same Results	4.29%	4.81%	0.66
Isolated + Top-Down	Unordered	Same Results	1.55%	4.31%	0.59
Isolated + Top-Down	37 or more cells (4th quartile)	Same Results	0.59%	2.62%	0.36

the preparation and execution of another notebook. We attempted to minimize this threat by running the new analyses in isolated docker environments. However, in the new analyses, we did not attempt to install the dependencies declared in the repositories, due to time constraints. Instead, we did try to install all modules imported by the notebooks in separate environments.

Additionally, we examined all notebooks from GitHub as valid subjects in this work. We did not account for all the perils of mining software repositories from GitHub (Kalliamvakou et al. 2014). Some analyzed notebooks may not be intended to be reproducible and may not value quality. For instance, students prepare exercises with the goal of studying for a course. These exercises have a short life-span and are often not classified as engineered software projects (Munaiah et al. 2017). A basic check for notebooks containing words related to exercises ("assignment", "course", "exercise", "homework", "lesson") returns 164,463 unique notebooks (16.06%). Even though this check is very susceptible to false positives and false negatives, it indicates that exercises are a solid use case for notebooks and deserve investigations. Other use cases for notebooks (e.g., tutorial notebooks, research notebooks, dashboards, and others) may also have different goals in terms of quality and reproducibility and also require further investigations. In the sampled notebooks, we observed that education is a big use case for notebooks. Hence, even though these notebooks may have different goals in terms of quality and reproducibility, it is still worth it to understand them to improve these aspects.

Moreover, during sampling, we manually analyzed the characteristics of the notebooks. This analysis is subject to human error. We attempted to mitigate this threat by comparing some results with proxies on the database. However, these proxies are not complete (i.e., there are things that we only observed in the sample) nor reliable for qualitative analyses (i.e., they do not capture nuances that we could interpret by reading the notebook).

Construct The methods we use to answer the research questions aim to attain an approximated answer since it is not possible to get accurate answers that precisely represent all notebooks without false positives and false negatives. For instance, a module for statistical tests could have "test" in its name and appear as an answer to **RQ4** without being a module for testing software. Similarly, we may not detect a testing module that does not have "test" or "mock" in its name, and that does not appear in the Python testing tools taxonomy (Python-Wiki 2019).

Moreover, in the reproducibility study, we did not consider the maintainability of notebooks and libraries. Many libraries might have been updated since the notebooks were originally developed. This should be a threat for the bloated execution modes, which uses arbitrary versions of the libraries. However, we found that these modes were more reproducible than the shared environment, which attempted to install pinned versions declared in dependency files (`setup.py`, `requirements.txt`, and `Pipfile`). Similarly, many repositories may have been updated to account for library changes since we first collected them. However, when assessing the maintainability of repositories with notebooks, we found that only 12.68% of them still had some active development six months after the collected commit, and only 2.84% of them were still active in the six months prior to the moment we queried GitHub again (July 22nd, 2020). Hence, it is reasonable to assume that most repositories are not maintained, we perform the reproducibility study as is.

Additionally, we only checked whether the notebooks generated the same results when they successfully ran all cells. However, we stopped the executions on exceptions and did not consider these notebooks as reproducible. An exception may be the expected (although unusual) behavior of a notebook, and it may have executed code after the exception as well.

To account for small deviations in the notebooks' results that were leading to false negatives in the analyses of same results, we performed normalizations on the outputs. While some normalizations reduce the number of false negatives without drawbacks (e.g., encode normalization and execution counter normalization), other normalizations increase false positives. For instance, after applying the image normalization, two notebooks can generate completely different images, but we will consider them as generating the same results.

To assess the popularity of notebooks, we used the number of stars and forks from repositories as a proxy for the notebooks due to the lack of a better number. Since these numbers are from the repositories, they may not represent the popularity of a notebook. For instance, a tool repository that uses a notebook as an example of how to use it may be popular because of the tool and not because of the notebook. However, in our comparisons, popular notebooks had more quality features and reproducibility than the overall group, indicating that the proxy was sound.

External We collected repositories from GitHub for over one year. During this period, many repositories were updated, and many repositories were removed. Despite having data until April 16th, 2018, the repository states represent their state during the collection and not their state on this date. Additionally, we restricted our analysis to committed notebooks. Presumably, these notebooks receive more attention than the average scratchpad notebook and follow better practices. For instance, Grus (2018) pointed out the problem of untitled notebooks, but in our data, these notebooks correspond only to 1.93% of the notebooks.

4 Best Practices for the Reproducibility of Notebooks

In Section 3.6, we identified a set of bad practices that hinder the reproducibility and the benefits of the literate programming aspects of notebooks. Based on our findings, we propose the following best practices for the development of notebooks (Pimentel et al. 2019b):

1. **Use short titles with a restrict charset (A-Z a-z 0-9 . _ -) for notebook files and Markdown headings for more detailed ones in the body.** As discussed in Section 3.6.2, some operating systems may not support characters that many notebook titles use. Since notebooks support Markdown, we recommend using it to write the complex titles inside the notebooks and leave the notebook title as simple as possible.
2. **Pay attention to the bottom of the notebook. Check whether it can benefit from descriptive Markdown cells. Additionally, check whether the bottom cells have been executed. If not, consider either executing or removing them.** Users seem to pay more attention to the beginning of the notebook, as depicted in Sections 3.6.1, 3.6.3 and 3.6.6. Particularly, the bottom of notebooks usually has fewer Markdown cells and fewer executed code cells, compromising reproducibility.
3. **Abstract code into functions, classes, and modules, and test them.** As presented in Section 3.6.3, most users do not extract code into modules. This hinders the reuse and test of the notebooks. This is especially serious because notebooks are not packed together with tests. Thus, we recommend to abstract and test notebooks.
4. **Declare the dependencies in requirement files and pin the versions of all packages.** In Section 3.6.7, we identified that *requirements.txt* files fail less than other formats. We also recognized that many failures occur due to the lack of module dependencies. Hence, we recommend defining the dependencies explicitly and pinning the versions on a *requirements.txt* file.
5. **Use a clean environment for testing the dependencies to check if all of them are declared.** In the original reproducibility study (Pimentel et al. 2019b), we identified that installing dependencies in a clean environment failed more than just using an anaconda environment. Similarly, in the new study (Section 3.6.7), the bloated environment failed less due to ImportError and more due to updates on the modules. Thus, we recommend setting a clean environment and testing the notebooks dependencies before releasing it to check whether all of them are declared.
6. **Put imports at the beginning of notebooks.** This is not only close to the PEP 8 (van Rossum et al. 2001) recommendation but also helps in the verification of imports that we discussed above.

7. **Use relative paths for accessing data in the repository.** We identified that accessing files was also a common cause of errors in Section 3.6.7. Accessing project files using relative paths can reduce this issue.
8. **Re-run notebooks top to bottom before committing.** As presented in Section 3.6.6, many notebooks have out-of-order cells and skips. Moreover, these issues seem to impact the reproducibility (Section 3.6.7). Thus, we recommend re-running notebooks for restoring the execution counters and minimizing the impact of hidden states and out-of-order cells.

In the next section, we propose a tool that helps users to automatically verify most of these best practices.

5 Julynter: A Jupyter Linting Tool

Based on the results of our analyses and the proposed best practices, we propose Julynter,[1] a tool that performs linting on notebooks. Julynter is a Jupyter Lab extension that performs many checks on the quality and reproducibility of notebooks in real-time and produces recommendations. Figure 19 presents Julynter in action for the notebook of Fig. 1. Julynter recommended ten changes related to four categories: Invalid Title, Hidden State, Confuse Notebook, and Import. In addition to these categories, Julynter also has an Absolute Path category.

This section is organized as follows. Section 5.1 describes the approach. Section 5.2 presents the experiment design we defined to evaluate Julynter. Section 5.3 indicates how we collected the experiment data. Section 5.4 presents the experiment results. Finally, Section 5.5 describes the threats to the validity of the Julynter experiment.

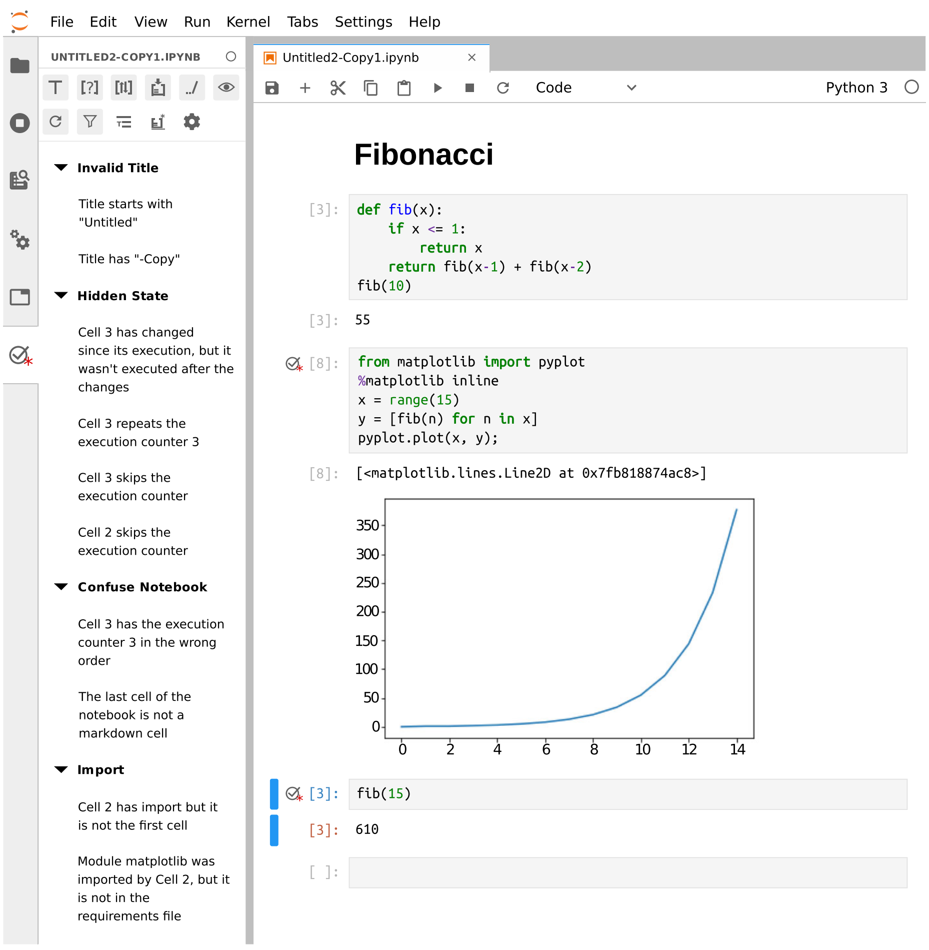
5.1 Approach

In addition to showing linting recommendations to users, Julynter also has filtering and display features to provide better readability. Users can filter recommendations by category, recommendation code, and appearance in a specific cell. Additionally, they can group the recommendations by category or by cell through the interface. They can store their preferences in the notebook, the project folder (i.e., the working directory of the Jupyter Lab execution), or the user directory.

The interface also allows users to click on the recommendations to apply actions. In the Invalid Title recommendations, it opens the rename notebook form. In the Import recommendation related to an import that does not exist in the requirements file, it adds the imported package to the requirements file, indicating its version. In a recommendation related to a cell that depends on a variable that was defined in a cell that does not exist anymore, it recreates the cell. Finally, for the other recommendations, it moves to the cell with the issue to allow users to fix them.

Julynter currently identifies 21 issues from notebooks. Table 9 presents these issues with their categories and the Julynter recommendations on how to fix them. Note that some recommendations require a kernel restart to really ensure the reproducibility. After some feedback from user experiments, we added a button to hide these recommendations for development notebooks. The Julynter extension detection covers six out of the eight best

[1] https://dew-uff.github.io/julynter



Invalid Title

Title starts with "Untitled"

Title has "-Copy"

Hidden State

Cell 3 has changed since its execution, but it wasn't executed after the changes

Cell 3 repeats the execution counter 3

Cell 3 skips the execution counter

Cell 2 skips the execution counter

Confuse Notebook

Cell 3 has the execution counter 3 in the wrong order

The last cell of the notebook is not a markdown cell

Import

Cell 2 has import but it is not the first cell

Module matplotlib was imported by Cell 2, but it is not in the requirements file

Fibonacci

```
[3]: def fib(x):
         if x <= 1:
             return x
         return fib(x-1) + fib(x-2)
     fib(10)
[3]: 55
[8]: from matplotlib import pyplot
     %matplotlib inline
     x = range(15)
     y = [fib(n) for n in x]
     pyplot.plot(x, y);
[8]: [<matplotlib.lines.Line2D at 0x7fb818874ac8>]
[3]: fib(15)
[3]: 610
[ ]:
```

Fig. 19 Julynter in action (left pane). By analyzing the notebook on the right pane, Julynter identified ten issues from four different categories

practices proposed in Section 4. To cover the seventh (using a clean environment for testing dependencies), we added a command-line interface (CLI) to Julynter that allows users to use pyenv environments, Conda environments, or Docker containers to detect dependency files and install them. Users can use this CLI to check if the installation is enough to import all modules. They can also use it to check the reproducibility of the notebooks after installing the modules. Finally, they can use it to automatically prepare an isolated environment with only the project dependencies. Hence, the only best practice that Julynter still does not cover is suggesting users to abstract code. Nonetheless, this suggestion is on our radar for future releases.

For detecting the issues, Julynter has two linting modules: a language-agnostic and a language-specific one. The language-agnostic module checks for common issues on the notebook structure that do not depend on the notebook language. This is the case for issues C1, C2, C3, C4, C5, H3, H4, T1, T2, T3, T4, T5, T6, T7. The language-specific module connects to the kernel to obtain information about the execution history, the cell dependencies,

Table 9 Issues detected by Julynter. The first character of the Code indicates the category: C—Confuse Notebook; H—Hidden State; I—Import; P—Path; T—Invalid Title

Code	Message	Suggestion
C1	Cell *:index* is a non-executed cell among executed ones.	Please consider cleaning it to guarantee the notebook reproducibility.
C2	Cell *:index* has the execution counter *:excount* in the wrong order.	Please consider re-running the notebook to guarantee its reproducibility.
C3	Cell *:index* is empty in the middle of the notebook.	Please consider removing it to improve the notebook readability.
C4	The first cell of the notebook is not a Markdown cell.	Please consider adding a Markdown cell to describe the notebook.
C5	The last cell of the notebook is not a Markdown cell.	Please consider adding a Markdown cell to conclude the notebook.
H1	Cell *:index* has execution results, but it was not executed in this session.	Please consider re-executing it to guarantee the reproducibility of the notebook.
H2	Cell *:index* has changed since its execution, but it was not executed after the changes.	Please consider re-executing it to guarantee the reproducibility of the notebook.
H3	Cell *:index* repeats the execution counter *:excount*.	Please consider re-running the notebook to guarantee its reproducibility.
H4	Cell *:index* skips the execution counter.	Please consider re-running the notebook to guarantee its reproducibility.
H5	Cell *:index* uses name "*:variable*" that was defined in *In [:excount]*, but it does not exist anymore.	Please consider restoring the cell and re-running the notebook to guarantee its reproducibility.
H6	Cell *:index* has the following undefined names: *:undefined*.	Please consider defining them to guarantee the reproducibility of the notebook.
I1	Cell *:index* has import but it is not in the first cell.	Please consider moving the import to the first cell of the notebook.
I2	Module *:module* was imported by Cell *:index*, but it is not in the requirements file.	Please consider adding it to guarantee the reproducibility of the notebook.
P1	Cell *:index* has the following absolute paths: *:paths*.	Please consider using relative paths to guarantee the reproducibility of the notebook.
T1	Title is empty.	Please consider renaming it to a meaningful name.
T2	Title starts with "Untitled".	Please consider renaming it to a meaningful name.
T3	Title has "-Copy".	Please consider renaming it to a meaningful name.
T4	Title has blank spaces.	Please consider removing them to support all OS.
T5	Title has special characters.	Please consider replacing them to support all OS.

Table 9 (continued)

Code	Message	Suggestion
T6	Title is too big.	Please consider renaming it to a shorter name and using a Markdown cell for the full name.
T7	Title is too small.	Please consider renaming it to a meaningful name.

the executed cells with absolute paths, and the status of imported modules on requirement files (issues H1, H2, H5, I1, I2, P1). Both modules connect to each other using Jupyter Comm. Hence, they do not interfere with the execution.

Figure 20 presents the architecture of Julynter and Jupyter communications. When the Jupyter Lab web application sends a cell to the kernel to execute, the Julynter extension triggers both linting modules. The language-specific module sends an invocation of a query function to the kernel, which then returns the execution history, the cell dependencies, the imports, and the absolute paths. Julynter processes this data together with the notebook definition and presents it back in the Jupyter Lab Application. The language-agnostic module processes only the notebook definition to report the issues.

Julynter has some limitations. First, the detection is restricted to run as an extension of Jupyter Lab. Currently, it cannot run as a standalone module nor as a Jupyter Notebook extension. Second, it must be executed in real-time. Starting Julynter in an existing notebook with a new kernel results in many warnings related to the presence of results from previous executions, and no warnings related to imports and absolute paths. This situation can be easily solved by running the whole notebook again, but it may not be what users expect when they use traditional linting tools. Finally, the language-specific module currently only supports Python.

5.2 Experiment Design

Since Julynter connects to the kernel to get the execution history in real-time, it is more capable of detecting hidden states and other issues than we were in the experiments we

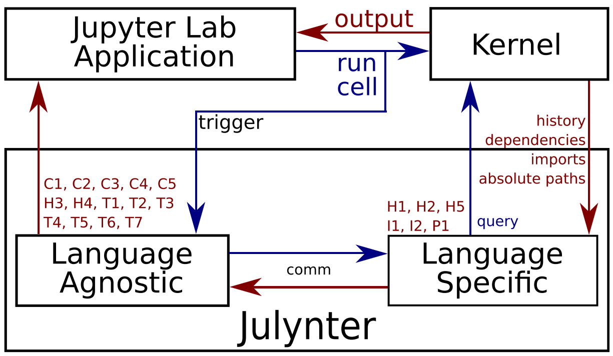


Fig. 20 Architecture of Julynter. Blue arrows represent input messages that occur before the cell execution. Red arrows represent output messages that occur after the kernel executes the cell

reported in the previous sections. However, since this detection relies on real-time history, it cannot detect hidden states in notebooks executed in previous sections. Thus, we could not simply use the notebooks we collected in Section 3.2 to evaluate Julynter as it would at most produce the results we presented before.

Hence, for evaluating the usability and capability of Julynter to ensure the quality of notebooks in the wild, we designed an experiment with users using Julynter over their own notebooks. The experiment was composed of three parts: a characterization form, the main experiment, and an exit questionnaire. In the characterization form, we asked questions about how frequently do they use notebooks, their experience with linting tools, Jupyter Notebook, Jupyter Lab, Python, R, and Julia, their preference between Jupyter Lab and Jupyter Notebook, and their usage of notebooks.

Due to the COVID-19 pandemic, we had to run the experiment remotely. Hence, for the main experiment, we adapted Julynter to collect usage data and asked the participants to install Julynter in their own machines and use it with their own notebooks for a week. For collecting the usage data, we also asked the participants to run a configuration tool to indicate which data they would like to share. Additionally, we added buttons for each recommendation in the tool to allow users to send feedback through positive, negative, and textual reports.

In the exit questionnaire, we asked users to send their collected data. We also asked about their satisfaction with each linting category using a Likert scale, and their overall satisfaction with the tool using both a System Usability Scale Questionnaire (Brooke 1996) and Microsoft Reaction Cards (Benedek and Miner 2002). Finally, we asked for suggestions to improve Julynter.

5.3 Data Collection

We conducted the experiment in three phases: I, II, and III. Phase I was a pilot and had the goal of identifying problems in the experiment itself. Two people participated in this phase: one coauthor and one undergrad student, and they identified five minor problems in the experiment. We do not use their results in the next section.

After fixing the experiment problems, we directly invited ten people for the next phase of the experiment. We selected these people based on our knowledge that they use notebooks. Only six of them completed the experiment during Phase II, and all six gave feedback on how to improve the tool. We implemented the requested features and started the last phase of the experiment. We shared the experiment in Data Science groups, Graduate Student groups, Python groups, and Twitter for this phase. Two people that were invited to Phase II but did not have time for the main experiment decided to participate in Phase III. Fourteen people answered the initial form, but only six completed the experiment. Figure 21 presents the flow of completion of the experiment for the main phases. Note that one participant did not reply to our invitation in Phase II, and one interrupted the experiment after filling the initial form. In Phase III, eight participants interrupted after the initial form, and two interrupted after starting the experiment.

Figure 22 presents the experience of the 12 participants that concluded either Phase II or Phase III. While all of them have at least an average experience with Jupyter Notebook and Python, most of them are novices in Jupyter Lab, which is the tool Julynter supports. It is expected, as Jupyter Lab is a newer tool released in 2018. When we asked which tool they prefer, seven participants prefer Jupyter Notebook, four participants prefer Jupyter Lab, and a participant has never used Jupyter Lab to have a preference.

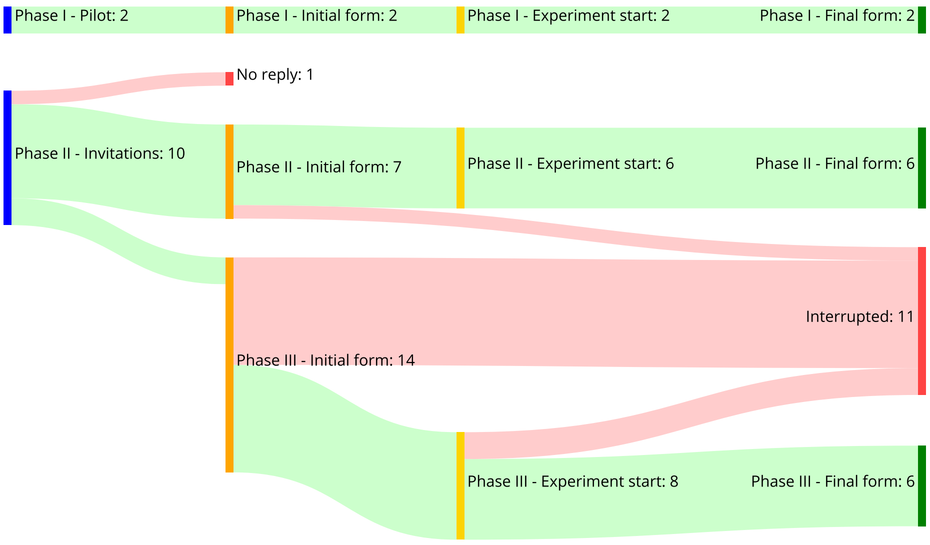


Fig. 21 Participants experiment flow

When we asked the participants to report their use-cases for Jupyter in a text field (i.e., we did not give predefined options and a participant could write multiple things), nine participants answered data-centric use-cases, such as data analysis, data cleaning, and data visualization; four use Jupyter for prototyping scripts and tools; four use or have used Jupyter for education tasks such as preparing course material or doing homework; three use it for research; two use it for communicating results and workflows; and one uses Jupyter to build interactive reports.

During the experiment, seven participants worked on data analysis projects, four participants used notebooks as scratchpads for prototyping and developing packages, and one participant prepared class materials.

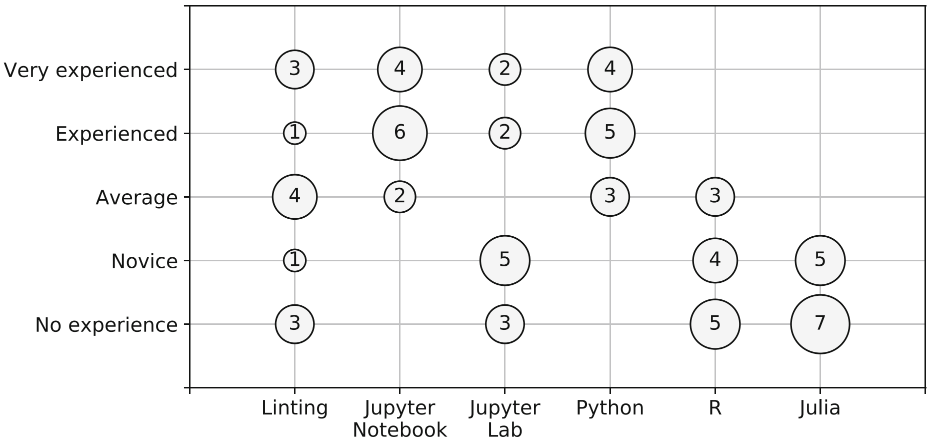


Fig. 22 Participants' experience

In the next subsection, we present the experiment results, filtering out both the participants of the pilot experiment (Phase I) and the participants that did not conclude the experiment.

5.4 Results and Discussion

Usage Since the participants used Julynter at their own pace with their own notebooks, the number of recommendations they received varied. Table 10 presents the number of days, usage sessions, notebooks each participant worked on, and the number of lint recommendations they received, solved, or clicked. We count a usage session as the moment a participant opens a notebook in the Jupyter Lab interface. Note that while many participants worked on a single notebook during the experiment, most of them opened the same notebook multiple times and through many days.

The participants #1 and #2 worked on the same usage session through different days, indicating that they did not close Jupyter Lab from a day to another. #6 was the only participant that worked on a single usage session of a single notebook during the experiment. Nonetheless, #6 was also the participant that received the most lint recommendations during Phase II. During Phase III, #10 also used Julynter for a single day, but in eight notebooks across 22 kernel sessions.

In this table, we count lints as all recommendations that Julynter shows and solved lints as all recommendations that disappear after a participant action. Despite the tool showing hundreds of recommendations to most participants, this number does not reflect directly on the effort they had to solve them. For instance, opening a big notebook with execution results leads to many `H1` recommendations, indicating that it has results from previous kernel sessions. Solving them is as easy as running all notebook cells. On the other hand, solving cells with `H4` recommendations (which identify skips) requires restarting the kernel and re-running all cells.

Table 10 Julynter usage statistics

Phase	P#	Days	Sessions	Notebooks	Lints	Solved	Solved (%)	Lint Clicks
II	#1	4	2	1	77	72	93.5%	62
II	#2	3	2	1	330	330	100.0%	218
II	#3	4	10	4	317	217	68.5%	202
II	#4	5	8	2	201	154	76.6%	129
II	#5	2	4	1	71	48	67.6%	40
II	#6	1	1	1	587	521	88.8%	124
III	#7	6	18	15	602	534	88.7%	460
III	#8	3	7	1	1,888	1,873	99.2%	880
III	#9	3	29	7	106	66	62.3%	85
III	#10	1	22	8	85	43	50.6%	54
III	#11	5	28	4	1,053	751	71.3%	333
III	#12	2	14	4	58	20	34.5%	34
Total	**12**	**28**	**145**	**49**	**5,375**	**4,629**	**86.1%**	**2,621**

The last line of the table was bold to indicate it represents the total (sum) of the other rows

Recommendations Figure 23 presents all lints that appeared to the participants, indicating the percentage of solved and unsolved lints. The recommendations T1 (empty title), T3 (title with "-Copy"), T6 (big title), and T7 (small title) did not appear for any participant. As expected, recommendations that can appear for any cell were more prevalent than the ones that appear for the notebook (C4, T1—T7) or in sporadic events such as importing modules (I1—I2) or using absolute paths (P1). Recommendations related to the organization of the notebook (H4 – skips, C2—out-of-order cells) appeared the most.

These results suggest that Julynter recommends changes to improve the quality of the notebook that the participants are willing to apply. Nonetheless, the participants solved more some types of recommendations than others.

Recommendation Feedback In the *Confuse Notebook* group, C4 and C5 were the least solved recommendations. These recommendations suggest using Markdown cells in the beginning to describe the notebook and in the end to conclude it, respectively. We received four negative reports about C5, three textual reports asking why it was necessary, and one textual report complaining that it appeared too soon (i.e., before finishing the notebook to draw conclusions). C4 was more controversial: we received two negative reports and three positive ones about it. #10 sent a textual report indicating that the recommendation was good, but it would not be fixed because the notebook was part of a tool written by someone else. #12 sent both positive and negative reports about it, with a textual report indicating that "not all notebooks are literate ones".

In the *Hidden State* group, participants solved the least H1, H4, and H6 recommendations. As described before, H1 appears when the user first opens a notebook that has results from previous executions. If the user does not want to run the notebook, it is expected not to have it solved. We received two negative reports with textual reports. A participant indicated that the notebook was not executed yet. The other indicated that an error in a previous part of the notebook prevented its normal execution.

The recommendation H4 is harder to solve, as it requires restarting the kernel and re-running all cells. This recommendation received a textual report indicating that the participant did not understand the suggestion. It also received a positive report. Related to this recommendation, in the exit questionnaire, two participants suggested that linting notebooks should occur in two phases: a phase for supporting exploratory analyses with skips in the cell execution counter and a phase to guarantee the reproducibility.

The recommendation H6 appears when a cell uses a variable that is not defined in the notebook. A participant sent a textual report indicating that the recommendation was not appropriate because the variable was actually defined. When we analyzed the notebook

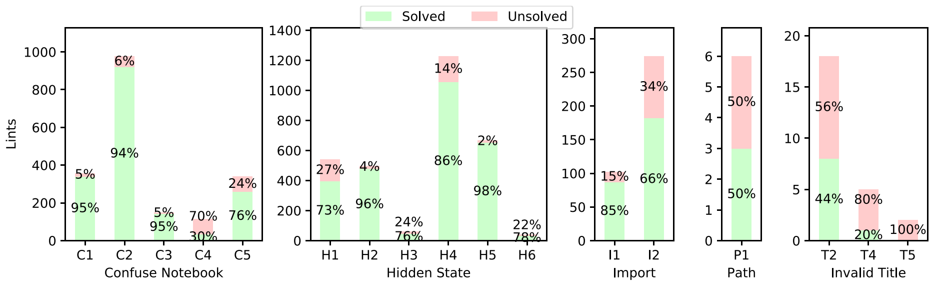


Fig. 23 Solved and unsolved lints

code, we noted that a widget uses IPython functions to change the global state. As this is a very unusual situation, we suggest using Julynter filters for this type of false-positive recommendation.

In the *Import* group, participants solved the least the recommendation `I2` (adding packages to "requirements.txt") and two of them sent textual reports indicating that they do not use these files. Once again, #12 sent both positive and negative reports in different notebooks. The other recommendation (`I1`—moving imports to the beginning) also received feedback. A participant sent a negative report without indicating why, but two participants sent positive reports. #10 sent a textual report indicating that imports should indeed stay in the first cell, but the issue would not be fixed as the notebook was designed by someone else.

The recommendation to not use absolute *Paths* (`P1`) only appeared for two participants. One could not solve it due to a bug that Julynter had during Phase II in Windows. The participant who experienced the bug sent a positive report about the suggestion, though.

Finally, the *Invalid Title* group had the recommendations the participants solved the least: `T5` (title with special characters), `T4` (title with blank spaces), and `T2` (title with "Untitled"). A participant sent a negative report about `T4`, asking why the title could not have blank spaces. The same participant sent a positive report about `T2`.

In the exit questionnaire, we asked the participants to indicate the extent to which they are satisfied with each linting recommendation group. Figure 24 presents the results. In this figure, we filtered out groups that did not appear for the participant. The participants are mostly satisfied with the recommendations, and the only groups that caused dissatisfaction are the *Confuse Notebook* and *Hidden States*. The participants that did not like these groups are the ones that sent negative feedback to `C5`, `H1`, and `H4`.

Usability We used the System Usability Score (SUS) (Brooke 1996) to evaluate the usability of Julynter. This score is calculated based on ten standard statements presented in the exit questionnaire. The user can select answers ranging on a Likert scale from 1 (completely disagree) to 5 (completely agree). We calculated an average SUS score of 77.5 on a scale between 0 and 100. According to Bangor et al. (2008), this is a good score in the acceptability range. The minimum score was 52.5, which is close to the minimum OK score in the

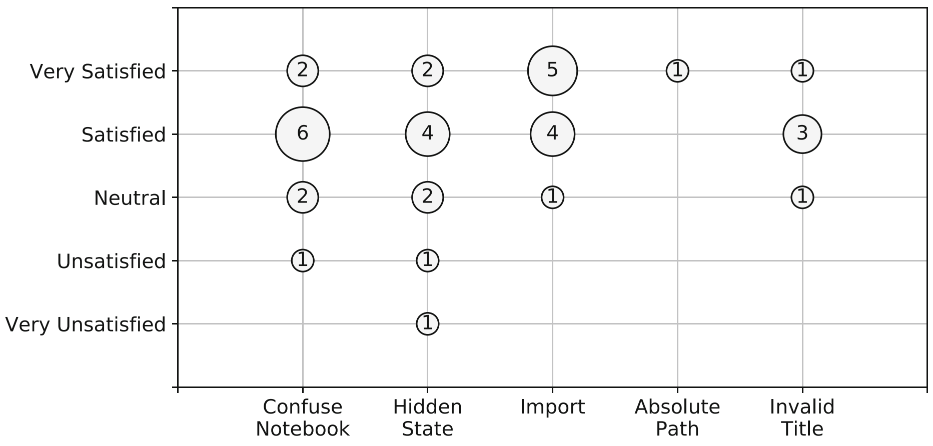


Fig. 24 Satisfaction with the lint groups

acceptability range. On the other hand, the maximum score was 100, which is categorized as the best imaginable score (Bangor et al. 2008).

Figure 25 presents a word cloud representing the Microsoft Product Reaction Cards (Benedek and Miner 2002) we presented in the exit questionnaire. The most selected cards are *usable* and *useful*. These cards are a positive indication that the participants see value in the adoption of Julynter and that this adoption does not have a high barrier.

In Fig. 25, the *usable* word was more prevalent among the participants of Phase III. It occurred because we made many changes to improve the tool based on the reports that the participants submitted using the tool and the exit questionnaire of Phase II: we fixed several bugs, improved the interface, and introduced several new features, including the possibility of checking the reproducibility of notebooks and validating the recommendations in a command-line interface. We also contacted the participants of Phase II after the changes and asked them what they thought about the changes. #1 did not reply. #3 had an issue updating and running Julynter and also did not give feedback. The other four participants liked the changes.

Besides tool-related reports, some participants of both phases expressed concerns about the data intrusiveness of the experiment itself. Some even suggested designing a controlled remote experiment in a virtual machine or Binder for security reasons. We considered it when designing the experiment, but we anticipated it would be artificial and would not detect which recommendations occur in the wild nor whether the recommendations are good enough for users to apply in their own notebooks. Moreover, some indicated that they use Jupyter Notebook instead of Jupyter Lab daily and did not run the experiment much. Finally, some participants reported bugs that were not caused by Julynter. A participant indicated that despite all effort with linting, "the biggest problem remains the lack of training of scientists in software engineering".

5.5 Threats to Validity

The Julynter experiment also has some threats to validity that we depict below.

Internal We selected participants for Phase II based on our previous knowledge that they used Jupyter. This may bias the selection of participants to close contacts. In an attempt to mitigate this threat, we distributed the invitation for Phase III to public data science

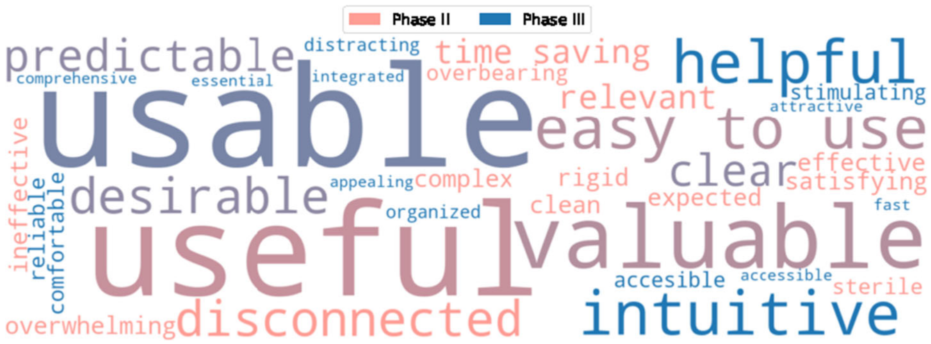


Fig. 25 Chosen words in the Microsoft Product Reaction Cards (Benedek and Miner 2002). The colors vary according to the experiment phase in a gradient. Mixed colors indicate that participants of both phases chose the word and the mixing intensity indicates the proportion

and research groups in Telegram and Whatsapp, to the official Jupyter Lab Gitter, and on Twitter. According to Twitter's current statistics, the tweet was retweeted 22 times, people saw the invitation tweet 4,049 times, and 326 people interacted with the tweet, despite only 12 filling the initial form—the remaining two participants came from Phase II invitations. Nonetheless, the selection is also biased towards our reach in social media.

Construct Due to the COVID-19 pandemic, we had to design a remote experiment instead of a lab experiment to evaluate Julynter. In this experiment, we distributed Julynter to participants, for them to use at their own pace with their own notebooks. They may have had very distinct usages and different goals that may not justify the usage of Julynter. In fact, during the experiment, some participants used Julynter in scratchpad notebooks that usually do not have a high requirement of quality. Despite this threat, these participants had a positive feeling about Julynter overall.

External We had a small number of participants. Even though we listened to their feedback to improve the tool, the number is not significant to draw conclusions on which are the best recommendations and how users would use the tool in the wild.

6 Related Work

Neglectos (2018) analyzed 2,702 Jupyter Notebooks written in Python and reported on the most commonly-used modules and modules that are used together. Their results for the most used modules are similar to ours (see Fig. 10). In both analyses, *numpy* and *matplotlib* appear as the most imported modules, in this order. Additionally, six other modules appear in both analyses (*pandas*, *sklearn*, *os*, *scipy*, *tensorflow*, and *IPython*), but in distinct orders. They show *warnings* and *collections* in the top 10, while we indicate *seaborn* and *time*.

Kery et al. (2018) interviewed 21 data scientists and surveyed 45 data scientists to understand how they use notebooks. They identified three types of use cases for notebooks: (i) scratchpad notebooks, (ii) notebooks with code that is later extracted to scripts, and (iii) notebooks for sharing results and knowledge. The existence of use cases not too related to literate programming (i and ii) indicates why some notebooks do not have Markdown cells. For those notebooks that have Markdown cells, Kery et al. (2018) identified that data scientists go through a *cleaning* phase, in which they reduce the notebook size by merging small cells into bigger ones, adding Markdown annotations, and organizing the linearity of the execution.

Kery et al. (2018) also identified good and bad practices on notebooks. As a bad software engineering practice, they recognized that data scientists tend to copy and paste the code for reuse, instead of extracting the code to a function. As a good practice, they identified that data scientists do not let their notebooks grow too much beyond their scope. However, this good practice occurs mainly due to notebook constraints in performance and navigability.

Rule et al. (2018) performed three analyses over notebooks. In the first one, they analyzed 1.23 million notebooks from 191,402 GitHub repositories. While their goal on this analysis was to extract insights on the usage of notebooks, our goal is to dive into evidences of best practices. Nonetheless, we obtained similar results when we analyzed the distribution of notebooks by repository, the most used programming languages, the distribution of cell types in the notebook, the size of Markdown cells, and the top three most imported Python modules.

In the second analysis, Rule et al. (2018) sampled 221 notebooks from 52 repositories with an academic reference in the README to understand the narrative of academic notebooks. Most of these repositories contained not only the notebook files, but also raw data, figures, and manuscript files. They identified two types of notebooks: full analysis and tutorial notebooks. Both are related to the literate programming use case. They also identified that while 55% of notebooks had introductory Markdown text, only 3% had a conclusion.

Finally, in the third analysis, Rule et al. (2018) interviewed 15 data analysts that recognized the importance of cleaning and annotating notebooks and indicated four reasons for reusing a notebook: tracking provenance, code reuse, reproducibility of experiments, and presentation of results. However, our results suggest that the reproducibility of notebooks is far from ideal with only 4.90% to 15.04% of Python notebooks being replicated successfully. Additionally, since we identified a high amount of hidden state cells in our analysis, notebooks may not be very suitable for tracking provenance by themselves. Instead, it is better to use a tool designed for provenance tracking (Koop and Patel 2017; Pimentel et al. 2015; Samuel and König-Ries 2018; Brachmann et al. 2020). Pimentel et al. (2019a) provides a comprehensive list of tools for tracking provenance from scripts.

Since the publication of our first study (Pimentel et al. 2019b), some studies have been carried out to analyze some quality aspects of notebooks (Wang et al. 2020; Koenzen et al. 2020; Källén et al. 2020).

Wang et al. (2020) analyzed the quality in programming style and code contents of 1,982 notable projects curated by the Jupyter team. They found many problems on notebooks, such as not following PEP8 code style guidelines, defining variables and not using them, and using deprecated functions.

Koenzen et al. (2020) studied a sample of 1,000 repositories containing at least one notebook each and observed 8 participants doing a set of tasks to analyze the code duplication and reuse in notebooks. They found that 1 in 13 code cells in notebooks are duplicates, and they indicate that the users prefer to reuse code from online sources instead of reusing from existing notebooks. Moreover, they found that no participant reused code from the Git repository.

In a work that has not been peer-reviewed yet, Källén et al. (2020) analyzed 2.7 million notebooks to identify code clones. They found that clones are usually small and tend to occur more among notebooks of distinct repositories. They also found that more than 70% of code snippets are copies of other snippets, but many of these copies are accidental (e.g., two notebooks importing the same libraries appear as a clone).

Samuel and König-Ries (2020) built ReproduceMeGit on top of the analysis tools we developed in the previous study (Pimentel et al. 2019b) to calculate and report quality and reproducibility statistics for individual repositories.

Unlike prior work, we analyze not only the quality, but also the reproducibility of Jupyter Notebooks, and try to identify (and quantify the use of) practices that hinder reproducibility. Additionally, we extend the quality analyses of our first study by sampling and extracting a set of popular notebooks, which provide insights into the context of these practices.

Moreover, we propose Julynter, a linter tool for Jupyter Lab. While traditional linters already exist for linting code on Jupyter cells (McNutt 2019; Krassowski 2019), Julynter goes further and considers characteristics in the structure of notebooks and the presence of hidden-states. Hence, Julynter is complementary to traditional linters.

7 Conclusion

This paper has four main contributions. First, it analyzes evidence of good and bad practices on the development of Jupyter Notebooks regarding quality and reproducibility by going through the main criticisms that the format receives (Grus 2018; Mueller 2018; Pomogajko 2015). Second, it presents a detailed study that measures the reproducibility rate of notebooks under different settings (**RQ7**). Third, it proposes a set of good practices that aim to minimize the criticisms and improve the reproducibility rate for notebooks (Section 4). Finally, it proposes and evaluates Julynter, a Jupyter Lab linting extension (Section 5) that helps users maintain the quality and reproducibility of their notebooks.

In our experimental results, we found evidence of both good and bad practices. As good practices, we found the usage of literate programming aspects of notebooks (e.g., Markdown cells and visualizations), the application of abstractions on notebooks that have more complex control flows, and the usage of descriptive filenames. As bad practices, we found that most notebooks do not test their code and that a large number of notebooks has characteristics that hinder the reasoning and the reproducibility, such as out-of-order cells, non-executed code cells, and the possibility of hidden states.

While we discussed many criticisms of notebooks in this paper, we did not cover all of them. Other criticisms relate to the versioning (Pomogajko 2015; Mueller 2018), security risks (Pomogajko 2015), lack of IDE features (Grus 2018; Mueller 2018), lack of support for long asynchronous tasks (Mueller 2018), and lock-in aspects of Jupyter (Grus 2018).

In the reproducibility study (RQ7), we explored distinct environments to run the notebooks: shared environments with installations, isolated environments, and isolated bloated environments with pre-installed dependencies. We achieved a reproducibility rate that ranged from 4.90% to 15.04%. Further software reproducibility research could explore other environments such as defining different levels of bloated environments, installing dependencies on isolated environments, or guessing the dependencies on repositories that do not declare them to install in either of these environments.

Similarly, we ran notebooks following either the execution counter order or the top-down order and found that the top-down order was more reproducible. However, we foresee exploring other execution orders, such as executing cells with imports first, then following the cell execution order.

Not all types of notebooks are meant to be reproducible. We collected a representative sample of our corpus, and manually categorized the sample—correctly categorizing the whole corpus in an automatic way would be challenging. Most of the sample falls into the "education" category (course exercises, tutorials, etc.) and are thus toy projects. However, users that have no intention to have quality and reproducibility in toy projects may still benefit from knowing what makes a notebook reproducible and practicing it in their toy projects for the moment they move to more professional projects.

As another look into the problem of analyzing notebooks that are not meant to be reproducible, we selected a subset of our corpus containing only popular notebooks. We did this in the hopes that popular notebooks receive more attention and should be meant to be more reproducible. When comparing this popular subset with our original corpus, we noticed that, in fact, they are more reproducible and have more quality features than the general group.

Other reproducibility questions are also worth investigating. We intend to investigate strategies to assess the different types of projects (e.g., student notebooks, tutorial notebooks, research notebooks, scratchpads, dashboards, among others) to compare their metrics. We also foresee comparing notebooks to general-purpose scripts to understand whether one or another has better quality and reproducibility measures.

In addition to studying the reproducibility of existing notebooks, we also envision an opportunity to propose tools to improve the reproducibility rate by managing notebooks' dependencies and reducing the mess in their organization. Julynter is a small step towards this goal. Still, it has many improvement opportunities, such as improving the dependency inference through program slicing, adding more quality checks, and suggesting users to abstract and test code. Additionally, we foresee developing a cleaning tool that could help to transform scratchpad notebooks with bad practices into clean and reproducible notebooks for publishing.

The data, scripts, and notebooks used in this study are available at https://doi.org/10.5281/zenodo.3519618.

Acknowledgments This work is partially supported by CAPES, CNPq, FAPERJ, the NYU Moore-Sloan Data Science Environment, and NSF award OAC-1640864.

References

Agrawal R, Srikant R et al (1994) Fast algorithms for mining association rules. In: VLDB conference, VLDB, vol 1215, pp 487–499
Anaconda (2018) Anaconda software distribution. https://www.anaconda.com. Accessed: 2019-10-01
Arnaoudova V, Di Penta M, Antoniol G (2016) Linguistic antipatterns: what they are and how developers perceive them. Empir Softw Eng 21(1):104–158
Bangor A, Kortum PT, Miller JT (2008) An empirical evaluation of the system usability scale. Int J Hum–Comput Interact 24(6):574–594
Benedek J, Miner T (2002) Measuring desirability: new methods for evaluating desirability in a usability lab setting. Proc Usabil Prof Assoc 2003(8–12):57
Brachmann M, Spoth W, Kennedy O, Glavic B, Mueller H, Castelo S, Bautista C, Freire J (2020) Your notebook is not crumby enough, replace it. In: Conference on innovative data systems research, CIDR, pp 1–16
Brooke J (1996) Sus: a "quick and dirty" usability. Usability Evaluation in Industry, p 189
Burns T, Ward G (2013) ipython-nose. https://github.com/taavi/ipython_nose. Accessed: 2019-10-01
Cannon B, Smith N, Stufft D (2016) Pep 518: specifying minimum build system requirements for python projects. https://www.python.org/dev/peps/pep-0518/. Accessed: 22 Sep 2020
Chirigati F, Rampin R, Shasha D, Freire J (2016) Reprozip: computational reproducibility with ease. In: International conference on management of data, ACM, SIGMOD, pp 2085–2088
Collberg C, Proebsting T, Moraila G, Shankaran A, Shi Z, Warren AM (2014) Measuring reproducibility in computer systems research. Tech rep. Department of Computer Science, University of Arizona
Danilak MM (2016) langdetect. https://pypi.org/project/langdetect/. Accessed: 2019-10-02
Freire J, Koop D, Santos E, Silva C (2008) Provenance for computational tasks: a survey. Comput Sci Eng 10(3):11–21. https://doi.org/10.1109/MCSE.2008.79
Garousi V, Küçük B (2018) Smells in software test code: a survey of knowledge in industry and academia. J Syst Softw 138:52–81
Grus J (2018) I don't like notebooks. https://conferences.oreilly.com/jupyter/jup-ny/public/schedule/detail/68282, jupyterCon
Han J, Pei J, Kamber M (2011) Data mining: concepts and techniques. Elsevier
Hook D, Kelly D (2009) Testing for trustworthiness in scientific software. In: ICSE workshop on software engineering for computational science and engineering, SE-CSE, pp 59–64. https://doi.org/10.1109/SECSE.2009.5069163
Horwitz S, Reps T (1992) The use of program dependence graphs in software engineering. In: International conference on software engineering, ACM, ICSE, pp 392–411
Hürsch WL, Lopes CV (1995) Separation of concerns. Tech. rep., Northeastern University
Israel GD (1992) Determining sample size. Tech. rep., University of Florida
Källén M, Sigvardsson U, Wrigstad T (2020) Jupyter notebooks on github: characteristics and code clones. arXiv:200710146

Kalliamvakou E, Gousios G, Blincoe K, Singer L, German DM, Damian D (2014) The promises and perils of mining github. In: Working conference on mining software repositories. ACM, MSR, pp 92–101

Kery MB, Radensky M, Arya M, John BE, Myers BA (2018) The story in the notebook: exploratory data science using a literate programming tool. In: CHI conference on human factors in computing systems. ACM, CHI, New York, pp 174:1–174:11. https://doi.org/10.1145/3173574.3173748

Kluyver T, Ragan-Kelley B, Pérez F, Granger BE, Bussonnier M, Frederic J, Kelley K, Hamrick JB, Grout J, Corlay S et al (2016) Jupyter notebooks—a publishing format for reproducible computational workflows. In: Loizides F, Scmidt B (eds) International conference on electronic publishing. IOS Press, ELPUB, Göttingen, pp 87–90. https://eprints.soton.ac.uk/403913/

Knuth DE (1984) Literate programming. Comput J 27(2):97–111

Koenzen A, Ernst N, Storey MA (2020) Code duplication and reuse in jupyter notebooks. arXiv:200513709

Koop D, Patel J (2017) Dataflow notebooks: encoding and tracking dependencies of cells. In: Workshop on the theory and practice of provenance. USENIX, TaPP, Seattle, pp 1–7

Krassowski M (2019) Language server protocol integration for jupyter(lab). https://github.com/krassowski/jupyterlab-lsp/, accessed: 2020-10-13

Lewine D (1991) POSIX programmers guide. O'Reilly Media, Inc.

McNutt A (2019) Jupyterlab-flake8. https://github.com/mlshapiro/jupyterlab-flake8. Accessed: 2020-10-13

Microsoft (2018) Naming files, paths, and namespaces. Windows Dev Center. https://docs.microsoft.com/en-us/windows/desktop/FileIO/naming-a-file. Accessed: 2019-10-01

Mueller A (2018) 5 reasons why jupyter notebooks suck. https://towardsdatascience.com/5-reasons-why-jupyter-notebooks-suck-4dc201e27086/, accessed: 2019-10-01

Munaiah N, Kroh S, Cabrey C, Nagappan M (2017) Curating github for engineered software projects. Empir Softw Eng 22(6):3219–3253

Myers GJ, Badgett T, Thomas TM, Sandler C (2004) The art of software testing, vol 2. Wiley Online Library, Hoboken

Neglectos (2018) A preliminary analysis on the use of python notebooks. https://blog.bitergia.com/2018/04/02/a-preliminary-analysis-on-the-use-of-python-notebooks/, Accessed: 2019-10-01

Parente P (2020) nbestimate. https://nbviewer.jupyter.org/github/parente/nbestimate/blob/master/estimate.ipynb, accessed: 2020-12-03

Pérez F, Granger BE (2007) Ipython: a system for interactive scientific computing. Comput Sci Eng 9(3):21–29

Pimentel JF (2016) ipython-unittest. https://github.com/JoaoFelipe/ipython-unittest. Accessed: 2019-10-01

Pimentel JFN, Braganholo V, Murta L, Freire J (2015) Collecting and analyzing provenance on interactive notebooks: when ipython meets noworkflow. In: Workshop on the theory and practice of provenance. USENIX, TaPP, Edinburgh, pp 155–167

Pimentel JF, Freire J, Murta L, Braganholo V (2019a) A survey on collecting, managing, and analyzing provenance from scripts. ACM Comput Surv 52(3):47:1–47:38. https://doi.org/10.1145/3311955. http://doi.acm.org/10.1145/3311955

Pimentel JF, Murta L, Braganholo V, Freire J (2019b) A large-scale study about quality and reproducibility of jupyter notebooks. In: Proceedings of the international conference on mining software repositories. IEEE Press, MSR, pp 507–517

Pomogajko K (2015) Why I don't like Jupyter (FKA IPython Notebook). https://yihui.name/en/2018/09/notebook-war/. Accessed: 2019-10-01

PyPA (2020) Python Packaging Documentation: install_requires vs requirements files. https://packaging.python.org/discussions/install-requires-vs-requirements/#requirements-files. Accessed: 2020-10-08

Python-Wiki (2019) Python testing tools taxonomy. https://wiki.python.org/moin/PythonTestingToolsTaxonomy. Accessed: 2019-10-01

ReproZip (2017) Making Jupyter Notebooks Reproducible with ReproZip. https://docs.reprozip.org/en/1.0.x/jupyter.html. Accessed: 2019-10-01

Rule A, Tabard A, Hollan JD (2018) Exploration and explanation in computational notebooks. In: Proceedings of the CHI conference on human factors in computing systems. ACM, CHI, New York, pp 32:1–32:12. https://doi.org/10.1145/3173574.3173606. http://doi.acm.org/10.1145/3173574.3173606

Samuel S, König-Ries B (2018) Provbook: provenance-based semantic enrichment of interactive notebooks for reproducibility. In: The international semantic web conference, ISWC. Monterey, Springer, pp 1–4

Samuel S, König-Ries B (2020) Reproducemegit: a visualization tool for analyzing reproducibility of jupyter notebooks. In: ProvenanceWeek, pp 1–2

Shen H (2014) Interactive notebooks: sharing the code. Nature News 515(7525):151

Staley T (2017) Making git and jupyter notebooks play nice. http://timstaley.co.uk/posts/making-git-and-jupyter-notebooks-play-nice/. Accessed: 2019-10-01

Tim, Doorknob (2014) Is space not allowed in a filename? Unix & Linux. https://unix.stackexchange.com/q/148043, accessed: 2019-10-01

Tufano M, Palomba F, Bavota G, Di Penta M, Oliveto R, De Lucia A, Poshyvanyk D (2017) There and back again: can you compile that snapshot? J Softw: Evol Process 29(4):e1838

Udacity (2017) Deep learning nanodegree foundation. https://github.com/udacity/deep-learning. Accessed: 2019-10-01

van Rossum G, Warsaw B, Coghlan N (2001) Pep 8: style guide for python code. https://www.python.org/dev/peps/pep-0008/. Accessed: 2019-10-01

Vavrová N, Zaytsev V (2017) Does python smell like java? The Art. Sci Eng Program 1(2):11–1

Wang J, Li L, Zeller A (2020) Better code, better sharing: on the need of analyzing jupyter notebooks. In: International conference on software engineering: new ideas and emerging results, ICSE, pp 53–56

Wilson G, Aruliah DA, Brown CT, Chue Hong NP, Davis M, Guy RT, Haddock SHD, Huff KD, Mitchell IM, Plumbley MD, Waugh B, White EP, Wilson P (2014) Best practices for scientific computing. PLOS Biol 12(1):1–7. https://doi.org/10.1371/journal.pbio.1001745

Affiliations

João Felipe Pimentel[1] · **Leonardo Murta**[1] · **Vanessa Braganholo**[1] · **Juliana Freire**[2]

Leonardo Murta
leomurta@ic.uff.br

Vanessa Braganholo
vanessa@ic.uff.br

Juliana Freire
juliana.freire@nyu.edu

1 Instituto de Computação, Universidade Federal Fluminense, Niterói, RJ, Brazil

2 Department of Computer Science and Engineering, New York University, New York, NY, USA