A Large-scale Study about Quality and Reproducibility of Jupyter Notebooks

João Felipe Pimentel*, Leonardo Murta*, Vanessa Braganholo*, and Juliana Freire†
*Universidade Federal Fluminense
Niterói, Brazil
{jpimentel,leomurta,vanessa}@ic.uff.br
†New York University
New York, USA
juliana.freire@nyu.edu

***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 all sorts of 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 notebooks are being used leads to unexpected behavior, encourage poor coding practices, and that their results can be hard to reproduce. To understand good and bad practices used in the development of real notebooks, we studied 1.4 million notebooks from GitHub. We present a detailed analysis of their characteristics that impact reproducibility. We also propose a set of best practices that can improve the rate of reproducibility and discuss open challenges that require further research and development.**

***Index Terms*—jupyter notebook, github, reproducibility**

I. Introduction

Literate programming is a paradigm that seeks to help in the communication of programs [1] by interleaving formatted natural language text, executable code snippets, and computation results. Code snippets generate the computation results and natural language text explains both the code and the results.

Jupyter Notebook is the most widely-used system for interactive literate programming [2]. It was designed to make data analysis easier to document, share, and reproduce. The system was released in 2013, and today there are over 1 million notebooks in GitHub [3]. Jupyter Notebook originated from IPython [4] and, in addition to Python, it supports a variety of programming languages, such as Julia, R, Javascript, and C. Notebooks interleave 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. [5] 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 [6]–[8]. Among the main criticisms are hidden states, unexpected execution order with fragmented code, and bad practices in naming, versioning, testing, and modularizing code. Also, the notebook format does not encode its library dependencies with associated versions, which can make it hard (or even 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 [9], regarding separation of concerns [10], tests [11], and maintenance [12].

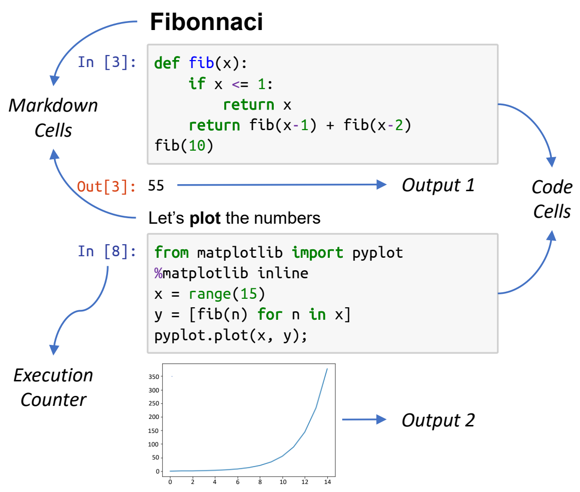
Existing work attempted to understand how notebooks are used [3], [13], [14]. They analyzed different aspects of notebooks, including use cases [13], narrative [3], [13], and structure [3], [14]. However, they did not attempt to run the notebooks and check characteristics related to reproducibility.

In this paper, we present a study that aims to provide insights into the reproducibility aspects of real notebooks. To better understand the different characteristics that impact reproducibility, using the aforementioned criticisms as a guide, we define metrics to analyze the extent of adoption of both good and bad practices. To compute these metrics, we created a corpus consisting of 1,159,166 unique notebooks collected from 264,023 GitHub repositories and extracted information about the structure of the notebooks. Besides, to assess the reproducibility rate, we attempted to execute the notebooks. As we discuss in Section IV, 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 propose a set of best practices for the development of Jupyter Notebooks.

This paper is organized as follows. Section II provides some background about literate programming and Jupyter Notebooks. Section III describes the method we followed in this study and our notebook corpus. We present the analysis results in Section IV. In Section V, we propose a set of best practices for the development of Jupyter Notebooks. We discuss the threats to the validity of our study in Section VI and present related work in Section VII. Finally, we conclude in Section VIII where we outline directions for future work.

II. Background

Knuth [1] introduced the *literate programming* paradigm that, by combining code and natural language, enables programmers to explicitly state the thoughts behind a program's



Fibonnaci

```
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. A notebook example with markdown, code, and output.

logic. It allows the programmers themselves and others to more easily understand the code. Nowadays, literate programming is used in *interactive computational notebook environments* [2], which allow parts of a notebook to be executed with immediate visualization of results and formatted text.

A *Jupyter Notebook* [2] is at the same time an interactive literate programming document and an application that executes the document. In this paper, to avoid the ambiguity, we use the term *Jupyter* to refer to the application, as well as to *Jupyter Lab* and other applications that execute notebooks. 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. 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 [15], i.e., the notebook title and definition of its cells. An executed notebook contains *prospective* data plus *retrospective* data [15] 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 of *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 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 presents two 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 the other.

III. Materials and Methods

In this section, we discuss the method for the analyses we have carried out and data collection procedures we used to obtain evidence for quality and reproducibility best practices (or lack of thereof) in Jupyter notebooks.

A. Analyses

As discussed in Section I, Jupyter has received substantial criticism for encouraging bad coding habits and practices that hinder reproducibility [6]–[8]. In what follows, we discuss these criticisms and propose analyses to quantify their impact on notebooks present in GitHub. These criticisms relate to both prospective and retrospective components of notebooks [15]. We thus frame our analyses in terms of seven research questions (RQ1, RQ2, RQ3, RQ4, RQ5, RQ6, and RQ7), which we organize into these two categories.

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

RQ1. *How are literate programming features used in notebooks?* According to Wilson et al. [9], 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 [7] 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 literate programming tool by looking at the number of markdown cells and their positions in the notebooks. Investigating the presence of linguistic anti-patterns [16] 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". This discourages users to choose meaningful names [7]. Also, the notebook title is the same as the filename. Using the 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 [17]). Moreover, it makes the notebook title susceptible to filename conventions (e.g., not using space characters [18]). 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 . _ -) [19].

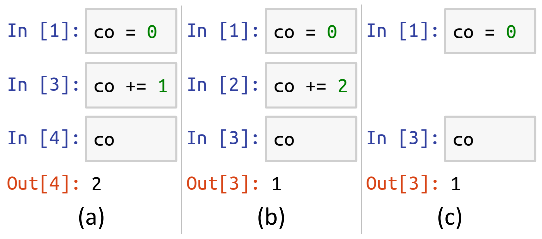
RQ3. *How do notebooks use modules, functions, and classes?* In traditional programming languages, modules, functions, and classes are essential constructs to maintain the separation of concerns in software [10]. In literate programming environments, markdown cells could be used to separate the concerns. However, this would suffer from 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 [7]. 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 [11]. Since notebooks are not modules, testing code in a notebook is challenging as it requires mixing text code with the notebook narrative code [7], [8]. To search for evidence of testing in notebooks, we analyze the imported modules names that contain "test", "Test", "TEST", "mock", "Mock", or "MOCK" as a substring. 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 [20].

2) 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 [21]. Furthermore, Jupyter is also used as an IDE for general purpose software development with the goal of extracting the produced code to scripts afterward [13]. We analyze the number of notebooks that have retrospective data and whether Jupyter is used as 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 is not intuitive to how most people expect to run code [6]–[8]. Moreover, cells that appear at the beginning of notebooks may depend on cells that appear later, causing even more issues to


```
In [1]: co = 0      In [1]: co = 0      In [1]: co = 0

In [3]: co += 1     In [2]: co += 2

In [4]: co          In [3]: co          In [3]: co

Out[4]: 2           Out[3]: 1           Out[3]: 1
```

Fig. 2. Three types of Hidden States: (a) Re-execution; (b) edited cell; (c) removed cell.

people that run them in the default order [22]. To quantify the prevalence of this issue, we identify notebooks that have cells in a non-linear order.

In addition to the out-of-order cell issue, 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 the previously executed cells. However, when the user runs the same 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 [6], [7].

Figure 2 presents three examples of hidden states caused by these situations. 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. 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.

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 there exist 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 [7]. The lack of versions may cause incompatibilities and prevent the usage in other systems. In Python, this issue can be addressed by defining dependencies in standard files: `requirements.txt`, `setup.py`, 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. We define *unambiguous execution order notebooks* as notebooks that have only one valid execution sequence. That is, they have neither cells with repeated execution counters, nor cells whose counter count 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.

In this analysis, we execute notebooks following the execution counter order to identify notebooks that when executed lead to results that are the same as the results stored with the notebooks.

While Jupyter supports multiple kernels, a Python installation is usually restricted to a single IPython kernel. Thus, we use *conda* environments to manage multiple Python installations and kernels in the experiments [23]. Conda is a package and environment management system that installs and manages the dependencies of packages. It simultaneously supports multiple versions of Python installed with different dependencies. Before executing each notebook, we prepared a conda environment with the declared Python version and installed all dependencies declared in the repository through the files mentioned above. In repositories that did not declare dependencies, we installed all anaconda [24] dependencies. Anaconda is both a conda package and a Python and R distribution that includes over 100 Scientific Packages, including `numpy`, `scipy`, and `matplotlib`. We set a time limit of 5 minutes for the execution of each notebook.

B. Data Collection

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 returned 265,143 repositories with 1,450,071 notebooks. We did not collect checkpoint notebooks that were stored in `.ipynb_checkpoints` directories.

Most repositories (59.97%) have 2 or fewer notebooks. Only 12.46% of the repositories have 10 or more notebooks. However, 61.45% of the notebooks belong to repositories with 10 or more notebooks.

After collecting the notebooks, we excluded invalid notebook files, empty notebooks, and empty repositories resulting in 1,423,676 notebooks from 264,023 repositories. From this result, we also excluded 264,510 (18.58%) duplicated notebooks. The goal was to reduce the bias towards forks and notebook copies. We detected these notebooks by calculating the SHA1 hashes from cell sources and output formats. We did not use the output results nor other metadata when we calculated the hashes to be able to detect notebooks that only had distinct retrospective data as duplicates. This resulted in 1,159,166 notebooks for the analyses.

We then analyzed the declared programming languages. Figure 3 presents, in log scale, the 15 most declared programming languages we found. Python is by far the most used programming language, corresponding to 93.32% of the notebooks. It is followed by R (1.31%) and Julia (0.93%). 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 43,204 notebooks do not declare a programming language, and 33,378 of them use *nbformat* lower than 4, which predates the release of the language-agnostic Jupyter. Although this is

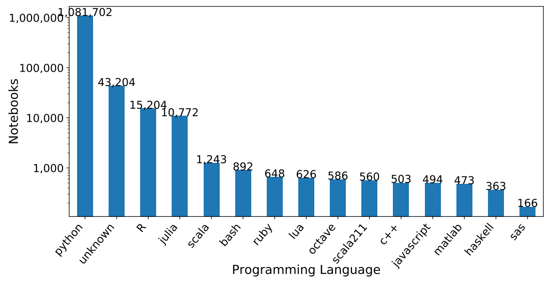


Fig. 3. Top 15 most declared programming languages. Notebooks axis in logarithmic scale.

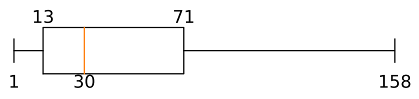


Fig. 4. Distribution of maximum execution counter. Max. outlier: 11,037.

a strong indication that these notebooks also use Python, we opted for removing them from Python-specific analyses.

Since most notebooks contain Python code (1,081,702) 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 scripts, and we used the Python AST to extract Python constructs and imported modules. The most used version is Python 2.7, which corresponds to 36.00% of the notebooks. However, by combining major releases, Python 3 surpassed Python 2. In fact, 63.91% 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 1,005,689 (86.76%) 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 863,878 (74.53%) notebooks.

In addition to these restrictions, we analyzed only executed notebooks for RQ5 and RQ6. It corresponds to 985,595 (85.03%) notebooks. Figure 4 presents the distribution of the maximum execution counter value by notebooks. Note that 50% of the notebooks executed 30 or more cells.

IV. Results and Discussion

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

A. How are literate programming features used in notebooks?

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, notebooks have a median of 13 code cells. Note, however, that 30.93% of the notebooks have no markdown cell at all. In notebooks that have markdown cells, these cells concentrate at the beginning of the notebooks, as presented in Figure 5.

Among the 69.07% notebooks that have markdown cells, 50% have at least 26 meaningful markdown lines. Additionally, 50% of the notebooks have at least 168 meaningful words, which is a bit bigger than two times the size of this paragraph.

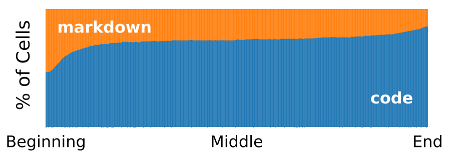


Fig. 5. Distribution of cell types in the notebook.

We consider meaningful words all the words that are not part of the markdown syntax. Similarly, meaningful lines are lines that have meaningful words. We also count stopwords as meaningful. Stopwords correspond to a median of 44 words in the notebooks for which we could detect the language.

English appears in the markdown cells of 87.16% (697,825) of notebooks with markdown cells. However, only 309,334 are solely in English. We detected other languages in cells of 46.44% (371,791) 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 38.17% of notebooks with markdown.

Finally, the most common markdown elements are headers (H1, H2, and H3), and paragraphs. These elements appear respectively in 90.71% and 79.53% of notebooks with markdown. Notebooks have a median of 18 words in all headers and 88 words in all paragraphs.

RQ1. *How are literate programming features used in notebooks?*
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.

Implications: Markdown plays a considerable role in notebooks, but the size of markdown cells may not be enough for well-described narratives, potentially compromising reproducibility. Their position indicates that users give more attention to the beginning of notebooks. Additionally, markdown could provide descriptions on how to reproduce the notebook. In both cases, reproducing the last cells may represent a challenge (in fact, we observe a decay of reproducibility in the last cells when answering **RQ7**).

B. How are notebooks named?

Only 1.99% of the notebooks start with "Untitled", and only 0.69% of the notebooks have "-Copy" in their names. A considerable number of notebooks (26.91%) have characters not recommended by the POSIX fully portable filenames guide. Many of these names do not cause problems for most systems, but 0.15% of the notebooks would not work on Windows. Since we used Linux to clone the repositories, we do not know how many titles Linux does not support, if any.

Figure 6 presents the length of filenames. Note that all notebooks finish with ".ipynb". We found 12 notebooks without title (i.e., their filename was just ".ipynb"). Excluding the extension, 50% of the notebooks have 16 letters or less. It corresponds to an average of 2 to 3 English words.

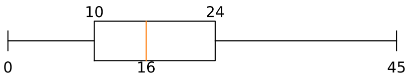


Fig. 6. Distribution of filename lengths. Max. outlier: 187.

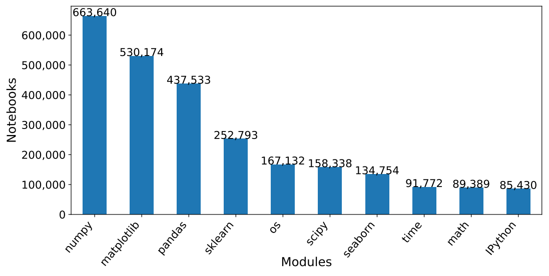


Fig. 7. Top 15 most imported modules.

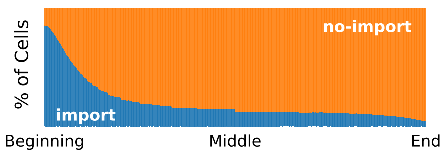


Fig. 8. Distribution of imports in notebooks.

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 not long names. On the other hand, a considerable number of users do not seem to care about OS-based restrictions and conventions in naming files.

Implications: Although caring about the title is important for the narrative, not caring about OS-based restrictions may hamper the reproducibility on other operating systems.

C. How do notebooks use modules, functions, and classes?

To answer this question, we analyzed the AST of all 1,005,689 valid Python notebooks. While 91.25% of them had imports, only 10.30% of them had local imports (i.e., imports of modules defined in the repository directory). Figure 7 presents the top 15 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 15, but in a much lower number of notebooks. Figure 8 presents the distribution of cells with imports in notebooks. Note that most imports occur at the beginning of notebooks. In Python scripts, the official Python style guide (PEP 8) recommends writing imports at the top of the files [25].

Next, we analyzed the AST constructs from notebooks to understand if they define functions and classes. Figure 9 presents the used AST constructs from valid Python notebooks. Note that only 53.94% of valid Python Notebooks define functions and only 8.54% define classes. While these numbers may indicate that notebooks discourage writing functions, we found that 70.90% of notebooks that have loops or condition structures also have function definitions.

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. In Python, it is common to evolve object-oriented code from a functional or imperative script [26].

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. On the one hand, local modules could be better explored to extend the reusability to other notebooks and scripts, and reduce the size of code cells in notebooks. On the other hand, keeping the code inside the notebook can be good for reproducibility, as it allows users to share only the notebook file with all code.

D. How are notebooks tested?

Only 15,473 (1.54%) 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 3,211 times at the time of this writing [27]. 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 reason why we found very few notebooks with tests may not be the notebook environment. As presented in Section IV-C, Jupyter Notebooks are mainly used for scientific software and data analytics. Testing this kind of software is hard due to the lack of oracles, the large number of tests required, and the difficulty of judging the number of tests [28].

RQ4. *How are notebooks tested?*
Answer: Very few notebooks import testing modules. Some tools have been implemented to enhance tests on Jupyter [29], [30], but we could not detect many uses of these tools. Moreover, they require modifying the notebook code in a way that may break the narrative.

Implications: There is an opportunity for improving tests on notebooks. As presented in Section IV-C, 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.

TABLE I
OUTPUT FORMATS IN CELLS AND NOTEBOOKS

Format	% of cells with output	% of executed notebooks
Text	95.79%	81.98%
Image	31.41%	51.00%
HTML/JS	22.90%	36.86%
Error	3.22%	14.90%
Formatted	1.78%	1.93%
Extension	1.04%	0.00%
PDF	0.12%	0.12%

E. Do users store notebooks with retrospective data?

As stated in Section III-B, we collected 985,595 executed notebooks, which corresponds to 85.03% of the notebooks. These notebooks have retrospective data.

Among the executed notebooks, 26.69% of the cells had an output, and 85.28% of the notebooks had at least one cell with an output. Table I presents the percentage of cells and notebooks with each output format. 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, *Image* represents PNG, JPEG, and SVG formats, which are the default image formats supported by Jupyter. About 51.00% of executed notebooks displayed an image. *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 notebooks have outputs in cells.

RQ5. *Do users store notebooks with retrospective data?*
Answer: Most notebooks store cells with outputs.

Implications: This result fosters reproducibility. Knowing the expected output allows users to re-run notebooks and check if they reproduce the results. Additionally, the number of notebooks with images and rich-media indicates that users use the retrospective data to enhance the notebook narratives.

F. How are notebooks executed?

Among the 1,053,653 executed notebooks, 21.11% had non-executed code cells, and 62.08% had empty cells. Figure 10 presents the distribution of code cells in the notebooks. Note that the percentage of executed code cells drops towards the end of notebooks, while the percentage of non-executed and empty cells grows. While 59.07% of executed notebooks finish with empty cells, only 11.33% of executed notebooks have empty cells among non-empty ones.

We collected 912,343 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 86.59% of the executed notebooks. Among the notebooks with unambiguous execution order, 36.36% have cells out-of-order.

By following the sequence of execution counters in unambiguous execution order notebooks, we counted how many

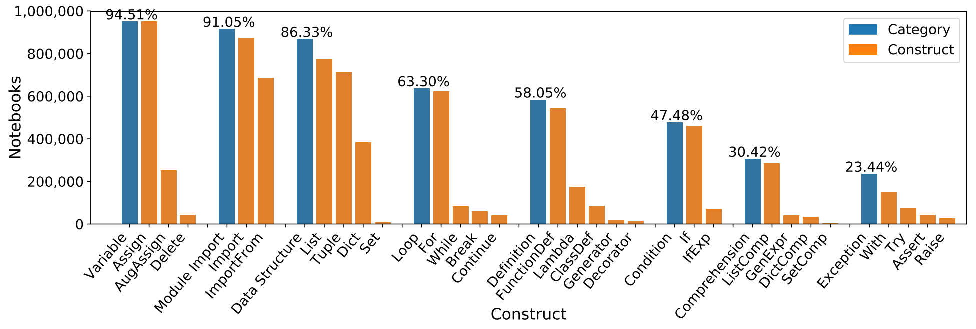


Fig. 9. Distribution of Python constructs in notebooks. This figure groups constructs into categories. The constructs of a category appear on the right of the category bar. A category corresponds to the union of its constructs.

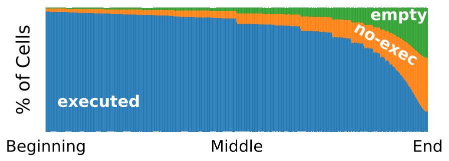


Fig. 10. Distribution of code cells in notebooks.

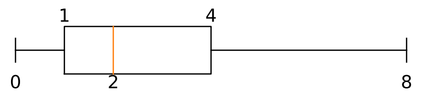


Fig. 11. Distribution of skips. Max. outlier: 220.

skips occurred. Since skips represent cell executions without explicit definitions, they may indicate the presence of hidden states. Figure 11 presents the distribution of skips by notebooks. 76.90% of unambiguous execution order notebooks have at least one skip. A skip contains 12.82 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 66.08%. Additionally, the average of skipped executions drops to 10.32.

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 count. 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 of these skips only at the beginning of a notebook is the re-execution of all of its cells without restarting the kernel.

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 [31]. Fortunately, most of these code smells are easily fixable by restarting the kernel and executing all cells again before committing. Nonetheless, such approach could detect out-of-order cells by looking not only to cell numbers but also to variable usages occurring before the definition.

G. How reproducible are notebooks?

Only 149,259 notebooks belong to repositories that declare module dependencies. Most of these repositories use *requirements.txt* (10.04%), while 5.98% use *setup.py*. Among these, many repositories (3.23%) 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 1,541 notebooks that belong to repositories with *Pipfile*.

In the remainder of this section, we describe a reproducibility study in which we executed all 863,878 Python notebooks with unambiguous execution order. Among these, 118,483 (13.72%) declared dependencies using the files mentioned above. Not all dependency declarations are valid. We attempted to install the dependencies of these notebooks in conda environments. However, the dependencies of 75,059 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 rate for these files were 67.55%, 61.17%, and 65.20%, 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, exists for less than two years, and its specification still goes through constant revisions.

Among the reasons for installation errors, we identified that 35.04% have files that require other unavailable files (e.g., sub-requirements and downloads from unavailable servers), 24.77% have malformed files (i.e., wrong syntax or conflicting dependencies), 25.67% have files that require a previous installation of Python packages (e.g., a *setup.py* that requires Cython to compile and build a package), 8.73% have files that require external tools (e.g., compilers and libraries), 4.77% have files designed for other systems (e.g., Raspberry Pi and Windows), and 1.02% 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).

We were able to install the dependencies of 43,424 notebooks. In addition to these notebooks, we prepared anaconda environments for 745,389 notebooks that did not declare dependencies. Different from the previous conda environments,

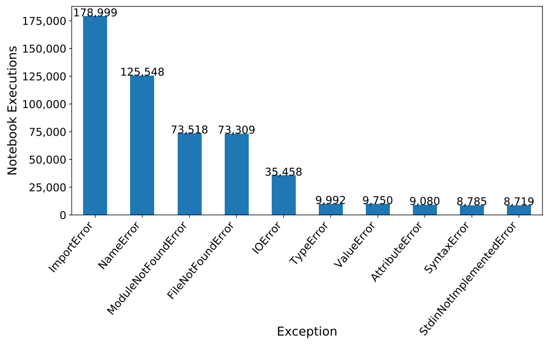


Fig. 12. Top 10 most common exceptions.

an anaconda environment comes with a comprehensive set of scientific Python packages, such as *numpy*, *matplotlib*, and *pandas*. Combining both sets, we had 788,813 notebooks for the reproducibility study. However, we excluded 32 of them because they had corrupted files.

Many notebooks failed to execute all the cells. A total of 9,982 notebooks failed because their execution exceeded a time limit of 5 minutes, while 570,476 notebooks failed due to an exception. Figure 12 presents the 10 most common exceptions the notebooks presented in our assessment.

29.23% of the notebook executions failed due to *ImportError* and *ModuleNotFoundError* exceptions. These exceptions are related to missing dependencies. Surprisingly, 45.18% of the notebooks from repositories with declared dependencies failed with one of these errors, while only 31.24% of the notebooks from repositories without declared dependencies failed with these errors. It probably occurred because we used bloated anaconda environments for the latter ones. Still, it indicates that many dependency files do not declare all the notebook dependencies.

Another very common exception was *NameError* (14.53% of notebook executions). This exception occurs when Python tries to access a variable that was not defined. While this exception is related to hidden states and out-of-order cells, the experiment design may also cause this issue. In the reproducibility study, we executed cells following their execution order and not a more natural top-down cell definition order. We opted for the former order because notebook users can execute cells at any order and skip the execution of some cells. Note that 21.11% of Python notebooks had non-executed code cells that could produce distinct result should we execute them by following the top-down order. The execution order may also be the reason for the meager rate of notebooks that execute and produce the same results (4.03%). Additionally, the lack of tests, the presence of hidden states, among many other factors, may influence this result.

Finally, the other very common exceptions were *FileNotFoundError* and *IOError* (12.59% of the notebooks). These errors occur when users use absolute paths to access data files or do not include the data in the repositories.

Only 208,323 notebooks finished their executions successfully. This represents 24.11% of the notebooks we attempted to reproduce. Despite being able to execute these notebooks

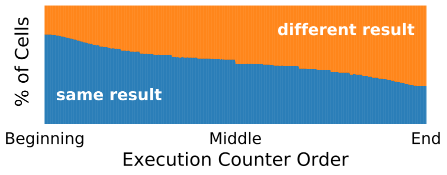


Fig. 13. Distribution of cell reproducibility.

following the cell execution order, 173,487 of them produced different results. The percentage of finished executions is very close to the reproducibility rate of 24.9% that Collberg et al. [32] achieved in their study of reproducibility in general computer systems research. In their study, they did not check the validity of results. Thus, we cannot compare this rate with the rate of notebooks that produce the same results.

Figure 13 presents the distribution of cell reproducibility in notebooks that finishes the execution. The position in this figure refers to the execution order and not to the cell position in the notebook. Note that different results appear more frequently towards the end of the notebooks. The area of *same result* is slightly bigger than the area of *different result* in this figure. It indicates that more cells produced the same results than the opposite. However, by selecting only notebooks that failed to produce the same results, we calculated that 50% of them have distinct results in more than 53% of their cells.

RQ7. *How reproducible are notebooks?*
Answer: We were able to successfully run 24.11% of the unambiguous execution order Python notebooks. This number is close to the results of a previous reproducibility study [32] about general computer systems research (24.9%). However, the rate is way smaller (4.03%) 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.

Implications: While the reproducibility rate is comparable to the rate in general computer systems research [32], it is far from ideal. The identification of the root causes indicates 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 execution order could improve the reproducibility rate. It is worthy of note that dependency resolution problems are also common in other contexts, such as building past snapshots of software [33]. Additionally, there are tools such as ReproZip [34] that automatically capture dependencies (both libraries and data) and create packages including these dependencies, thus ensuring reproducibility. ReproZip has a plugin for Jupyter [35].

V. Best Practices for the Reproducibility of Notebooks

In Section IV, 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.

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 IV-B, 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 or can have code cells executed or removed. Users seem to pay more attention to the beginning of the notebook, as depicted in Section IV-A, Section IV-C, and Section IV-F. 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 IV-C, 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 IV-G, 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.

5. Use a clean environment for testing the dependencies to check if all of them are declared. In Section IV-G, we identified that installing dependencies in a clean environment failed more than just using a bloated anaconda environment. 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 [25] 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 IV-G. Accessing project files using relative paths can reduce this issue.

8. Re-run notebooks top to bottom before committing. As presented in Section IV-F, many notebooks have out-of-order cells and skips. Moreover, these issues seem to impact the reproducibility (Section IV-G). Thus, we recommend re-running notebooks for restoring the execution counters and minimizing the impact of hidden states and out-of-order cells.

VI. Threats to Validity

This study attempts to obtain a picture of quality and reproducibility practices used in the design of Jupyter Notebooks. As presented in Section III, 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 reproducibility study, 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 the preparation and execution of another notebook.

Additionally, we examined all notebooks from GitHub as valid subjects in this work. We did not account for the perils of mining software repositories from GitHub [36]. 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 [37]. A basic check for notebooks containing words related to exercises ("assignment", "course", "exercise", "homework", "lesson") returns 253,008 non-duplicated notebooks (21.83%). 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 deserves 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.

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 [20].

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 scratch pad notebook and follow better practices. For instance, Grus [7] pointed out the problem of Untitled notebooks, but in our data, these notebooks correspond only to 1.99% of the notebooks.

VII. Related Work

Neglectos [14] 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 Figure 7). 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 a distinct order. They show *warnings* and *collections* in the top 10, while we indicate *seaborn* and *time*.

Kery et al. [13] interviewed 21 data scientists and surveyed 45 data scientists to understand how they use notebooks. They identified 3 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. [13] 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. [13] 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. [3] 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 is to extract insights on the usage of notebooks, our goal is to dive into evidence 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 3 most imported Python modules.

In the second analysis, Rule et al. [3] 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. [3] interviewed 15 data analysts that recognized the importance of cleaning and annotating notebooks and indicated 4 reasons for reusing a notebook: tracking provenance, code reuse, reproducibility of experiments, and presentation of results. Our results suggest that the reproducibility of notebooks is far from ideal with only 4.03% 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 [22], [38], [39].

Unlike prior work, we focus on the quality and reproducibility of Jupyter Notebooks and try to identify (and quantify the use of) practices that hinder reproducibility.

VIII. Conclusion

This paper has three main contributions. First, it analyzes evidence of good and bad practices on the development of Jupyter Notebooks regarding reproducibility by going through the main criticisms that the format receives [6]–[8]. Second, it presents a full reproducibility study that measures the reproducibility rate of notebooks (**RQ7**). Finally, it proposes a set of good practices that intends to minimize the criticisms and raise the reproducibility rate (Section V).

In our results, we found at the same time evidence of 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 high 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.

Despite discussing and analyzing many criticisms of notebooks in this paper, we did not discuss all of them. Other criticisms relate to the versioning [6], [8], security risks [6], lack of IDE features [7], [8], lack of support for long asynchronous tasks [8], and lock-in aspects of Jupyter [7].

In the reproducibility study, we found that most repositories do not declare their dependencies. Among the ones that do, many do not declare all their dependencies. We were only able to execute 24.11% of the notebooks that we attempted to run without exceptions, and only 4.03% produced the same results. As future work, we intend to infer the dependencies of notebooks and run them using different orders to check whether the reproducibility rate can be improved.

We propose a set of practices that intends to raise the reproducibility rate of notebooks on the current state-of-the-practice. At the same time, we foresee the development of approaches for reducing the bad practices as future work. A tool that detects imports and fills up requirement files should raise the number of notebooks with declared modules and reduce the number of undeclared dependencies. A notebook linting tool should be able to detect hidden states and out-of-order cells. Finally, a cleaning tool could help to transform scratchpad notebooks with bad practices into clean and reproducible notebooks for publishing.

This study is a first step towards assessing the quality and reproducibility of notebooks. There are many other questions that we plan to investigate in the future. We intend to investigate strategies to assess the different types of projects (e.g., student notebooks, tutorial notebooks, research notebooks, scratchpads, dashboards, and others) to compare their metrics. We also plan to extend the modularization analysis (**RQ3**) to check the prevalence of copy and paste of code, instead of the usage of functions. We foresee the comparison of the notebooks to general-purpose scripts to understand whether one or another have better quality and reproducibility measures. Finally, we intend to continue the research with qualitative studies to understand the reasoning behind some phenomena observed in this study.

The data, scripts, and notebooks used in this study are available at https://doi.org/10.5281/zenodo.2592524.

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

References

[1] D. E. Knuth, "Literate programming," *The Computer Journal*, vol. 27, no. 2, pp. 97–111, 1984.

[2] H. Shen, "Interactive notebooks: Sharing the code," *Nature News*, vol. 515, no. 7525, p. 151, 2014.

[3] A. Rule, A. Tabard, and J. D. Hollan, "Exploration and explanation in computational notebooks," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, ser. CHI '18. New York, NY, USA: ACM, 2018, pp. 32:1–32:12. [Online]. Available: http://doi.acm.org/10.1145/3173574.3173606

[4] F. Pérez and B. E. Granger, "Ipython: a system for interactive scientific computing," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 21–29, 2007.

[5] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. E. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. B. Hamrick, J. Grout, S. Corlay *et al.*, "Jupyter notebooks ? a publishing format for reproducible computational workflows," in *ELPUB*, F. Loizides and B. Scmidt, Eds. Gttingen, Germany: IOS Press, 2016, pp. 87–90. [Online]. Available: https://eprints.soton.ac.uk/403913/

[6] K. Pomogajko. (2015) Why I Don't Like Jupyter (FKA IPython Notebook). [Online]. Available: https://yihui.name/en/2018/09/notebook-war/

[7] J. Grus, "I don't like notebooks." 2018, jupyterCon. [Online]. Available: https://conferences.oreilly.com/jupyter/jup-ny/public/schedule/detail/68282

[8] A. Mueller. (2018) 5 reasons why jupyter notebooks suck. [Online]. Available: https://towardsdatascience.com/5-reasons-why-jupyter-notebooks-suck-4dc201e27086/

[9] G. Wilson, D. A. Aruliah, C. T. Brown, N. P. Chue Hong, M. Davis, R. T. Guy, S. H. D. Haddock, K. D. Huff, I. M. Mitchell, M. D. Plumbley, B. Waugh, E. P. White, and P. Wilson, "Best practices for scientific computing," *PLOS Biology*, vol. 12, no. 1, pp. 1–7, 01 2014. [Online]. Available: https://doi.org/10.1371/journal.pbio.1001745

[10] W. L. Hürsch and C. V. Lopes, "Separation of concerns," Northeastern University, Tech. Rep., 1995.

[11] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*. John Wiley & Sons, 2011.

[12] S. Horwitz and T. Reps, "The use of program dependence graphs in software engineering," in *Proceedings of the 14th international conference on Software engineering*. ACM, 1992, pp. 392–411.

[13] M. B. Kery, M. Radensky, M. Arya, B. E. John, and B. A. Myers, "The story in the notebook: Exploratory data science using a literate programming tool," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, ser. CHI '18. New York, NY, USA: ACM, 2018, pp. 174:1–174:11. [Online]. Available: http://doi.acm.org/10.1145/3173574.3173748

[14] Neglectos. (2018) A preliminary analysis on the use of python notebooks. [Online]. Available: https://blog.bitergia.com/2018/04/02/a-preliminary-analysis-on-the-use-of-python-notebooks/

[15] J. Freire, D. Koop, E. Santos, and C. Silva, "Provenance for computational tasks: A survey," *Computing in Science & Engineering*, vol. 10, no. 3, pp. 11–21, 2008.

[16] V. Arnaoudova, M. Di Penta, and G. Antoniol, "Linguistic antipatterns: What they are and how developers perceive them," *Empirical Software Engineering*, vol. 21, no. 1, pp. 104–158, 2016.

[17] Microsoft. (2018) Naming files, paths, and namespaces. Windows Dev Center. [Online]. Available: https://docs.microsoft.com/en-us/windows/desktop/FileIO/naming-a-file

[18] Tim and Doorknob. (2014) Is space not allowed in a filename? Unix & Linux. [Online]. Available: https://unix.stackexchange.com/q/148043

[19] D. Lewine, *POSIX programmers guide*. "O'Reilly Media, Inc.", 1991.

[20] P. Wiki. (2019) Python testing tools taxonomy. [Online]. Available: https://wiki.python.org/moin/PythonTestingToolsTaxonomy

[21] T. Staley. (2017) Making git and jupyter notebooks play nice. [Online]. Available: http://timstaley.co.uk/posts/making-git-and-jupyter-notebooks-play-nice/

[22] D. Koop and J. Patel, "Dataflow notebooks: encoding and tracking dependencies of cells," in *9th USENIX Workshop on the Theory and Practice of Provenance (TaPP 17). USENIX Association*. Seattle, Washington: USENIX, 2017, pp. 1–7.

[23] I. Anaconda. (2017) Conda documentation – managin environments. [Online]. Available: https://conda.io/docs/user-guide/tasks/manage-environments.html

[24] Anaconda. (2018) Anaconda software distribution. [Online]. Available: https://www.anaconda.com

[25] G. van Rossum, B. Warsaw, and N. Coghlan. (2001) Pep 8: style guide for python code. [Online]. Available: https://www.python.org/dev/peps/pep-0008/

[26] N. Vavrová and V. Zaytsev, "Does python smell like java?" *The Art, Science and Engineering of Programming (Programming)*, vol. 1, no. 2, pp. 11–1, 2017.

[27] Udacity. (2017) Deep learning nanodegree foundation. [Online]. Available: https://github.com/udacity/deep-learning

[28] D. Hook and D. Kelly, "Testing for trustworthiness in scientific software," in *ICSE Workshop on Software Engineering for Computational Science and Engineering, SE-CSE 2009, Vancouver, BC, Canada, May 23, 2009*. IEEE Computer Society, 2009, pp. 59–64. [Online]. Available: https://doi.org/10.1109/SECSE.2009.5069163

[29] T. Burns and G. Ward. (2013) ipython-nose. [Online]. Available: https://github.com/taavi/ipython_nose

[30] J. F. Pimentel. (2016) ipython-unittest. [Online]. Available: https://github.com/JoaoFelipe/ipython-unittest

[31] V. Garousi and B. Küçük, "Smells in software test code: A survey of knowledge in industry and academia," *Journal of Systems and Software*, vol. 138, pp. 52–81, 2018.

[32] C. Collberg, T. Proebsting, G. Moraila, A. Shankaran, Z. Shi, and A. M. Warren, "Measuring reproducibility in computer systems research," Department of Computer Science, University of Arizona, Tech. Rep., 2014.

[33] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "There and back again: Can you compile that snapshot?" *Journal of Software: Evolution and Process*, vol. 29, no. 4, p. e1838, 2017.

[34] F. S. Chirigati, D. E. Shasha, and J. Freire, "Reprozip: Using provenance to support computational reproducibility." in *TaPP*, 2013.

[35] N. Y. University. (2017) Making Jupyter Notebooks Reproducible with ReproZip. [Online]. Available: https://docs.reprozip.org/en/1.0.x/jupyter.html

[36] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining github," in *Proceedings of the 11th working conference on mining software repositories*. ACM, 2014, pp. 92–101.

[37] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating github for engineered software projects," *Empirical Software Engineering*, vol. 22, no. 6, pp. 3219–3253, 2017.

[38] J. F. N. Pimentel, V. Braganholo, L. Murta, and J. Freire, "Collecting and analyzing provenance on interactive notebooks: when ipython meets noworkflow," in *Workshop on the Theory and Practice of Provenance (TaPP)*. Edinburgh, Scotland: USENIX, 2015, pp. 155–167.

[39] S. Samuel and B. König-Ries, "Provbook: Provenance-based semantic enrichment of interactive notebooks for reproducibility," in *The 17th International Semantic Web Conference (ISWC)*, ser. ISWC. Monterey, California, USA: Springer, 2018, pp. 1–4.