

A STUDY OF JUSTIFICATION ON JUPYTER NOTEBOOK
QUALITY & FAIRNESS

BY
KAI SUN, M.Eng.

A REPORT
SUBMITTED TO THE COMPUTING AND SOFTWARE
AND THE SCHOOL OF GRADUATE STUDIES
OF McMASTER UNIVERSITY
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTERS OF ENGINEERING

© Copyright by Kai Sun, Feb 2024

All Rights Reserved

Masters of Engineering (2024)
(Computing and Software)

McMaster University
Hamilton, Ontario, Canada

TITLE: A study of Justification on Jupyter Notebook quality &
fairness

AUTHOR: Kai Sun
M.Eng. in Computing and Software,
McMaster University, Hamilton, Canada

SUPERVISOR: Dr. Sébastien Mosser

NUMBER OF PAGES: xiii, 75

Lay Abstract

Jupyter Notebooks have become essential tools for data scientists and machine learning engineers. However, their diverse tools and languages present challenges in ensuring consistent quality and reproducibility. Current research on maintaining notebook quality is limited, and there's a need to turn theoretical notebook best practices into practical quality checks. This report explores using the Justification Diagram Language to make these best practices checkable and implementable. This approach not only makes it easier for developers to apply these practices but also demonstrates their effectiveness in real-world situations, ultimately enhancing the quality of notebooks.

Abstract

Computational notebooks using the Jupyter Platform have become increasingly popular among data scientists and machine learning engineers. However, due to the diversity of tools and languages, Notebook developers face reproducibility challenges. Limited research has been conducted on verifying the quality of the notebook, and only a few studies have established best practices for notebook development. Thus, further study is needed to transform those conceptual best practices into concrete, actionable quality checks to ensure the quality & fairness of the notebook. This report aims to improve the quality of notebooks by investigating the possibility of using the Justification Diagram Language to convert best practices into tangible steps of quality checks that users can execute within the Continuous Integration and Deployment (CI/CD) pipeline. The report will focus on justifying 12 notebooks' best practices collected from existing studies using Justification Diagrams, and it will then map these diagrams into practical steps that can be run in GitHub Actions, demonstrating their effectiveness in real-life scenarios and thus enriching the quality of the notebooks.

Acknowledgements

I would like to start by expressing my deepest appreciation to my supervisor, **Dr. Sébastien Mosser**, for his exceptional guidance. Your wisdom has been a guiding light throughout my master's studies, and your thorough reviews have significantly enhanced my work. The resources and support you provided were vital to my growth.

Special thanks to my friend **Deesha Patel**. Your assistance in completing this report and sharing your experience have been instrumental in accelerating my project.

To my parents, I am deeply grateful for your unwavering support and financial backing. Your constant belief in me has been the foundation upon which I've built my achievements.

Lastly, I extend my thanks to McMaster University for the opportunity to pursue my studies in such a stimulating environment. This experience has been transformative, and I am thankful for every lesson learned.

Contents

Lay Abstract	iii
Abstract	iv
Acknowledgements	v
Abbreviations	xiii
1 Introduction	1
2 Background	4
2.1 Jupyter Notebooks	4
2.1.1 Introduction to Jupyter Notebook	4
2.1.2 Reproducibility Crisis	5
2.1.3 Pynblint	6
2.2 DevOps	7
2.3 CI/CD	8
2.4 Justification Diagram	10
2.4.1 Introduction to Justification Diagram	11
2.4.2 Representation of Justification Diagram	11

3	Notebook Best Practices and Justification	15
3.1	Execution Reproducibility	16
3.1.1	Linear Execution Order	16
3.1.2	Beginning Imports	18
3.1.3	Pinned Dependencies	18
3.1.4	Virtual Environment	20
3.2	High-Quality Code	21
3.2.1	Meaningful Name	22
3.2.2	PEP8 Standard	22
3.2.3	Relative Path	23
3.2.4	Notebook Testing	25
3.3	Literate Programming Paradigm	27
3.3.1	Code Document	27
3.3.2	Markdown Headings	28
3.4	Clean and Concise	29
3.4.1	Conciseness	29
3.4.2	Tidiness	30
3.5	Conclusion	31
4	Quality Check Implementations	33
4.1	GitHub Workflow Implementations	34
4.2	Implementation Separation	38
4.3	Conclusion	40
5	Operational Justification Diagram Language	41

5.1	Introduction	42
5.2	Eliciting Reusable Operations	43
5.3	Syntax	46
5.4	Example	47
5.5	Conclusion	49
6	Conclusion and Future work	50
6.1	Summary	50
6.2	Future work	51
A	Notebook Best Practices	53
A.1	Linear execution order	53
A.2	Beginning Imports	53
A.3	Pinned Dependencies	54
A.4	Virtual Environment	54
A.5	Meaningful Name	54
A.6	PEP8 Standard	55
A.7	Relative Path	56
A.8	Notebook Testing	56
A.9	Code document	57
A.10	Markdown Headings	58
A.11	Conciseness	58
A.12	Tidiness	59
B	Implementations	61
B.1	Check_file_exist.sh	61

B.2	Check_functions_defined.sh	62
B.3	Check_test_modules.py	64
B.4	Pynblint	67
B.5	Coverage	68
B.6	Nbconvert	70

List of Figures

2.1	Jupyter Notebook sample	5
2.2	Pynblint output sample	7
2.3	CI/CD structure	9
2.4	Python CI/CD pipeline using GitHub workflow	10
2.5	Justification Diagram for training a ML/DL model	12
2.6	Example .jd file	14
3.1	Justification Diagram for linear execution order	17
3.2	Justification Diagram for beginning imports	19
3.3	Justification Diagram for pinned dependencies	20
3.4	Justification Diagram for virtual environment	21
3.5	Justification Diagram for meaningful name	23
3.6	Justification Diagram for pep8 standard	24
3.7	Justification Diagram for relative path	25
3.8	Justification Diagram for notebook-testing	26
3.9	Justification Diagram for script-testing	27
3.10	Justification Diagram for script-testing	28
3.11	Justification Diagram for markdown headings	29
3.12	Justification Diagram for conciseness	30

3.13	Justification Diagram for tidiness	31
4.1	Mappings of Justification Diagrams to quality test steps for notebook testing	34
4.2	YAML file of GitHub workflow testing the usage of Pytest within notebooks	36
4.3	Implementations for Unittest module	39
5.1	Implementation for environment setting	47
5.2	Implementations for quality test: notebook-testing	49
A.1	Example of linear execution order	54
A.2	Example of beginning imports	55
A.3	Example of requirements.txt	55
A.4	Example of meaningful name	56
A.5	Example of pep8 standard	56
A.6	Example of notebook testing	57
A.7	Example of code document	58
A.8	Example of markdown headings	59
A.9	Example of tidiness	60
B.1	check_file_exist.sh	62
B.2	Check_functions_defined.sh	63
B.3	JSON output of Pynblint	63
B.4	check_test_modules.py	66
B.5	Example of Pynblint JSON lint result	69
B.6	Coverage report	69

List of Tables

3.1	Configuration files for self-contained environments	21
4.1	Scripts and tools used in notebook testing	37
5.1	Operations	45
5.2	Keywords	46
B.1	List of functions used in the script	64
B.2	List of Pynblint Lint rules [28]	68

Abbreviations

Abbreviations

AI	Artificial Intelligence
ML	Machine Learning
DL	Deep Learning
CI	Continuous Integration
CD	Continuous Delivery
YAML	YAML Ain't Markup Language
JD	Justification Diagrams

Chapter 1

Introduction

Jupyter Notebook, a notebook-wise computational environment under Project Jupyter, has been widely adopted by various groups, including students, researchers, and engineers. As of April 19, 2023, there are over 10 million notebooks in GitHub [17]. Unfortunately, with this level of popularity, there are no universally accepted notebook guidelines to follow. Thus, many people use notebooks according to their style preference without considering their capacity to be shared. As a result, reproducibility [4] has become a concern in notebooks these days, as there is a high motivation for notebooks to function properly in the environments of other researchers. Some research papers have worked on extracting best practices of notebook development from a massive amount of high-quality notebooks on GitHub [20, 21, 26, 29], aiming to improve the general quality & fairness of notebooks and reduce the communication gap among those who use notebooks as their primary tool in their work. Fairness stands for the ability of data reuse [31]; for the purpose of this report, it relates to the reusability of notebooks. Even though there are analyses summarizing best practices

from a large sample of notebooks, no studies were found on tools for testing the quality & fairness of notebooks. The only relevant tool, Pynblint [28], is an open-source notebook quality analysis tool published on GitHub last year. It is a linter that can only analyze notebooks' content and offer recommendations for quality enhancement based on best practices, but it does not justify the quality & fairness of notebooks, which means it does not provide the reasoning behind these recommendations or explain why they could lead to quality enhancement. This tool stopped being updated in 2023, leaving its progress incomplete.

There is a strong need to demonstrate the rationales behind these best practices and educate notebook users on why they should apply them in their notebooks. To address the problems mentioned above, we proposed a solution to use Justification Diagrams (JD) to transfer best practices into quality tests that can be integrated into the Continuous Integration and Deployment (CI/CD) pipeline. This solution of using JD could provide strong reasoning for how our proposed quality tests can enhance the quality of notebooks. In the Background section, more details about Jupyter Notebooks, DevOps, CI/CD and Justification Diagrams will be provided.

This report focuses on three questions:

1. **How** can best practices be converted into actionable steps using Justification Diagrams?
2. **How** can justification diagrams be transferred into quality checks in CI/CD pipelines?
3. **How** can we improve the above two steps by simplifying the translation between Justification Diagrams and quality tests?

To tackle the above three questions, we will first outline the best practices collected from existing research papers and blogs. Then, we will apply JD to those practices and show how the JD can help organize abstract best practices into concrete notebook quality tests. Subsequently, we will map those diagrams to actual implementations in GitHub workflows and propose the Operational Justification Diagram Language that simplifies the translation of those diagrams into GitHub workflow implementations, providing a practical usage for notebook quality & fairness checks.

Chapter 2 will provide essential background knowledge for readers to understand this study's context. Chapter 3 will describe notebook best practices and their corresponding JD, which answers question 1. Chapter 4 will choose one of the best practices as the case study for showing the actual implementations of quality checks in GitHub Actions, which solves question 2. Additionally, Chapter 4 will discuss the possibility of having multiple implementations for an existing JD. Chapter 5 will answer question 3 by proposing an Operational Justification Diagram Language that simplifies the translation from JD to quality test implementations. Finally, the report ends with a summary and an overview of potential future work in Chapter 6.

Chapter 2

Background

This section will introduce some relative background knowledge needed to understand this report, including Jupyter Notebooks, DevOps, CI/CD and Justification Diagram.

2.1 Jupyter Notebooks

This section starts by describing what Jupyter Notebooks are. It then mentions the current challenge of reproducibility encountered by Notebook developers, and finally briefly introduces a Notebook analysis linter called Pynblint.

2.1.1 Introduction to Jupyter Notebook

Literate programming is a programming paradigm introduced in 1984 by Donald Knuth. It emphasizes the importance of writing code humans understand [12], intertwining source code and documentation into a single narrative. Computational notebooks are a direct implementation of some Literate programming concepts. They provide an interactive environment where users can write, execute and visualize code,

making them essential tools in statistics, data science, machine learning and computer algebra. Jupyter Notebook is one of the most popular computational notebooks nowadays, alongside other options such as Org Mode¹. It is open-source software that enables users to edit and execute notebook documents through a web browser. It comprises three types of cells: raw text cell, markdown cell and code cell. The code cell is the most commonly used cell for performing data analysis, machine learning or other computational tasks. The markdown cell is used for writing documentation. The raw text cell is used for writing plain text. A screenshot of Jupyter Notebook running Python code can be found in Figure 2.1. Users can execute the “add” function and view the output by clicking the run button.

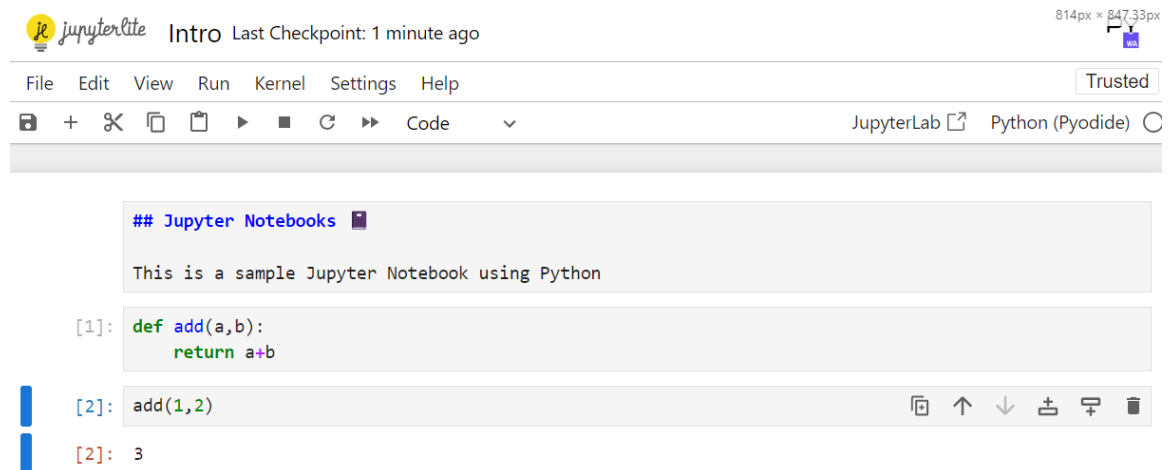


Figure 2.1: Jupyter Notebook sample

2.1.2 Reproducibility Crisis

The Reproducibility crisis has been found across many scientific research fields these days. Reproducibility is the ability of a researcher to re-obtain the results of a study

¹<https://orgmode.org/>

using the same code and input data as were used in the original experiment [4]. A 2016 survey [1] published in the Nature Journal found that about 70% of researchers failed to reproduce another scientist’s experiment, and more than half failed to reproduce their own experiments. This crisis has also extended to the field of Artificial Intelligence (AI) and Machine Learning (ML), where the lack of access to the source code and training conditions for ML models makes it difficult to reproduce existing publications [11]. At the same time, another reproducibility issues arise when using Jupyter notebooks for data analysis. One study found only 25% of Jupyter notebooks on GitHub are reproducible because of bad coding practices [20]. For instance, running a notebook requires the installation of dependencies first. Due to the varied computational environments of each user, the version of packages installed might differ from those in the authors’ environment, causing reproducibility problems. And that’s where the notebook best practice comes into play. In Chapter 3, more explanations of notebooks’ best practices that address the reproducibility issues will be provided.

2.1.3 Pynblint

Pynblint is a linter for Python Jupyter notebooks [28]. Linter are tools that analyze source code to flag programming errors and bad practices. Pynblint can discover potential defects in notebooks and provide recommendations. A sample output from Pynblint that lints the notebook in Figure 2.2 is illustrated below, including the statistics and the linting results. More explanations of Pynblint usage in quality tests will be provided in Chapter 4 and Appendix B.

```

***** PYNBLINT *****
NOTEBOOK: sample.ipynb
PATH: ./sample.ipynb

STATISTICS

Cells
Total cells: 2
Code cells: 1
Markdown cells: 1
Raw cells: 0

Markdown usage
Markdown titles: 1
Markdown lines: 1

Code modularization
Number of functions: 1
Number of classes: 0

LINTING RESULTS

(missing-opening-MD-text)
The initial notebook cells (i.e., the first 3 cells in the notebook) contain no Markdown text.
Recommendation: Begin your notebook by describing what you intend to do in one or more introductory Markdown cells.

(missing-closing-MD-text)
The final notebook cells (i.e., the last 3 cells in the notebook) contain no Markdown text.
Recommendation: Conclude your notebook by describing what you have accomplished in one or more concluding Markdown cells.

(non-executed-cells)
Non-executed cells are present in the notebook.
Recommendation: Re-run your notebook top to bottom to ensure that all cells are executed.
Affected cells: indexes[1]

In [ ]: def add(a,b):
        return a + b

```

Figure 2.2: Pynblint output sample

2.2 DevOps

The term “DevOps” was initially coined in 2008 and became widespread in 2009 [6]. It is a culture in software development that aims to bring a tighter relationship between Development (Dev) teams and Operations (Ops) teams within an organization. Traditionally, Dev and Ops teams were entirely isolated, often causing communication overheads [2]. DevOps introduces a series of practices that help with the software development cycle, such as increasing development speed by automating repetitive tasks like building, testing, and deploying software. Moreover, DevOps also improves development efficiency by monitoring changes, quickly resolving issues, and bridging gaps between dev and operation teams. Essentially, any practice that maintains software

quality regarding code and delivery mechanisms is considered a DevOps practice [6].

The same theory applies to notebook development; any practices benefiting notebook quality, such as reproducibility, readability or understandability, are considered as DevOps practices.

2.3 CI/CD

Continuous Integration and Continuous Delivery, commonly called CI/CD, are fundamental software development practices in DevOps. Continuous Integration often refers to techniques that regularly merge new or modified code into the existing code base, aiming to minimize delays between code commit and code build [9]. Continuous Delivery, following the steps of continuous integration, is the approach that allows developers to produce software in short cycles, ensuring that software is always deployable at any moment with minimal manual effort [7].

As shown in Figure 2.3, the CI process typically includes planning, coding, building and testing, whereas the CD stage involves releasing, deploying, operating and measuring the software.

In the CI stage, developers can frequently push their code changes to the repository several times daily. An automated build process is triggered immediately, or based on a predetermined timetable, to ensure that the source code is proper regarding executability. The next step is the testing phase, where testing tools can verify the software's functionality, ensuring the software meets engineers' expectations. These steps allow a team to rapidly develop their software with enhanced quality and fewer risks [9]. In the CD stage, developers can deploy their changes to the production environment often, allowing organizations to bring service improvement to the market

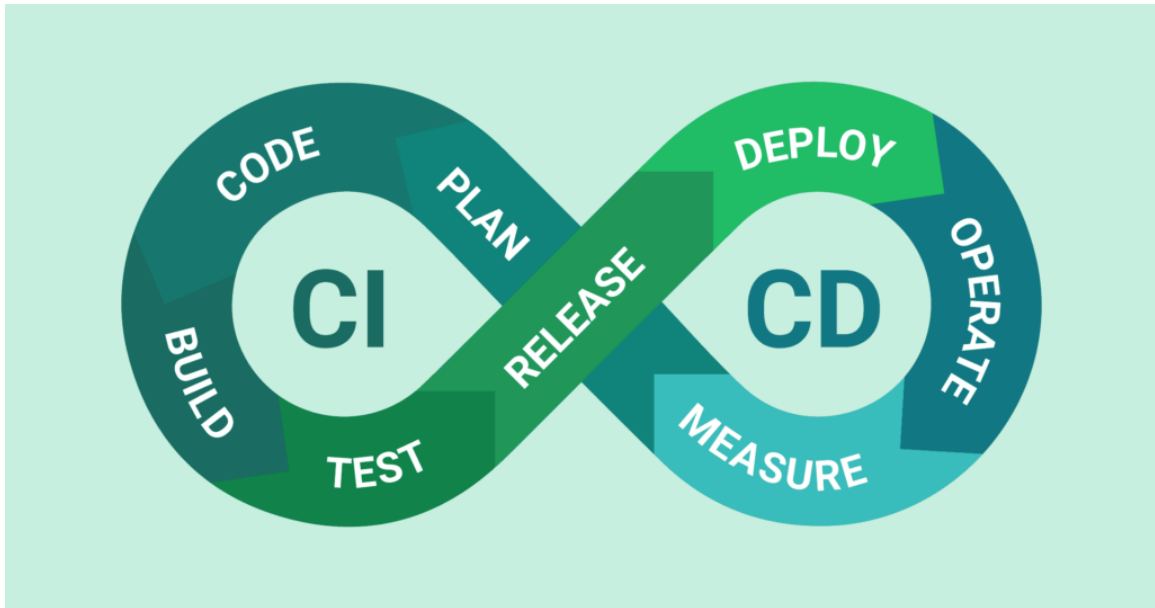


Figure 2.3: CI/CD structure [5]

and stay competitive [7].

GitHub Actions², a feature within GitHub, supports many phases in the CI/CD pipeline, including automated building, testing and deploying. GitHub Workflow³, a configurable automated process that sequences these individual Actions, is defined by a YAML file in the code repository and will be triggered manually or by an event in the repository. It helps organizations using GitHub accelerate their software development cycle and enhance software quality by adhering to DevOps principles. Figure 2.4 shows a sample YAML file of GitHub workflow, representing a simple Python CI/CD pipeline. The “**on**” keywords on line 3 indicate this pipeline’s triggering method. This pipeline example will capture any push events under the “**main**” branch and start the workflow. “**Jobs**” is the keyword that defines the steps of the workflow. This pipeline will create an Ubuntu environment, install Python 3.8 and Pylint, and

²<https://docs.github.com/en/actions>

³<https://docs.github.com/en/actions/using-workflows/about-workflows>

automatically execute the Pylint check on the `sample.py` file in the build phase. We will elaborate more about Jupyter Notebook quality checks implementations using GitHub workflow in Chapter 4.

```
! sample.yaml
1  name: Python
2
3  on:
4    push:
5      branches: [main]
6
7  jobs:
8    build:
9      runs-on: ubuntu-latest
10
11     steps:
12       - uses: actions/checkout@v4
13
14         - name: Set up Python
15           uses: actions/setup-python@v4
16           with:
17             python-version: 3.8
18
19         - name: Install Dependencies
20           run: |
21             python -m pip install --upgrade pip
22             pip install pylint
23
24         - name: Lint with pylint
25           run: |
26             pylint sample.py
27
```

Figure 2.4: Python CI/CD pipeline using GitHub workflow

2.4 Justification Diagram

This section will explain what is Justification Diagrams and show a real-life example that can be illustrated by the Justification Diagram Language.

2.4.1 Introduction to Justification Diagram

The Justification Diagram Language, inspired by Toulmin’s argumentation model, is designed to organize and visualize the key elements that validate a product’s outcomes [23]. During a development cycle, comprehensive documents that include input data, assumptions, and applied technologies are essential to explain the outcomes of a product. However, these documents are often not collected and presented formally because they lack concrete structures. The JD addresses this by listing those abstract documents in a structured fashion and showing each step of the reasoning process, resulting in a transparent process and an acceptable conclusion. Building on this, a study in DevOps demonstrated how the JD can justify CI/CD pipeline quality [25]. Similarly, this report will use the same techniques to justify the notebooks’ quality.

2.4.2 Representation of Justification Diagram

The Justification Diagram Language contains several key components: evidence, strategy, sub-conclusion and conclusion. Evidence is used to support the conclusion; it could be documents, source code, artifacts, or even a conclusion of another argumentation step (sub-conclusion). Strategy is the link between evidence and conclusion; it’s a method that explains why the conclusion is acceptable from our evidence. The conclusion is the final conclusion derived from input data or assumptions. A strategy can only have one conclusion, whereas a evidence can contribute to multiple strategies.

Figure 2.5 shows a Justification Diagram Language example, illustrating a general

process for training image classification models using computational notebooks. Before explaining the workflow, the language comprises four types of boxes: blue represents the evidence, green represents the strategy, white represents the sub-conclusion, and grey denotes the conclusion. For the workflow, three key components are necessary for training an ML/DL model for image classifications: a valid data set, a training environment, and the chosen training algorithms or model architecture. Researchers often prepare raw image files and corresponding labels to get a “*valid data set*”, so “*raw images*” and “*labels*” are the evidence of JD; the “*data processing*” step is the strategy that contributes to the sub-conclusion “*data set is ready*”. Next, “*setting up the training environment*” is crucial. This step includes installing the required “*ML/DL libraries*” within the notebook and ensuring “*CPU/GPU power is available*” for computational tasks, so they serve as the evidence, where “*setup environment*” as the strategy. The third step involves researchers selecting and understanding the specific algorithm or model architecture that will be used for training. Once these three keys are ready, they can start the strategy “*training the model*”, and finally, they can conclude that an image classification model is trained and prepared to be evaluated.

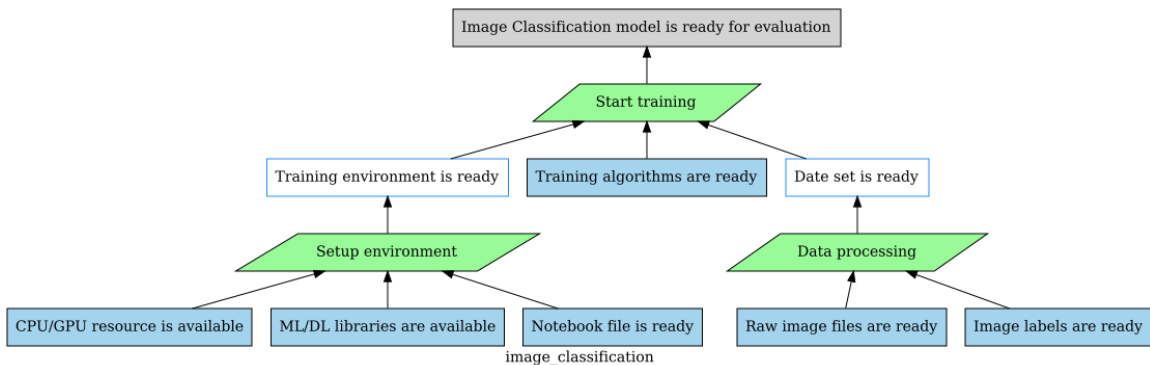


Figure 2.5: Justification Diagram for training a ML/DL model

Figure 2.6 is compiled from a ‘.jd’ file used in the JPipe compiler [15]. The ‘.jd’ file of Figure 2.5 can be seen in Figure 2.6. In the first line of the file, “image_classification” represents the name of this file as well as the title of the diagram. In the body of the file, each component is identified by an indicator keyword: “**evidence**”, “**strategy**”, “**sub-conclusion**” and “**conclusion**” to determine its type. Then, following the identifier, a variable name is needed for the component to build links with each other. The keyword “**is**” assigns text value to the variable. For example, “Su1” is an evidence variable representing “*Raw image files are available*”; “St1” is a strategy variable representing “*Data processing*”; “Sc1” is a sub-conclusion variable representing “*Data set is ready*”. To add a link between components, we need to use the “**supports**” keyword. It requires two arguments to establish a link in order, such as “Su1 supports St1”, meaning we need image files to start data processing.

```
1
2 justification image_classification {
3     evidence Su1 is "Raw image files are ready"
4     evidence Su2 is "Image labels are ready"
5     strategy St1 is "Data processing"
6     Su1 supports St1
7     Su2 supports St1
8
9     sub-conclusion Sc1 is "Date set is ready"
10    St1 supports Sc1
11
12    evidence Su3 is "ML/DL libraries are available"
13    evidence Su4 is "Notebook file is ready"
14    evidence Su5 is "CPU/GPU resource is available"
15    strategy St2 is "Setup environment"
16    Su3 supports St2
17    Su4 supports St2
18    Su5 supports St2
19
```

```
20     sub-conclusion Sc2 is "Training environment is ready"
21     St2 supports Sc2
22
23     evidence Su6 is "Training algorithms are ready"
24     strategy St3 is "Start training"
25     Sc1 supports St3
26     Sc2 supports St3
27     Su6 supports St3
28
29     conclusion C is "Image Classification model
30     is ready for evaluation"
31     St3 supports C
32 }
```

Figure 2.6: Example .jd file

Chapter 3

Notebook Best Practices and Justification

As discussed in the Background section, reproducibility issues were found in notebook developments [20]. Several studies have conducted experiments to recommend a set of best practices that users should follow to improve notebooks' quality. While these best practices are summarized, there is a lack of tools that allow researchers to apply them effectively and ultimately guide user practice toward better notebook development in real life. This gap highlights the need for research to justify these best practices and convert them to actionable quality tests. The Justification Diagram Language is an appropriate tool to justify the rationale behind those practices. And we can transform these diagrams into quality tests. Since these practices are independent, ideally, each practice should represent an individual quality test case, and users can choose to employ it freely.

This section explains 12 best practices collected from the state of the art [26] and focuses on answering the question: **“How can best practices be converted into**

actionable steps using Justification Diagrams?”. Each best practice will be further categorized into four themes and transformed into quality test steps under each section. The four themes are Execution Reproducibility in section 3.1, High-Quality Code in section 3.2, Literate Programming Paradigm in section 3.3, and Clean and Concise in section 3.4. All works in this section are collected in the repository, “*notebook-best-practices*” [16], serving as a book for users to understand each best practice and its test implementations. Please refer to Appendix A or the repository for a more precise description of each best practice.

3.1 Execution Reproducibility

Execution Reproducibility is critical in assuring good collaboration among scientists when working with notebooks. A well-written notebook should be easily re-executed or reproduced with the same output in different systems. In section 3.1, four corresponding best practices contribute to Execution Reproducibility: linear execution order, beginning imports, pinned dependencies and virtual environment.

3.1.1 Linear Execution Order

In Jupyter notebooks, code cells can be executed in any order. Even with proper version control and dependency management in notebooks, the non-sequential running of code cells can create hidden states that hinder the reproducibility of notebooks [26]. For example, executing a later cell before an earlier one in a Jupyter notebook can lead to using outdated variables. To achieve reproducibility, one best practice we shall follow is ensuring the notebook has a linear execution order. Six

articles [8, 14, 19, 20, 26, 29] suggest re-running notebooks from top to bottom before storing or sharing. This approach restores the execution counter and reduces issues caused by hidden states.

In this practice, ensuring notebooks have linear execution order could mitigate risks of hidden state when sharing with others, so the conclusion of this practice is “*notebook execution is reproducible*”. The evidence, the support where the conclusion is based, could be “*notebook file is ready*”, indicating that the notebook file is ready to be verified if it follows a linear execution order. However, considering the definition of ready [10], from a test perspective, we can only automatically check the notebook file’s existence at this point. So, the evidence is “*notebook file exists*”. More research is needed to check readiness. This will be further discussed in the Future work part of this report. The strategy is a method used to establish reasoning between evidence and conclusion. Once the notebook file exists, we can check its linear execution order. The strategy could be “*verify notebook has linear execution order*”. The diagram is represented in Figure 3.1.

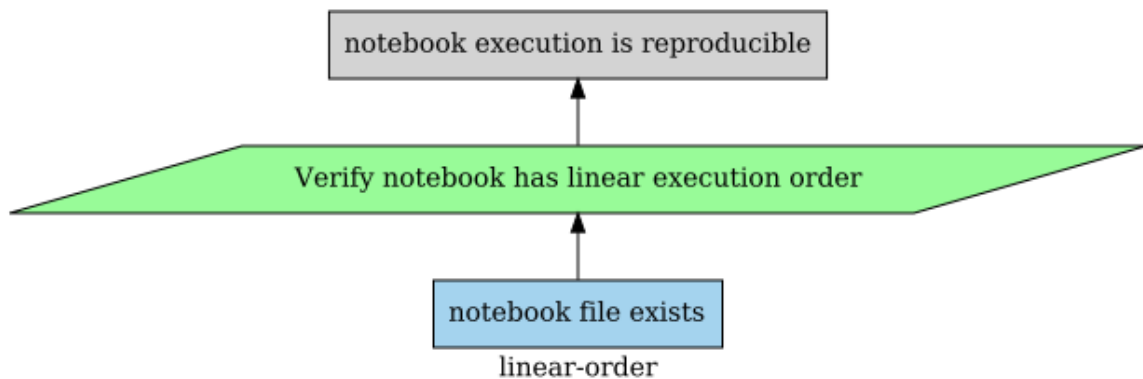


Figure 3.1: Justification Diagram for linear execution order

Thus, according to the diagram, we can build the first quality test case by checking

the notebook file’s existence first and then verifying that the notebook has a linear execution order.

3.1.2 Beginning Imports

When working with notebooks, developers often need to import extra dependencies to perform complicated computations. Another best practice extracted from the paper [26] is “putting import statements at the beginning of a notebook”, ensuring reproducibility of the execution environment. It is also mentioned in the paper [20], where authors discovered that over 90% of their studied notebooks put imports at the beginning cell. This practice also aligns with the official Python PEP8 style guide [24]. The Justification Diagram can be seen in Figure 3.2. It has two benefits:

1. The dependencies being used are apparent at first glance.
2. When restarting the notebook server, all imports can be restored by a single re-run.

By applying this practice, ensure the execution environment of the notebook is reproducible. So, the second quality test case for notebooks starts by checking the existence of the notebook file and then verifying the position of all import statements within it. If the notebook passes these two checks, we can conclude that “*notebook execution environment is reproducible*”.

3.1.3 Pinned Dependencies

When working on projects shared or used across different environments, we often install the required packages first. However, the lack of versions may cause system

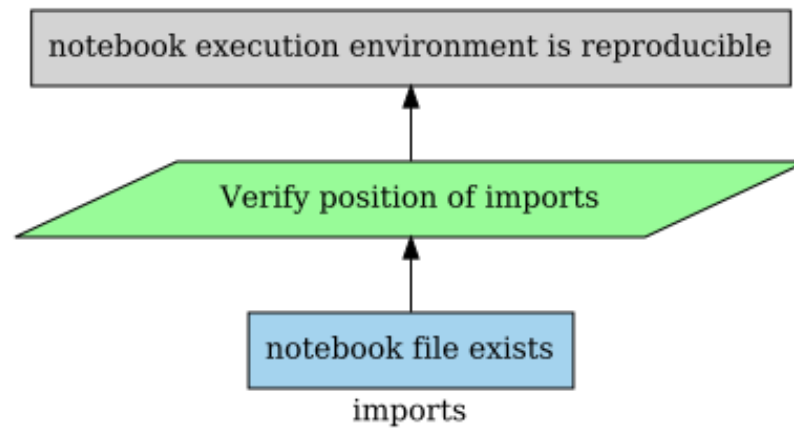


Figure 3.2: Justification Diagram for beginning imports

incompatibility issues [20]. For example, a function fails because it requires a feature not present in the older version. Several articles [19, 20, 26, 29] suggested that one practice contributing to notebooks’ reproducibility is maintaining a `requirements.txt` file and declaring each dependency’s version number so that other researchers can use it to install identical versions of every module and library as you did, ensuring notebook execution environment are reproducible. For this practice, The focus is not on the notebook but on the `requirements.txt` file. Figure 3.3 shows the evidence of this practice is “*requirement file exists*”, verifying the `requirements.txt` exists. Again, as discussed in section 3.1.1, the ideal evidence is “*requirement file is ready*”, which requires more research work in future. Then, we can apply a strategy to check that each dependency version is specified correctly, “*verify dependencies version are correctly pinned*”. The conclusion would be the same as the last practice: “*notebook execution environment is reproducible*”

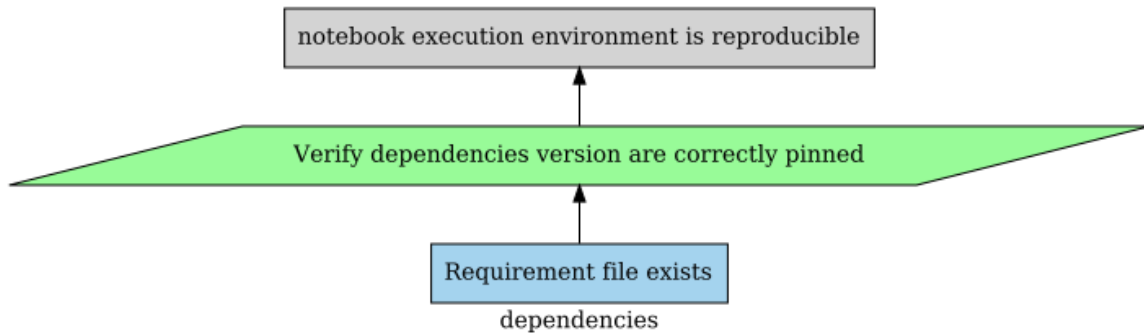


Figure 3.3: Justification Diagram for pinned dependencies

3.1.4 Virtual Environment

To make the execution environment even more straightforward to reproduce, many papers and blogs [26, 27, 29, 32] recommend running notebooks in self-contained environments, such as virtual environments: `conda`¹, `pipenv`², `virtualenv`³ or `docker` containers⁴. Because using a virtual environment ensures dependency consistency across different systems, preventing issues related to “it works on my machine”. In this practice, the conclusion stays the same, “*notebook execution environment is reproducible*”. The evidence is now the configuration files used to build virtual environments or containers, “*virtual environment configuration files exist*”. To support the conclusion, we need to verify that those configuration files are valid, “*verify configuration files are valid*”, and that the user used the virtual environment or container to run their notebooks, “*verify usage of virtual env*”, as shown in Figure 3.4.

Thus, this practice contains three checks: first, check the existence of the configuration files. A list of configuration files needed for each environment is outlined in

¹<https://conda.io/projects/conda/en/latest/user-guide/concepts/environments.html>

²<https://pipenv.pypa.io/en/latest/>

³<https://virtualenv.pypa.io/en/latest/>

⁴<https://www.docker.com/resources/what-container/>

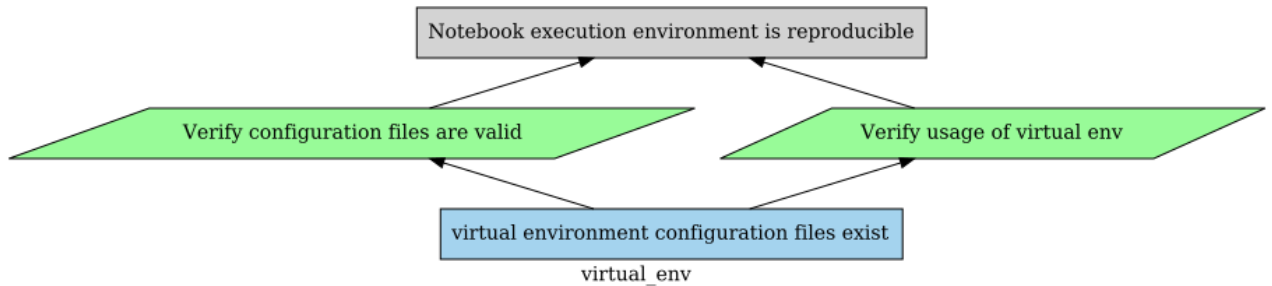


Figure 3.4: Justification Diagram for virtual environment

Table 3.1. Second, verify the validity of those configuration files. Third, verify that the notebook is indeed running in those environments.

Environment	File Name
pipenv	Pipfile and Pipfile.lock
conda	environment.yml
virtualenv	requirements.txt
docker	Dockerfile

Table 3.1: Configuration files for self-contained environments

3.2 High-Quality Code

Another practice we shall apply to notebooks is ensuring code quality. High-quality code is critical when working in teams. It helps team members understand others' work easier and move forward faster. Four best practices are related to this theme: meaningful name, pep8 standard, relative path and notebook testing, which are explained below.

3.2.1 Meaningful Name

A lousy name for notebooks is a nightmare when working in a team, causing difficulty finding the right notebook. Thus, a proper naming strategy is necessary. It could help everyone navigate file systems or shared repositories, find the right notebook more quickly and save more time. Paper [20, 26] recommends giving a meaningful name for notebooks. Paper [20] concludes explicitly three keys that build up a good notebook name from many notebook samples:

1. Always customize the notebook name.
2. Avoid using non-portable characters in the notebook name.
3. Avoid using excessively long names.

The “*notebook file exists*” is evidence of this practice because we can only verify its name if the notebook file exists. The conclusion is that “*notebook is shareable and searchable*” based on the benefits of a good naming strategy. The strategy should be “*verify that the notebook has a proper name*” based on the three points above. The Justification Diagram can be viewed in Figure 3.5.

Thus, another quality test we have is first to check the existence of the notebook file, then verify the notebook file has a valid name based on the above three keys.

3.2.2 PEP8 Standard

In software development, sticking to coding standards is a common way to ensure that the code produced by one developer will be easy to read, understand, and maintain by other developers [26]. Another best practice is sticking to widely accepted coding standards when developing computational notebooks, specifically the PEP8 style

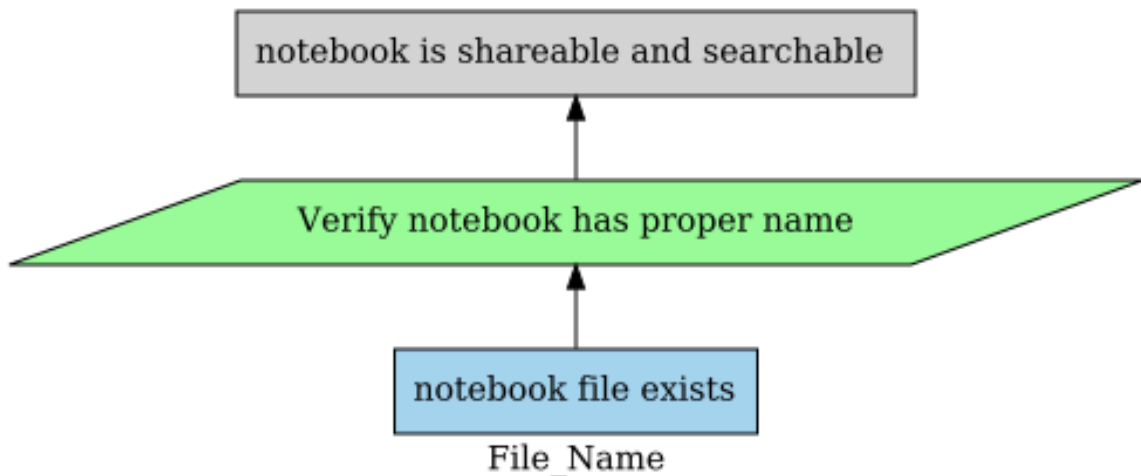


Figure 3.5: Justification Diagram for meaningful name

guide. This practice helps you seamlessly navigate between various of your projects, and there is no need to spend time deciding what style to use here and there. This practice shares a similar diagram with the above one, shown in Figure 3.5. Evidence: “*notebook file exists*”, strategy: “*check PEP8 coding standard*”, conclusion: “*notebook code quality is fair*”.

The quality test for this practice is also simple: first, verify the existence of the notebook file. Then, verify that the notebook follows the PEP8 style.

3.2.3 Relative Path

When processing large data sets, storing the data in separate files and reading it in the notebook is common. However, reading the data file might fail in different environments. According to a study, about 12.59% of their examined notebooks have errors like “**FileNotFound**” or “**IOError**” [20]. One cause of errors is when users use absolute paths to access data files, and the home directory varies in different machines, so someone else might have a different absolute path. Therefore, using

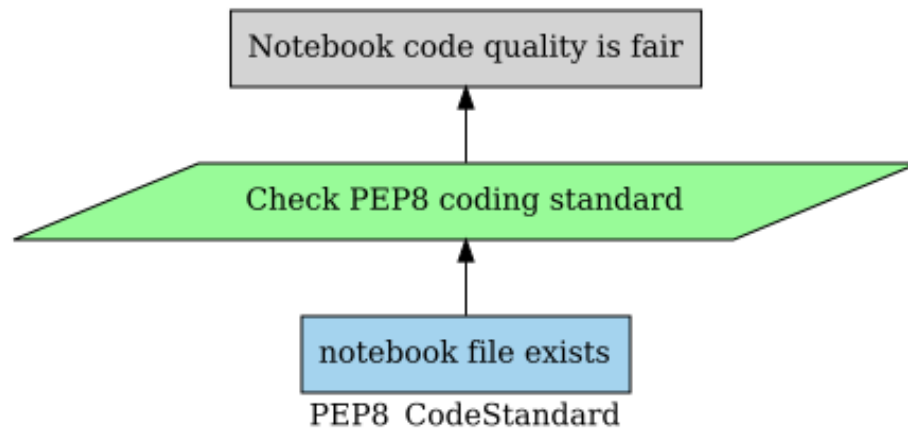


Figure 3.6: Justification Diagram for pep8 standard

relative paths in notebooks for data access is essential. Regular programs typically have two types of paths:

1. Static path, path generated at compile time.
2. Dynamic path, path generated at run time.

Given a notebook file, suppose the dynamic and static paths generated are relative paths. In that case, we are sure that no absolute path exists in the notebook, reducing the possibility of error occurrence.

Regarding the diagram, “*notebook file exists*” is the evidence, and the conclusion indicates the notebook is shareable, “*notebook is shareable*”. We shall have two strategies for this: one is verifying that statically generated paths in the notebook are relative paths, “*verify all static paths are relative paths*”, and the other is confirming that dynamically generated paths are relative paths, “*verify all dynamic paths are relative paths*”. The Justification Diagram can be seen in Figure 3.7.

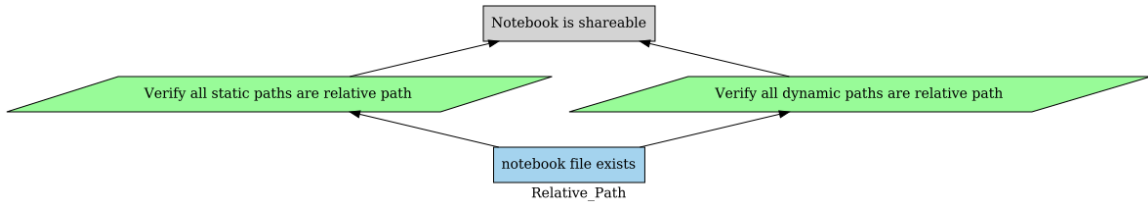


Figure 3.7: Justification Diagram for relative path

3.2.4 Notebook Testing

Testing is a standard practice in modern software development, aiming to ensure programs work as the engineers expect. Research papers and blogs [8, 19, 20, 26] acknowledge that testing is important, but none specify which type of tests is necessary. There are many types of testing methods nowadays: unit tests, mock tests, integration tests, regression tests and so on. In this practice, we will focus on checking the existence of unit tests to improve the quality of the notebooks because papers [20, 26] found the most imported testing modules in notebooks are unit test modules. Other types of tests shall be covered in future studies.

Performing unit tests in notebooks has two ways:

1. Directly write and run unit tests inside the notebook.
2. Write the tests in a separate Python script and execute it.

Therefore, two independent Justification Diagrams are needed for each execution method. The first case assumes the user writes and runs unit test cases inside the notebook directly. As pictured in Figure 3.8, we need “*notebook file exists*” as the evidence. Then, if the notebook contains an imported unit test module and a defined function, we are sure that the notebook is ready for unit tests. Because only if the test module is imported into the notebook the notebook can trigger unit tests. Only

if functions are defined inside the notebook are the unit tests needed. So, those two checks become the two strategies: “*verify the existence of common unit test import modules*” and “*verify notebook has functions defined*”, leading to the sub-conclusion: “*notebook is ready for unit tests*”.

Once we know that the notebook needs unit tests, we can have two approaches to check the test results statically and dynamically: “*verify unit test coverage*” and “*verify unit test correctness*”. If the test coverage is acceptable, around 80% [22], and all tests pass, we can conclude “*notebook quality is tested and proven*”.

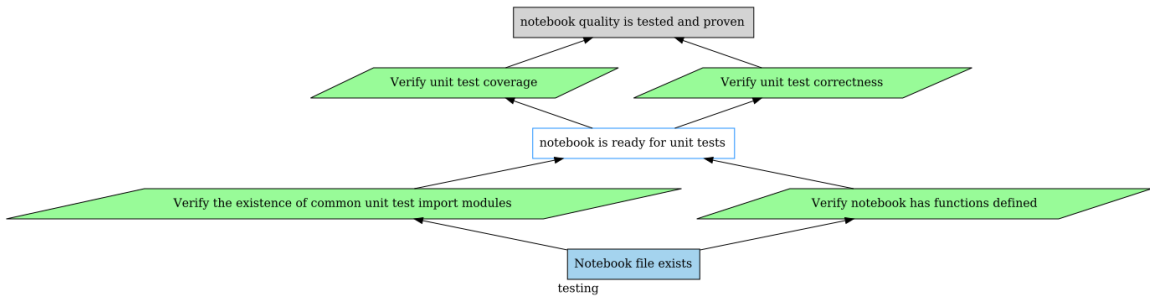


Figure 3.8: Justification Diagram for notebook-testing

The other case assumes the user would run a separate Python script to test the notebook, so we focus on the reasoning between the test script and the notebook quality. Instead of checking the notebook file’s existence, “*Python test script exists*” should be the evidence that supports a conclusion. Once the Python test script exists, we need to verify its test “*coverage*” and “*correctness*” to ensure that the test script is good. Additionally, we need to verify the user is indeed using this script to test the notebook, “*verify usage of test scripts*”. With the success of all three strategies, we conclude “*notebook quality is tested and proven*”. A detailed diagram is illustrated below in Figure 3.9.

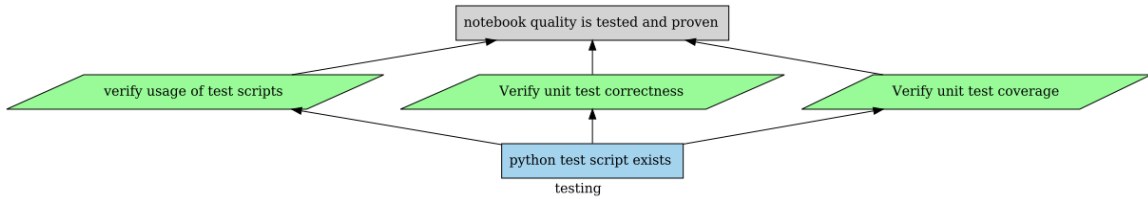


Figure 3.9: Justification Diagram for script-testing

3.3 Literate Programming Paradigm

Computational notebooks have gained popularity among data scientists mainly because they facilitate documentation alongside code, primarily inspired by literate programming [26]. Two best practices in this category are code documents and markdown headings. Both of them contribute to the notebooks’ readability and understandability.

3.3.1 Code Document

A well-written notebook should read like a book, detailing each computation step and its reasoning. This method of documenting directly within the code makes it easy for the author to track and update content. Other researchers or even the stakeholders can also examine notebook analysis easily. Several papers and blogs [19, 20, 26, 29, 30] highlight the importance of documentation in the notebook. Therefore, this best practice recommends documenting notebooks with markdown cells.

As concluded in paper [26], the documentation quality is impossible to verify within the notebook. However, we can check a few parts to support that a notebook is well documented, such as increasing the number of markdown cells used and checking whether the notebook begins and ends with a markdown cell. Additionally, consecutive code comments are recommended to transfer into markdown cells [26, 29].

In Figure 3.10, the diagram shows the quality test steps of this practice. First, the evidence: “*notebook file exists*”. Only if the notebook file exists can we verify the usage of markdown cells. The strategy should validate that the notebook has enough markdown cells and concise code comments, “*verify notebook has concise code comments and enough MD cells*”. If the notebook conforms to the check, we can conclude “*notebook is well-documented*”.

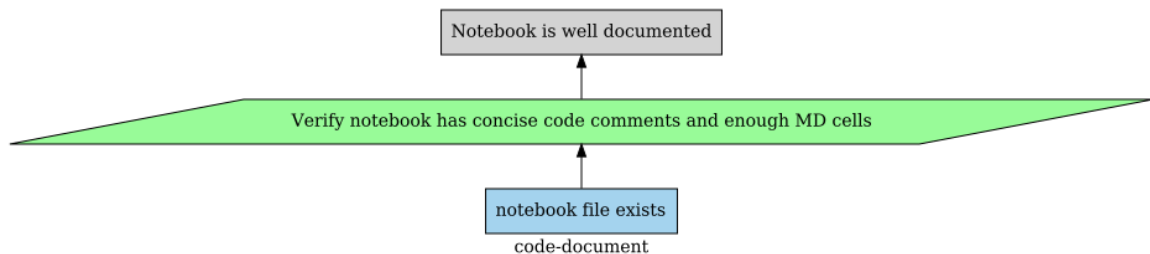


Figure 3.10: Justification Diagram for script-testing

3.3.2 Markdown Headings

This practice recommends the usage of markdown headers to organize the notebook, aiming for better readability and understandability. Papers [20, 26, 29] share the idea of leveraging markdown headers to give the notebook a good hierarchy structure and easy navigation. Specifically, paper [26] recommends using markdown headers from the beginning of the notebook to serve as a notebook introduction.

As visualized in Figure 3.11, a notebook file is needed, so “*notebook file exists*” is the evidence. The conclusion is that the notebook has a good structure, “*notebook has clear structure*”. For strategy, we can verify that markdown headers are adequately used in the notebook, “*verify MD headings used properly*”. Thus, this quality test involves two check steps.

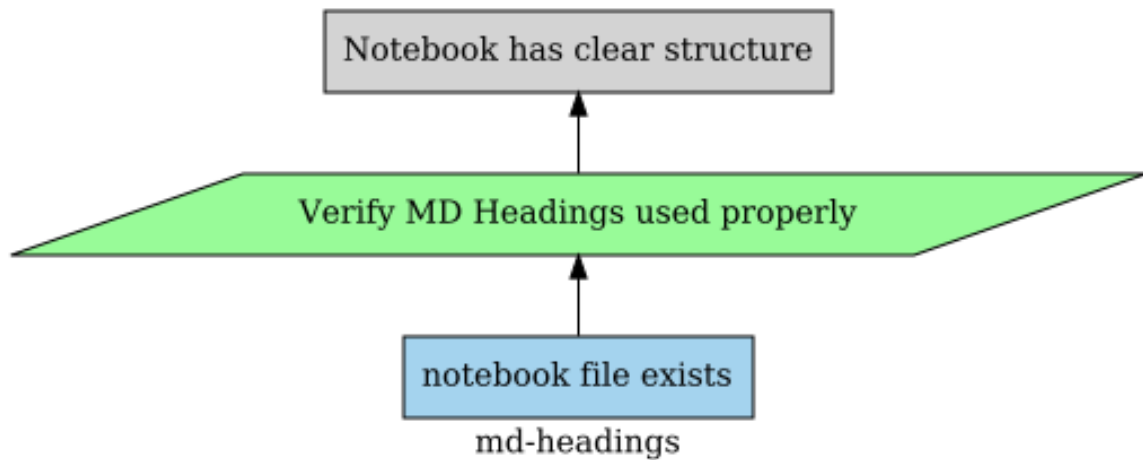


Figure 3.11: Justification Diagram for markdown headings

3.4 Clean and Concise

The notebook loses its advantages of literate programming if messy or overly complicated [26]. A common issue with notebooks is that they can easily become untidy, leading to difficulties in readability and understandability. This category emphasizes the importance of a concise and clean notebook. Each best practice in this category focuses on the quality of code cells.

3.4.1 Conciseness

Paper [26] suggests that developers should focus on creating concise, understandable cells, and each cell should perform one meaningful analysis step. Paper [29] suggests breaking long notebooks and cells into smaller ones and limiting the number of cells within each notebook. This practice advocates for writing concise notebooks and cells. According to paper [26], there are two properties we shall pay attention to ensure conciseness of the notebook:

1. the number of code cells.
2. the length of each code cell.

Thus, we can have the notebook file as the evidence, “*notebook file exists*”, and then the strategy is to check the notebook size and cell length, “*verify notebook cell length and overall size*”. If the notebook’s overall size and cell lengths are proper, we can conclude “*notebook is concise*”. To build the quality test, we first need to check the notebook occurrence. Then, check whether it has the proper number of code cells and whether each cell has the appropriate length.

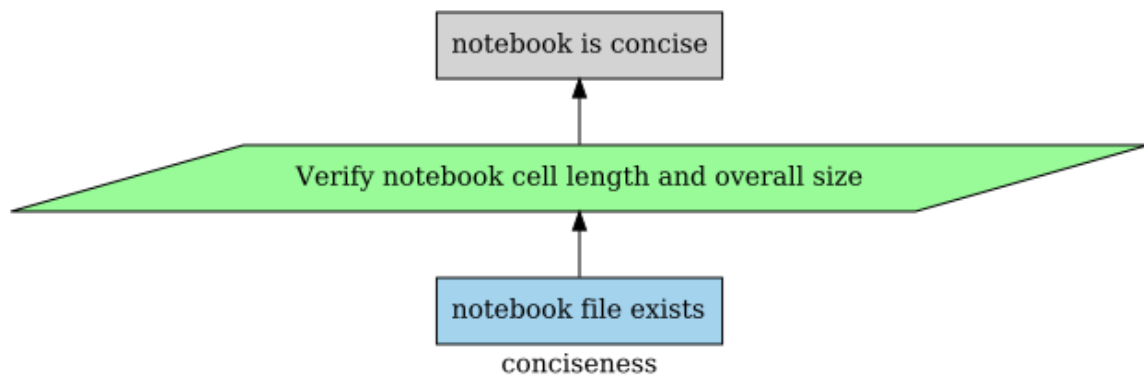


Figure 3.12: Justification Diagram for conciseness

3.4.2 Tidiness

The second best practice in this category focuses on the code cell’s cleanliness. Several papers [20, 26, 29] mentioned cleaning notebooks is a significant best practice. Based on their findings, there are three keys that we should pay attention to to keep the notebook clean:

1. Ensure no unexecuted code cells are in notebooks.

2. Ensure no empty code cells in notebooks.
3. Ensure no Python syntax errors in notebooks.

Therefore, we should verify the above three points in this practice and conclude “*notebook is clean*”. Clearly shown in Figure 3.13, “*notebook file exists*” is the evidence, and the strategy is to verify that there are no empty and unexecuted code cells and no syntax errors in the notebook, “*verify notebook for the absence of dead cells and code*”. The quality test involves a notebook file check and a check of the above three keys.

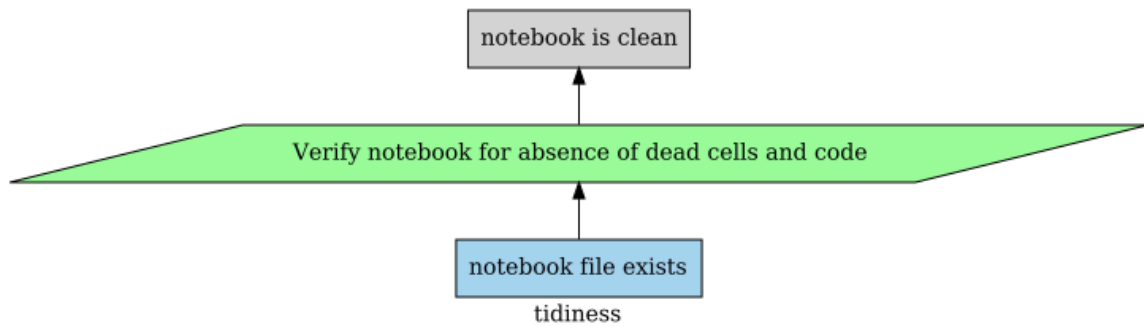


Figure 3.13: Justification Diagram for tidiness

3.5 Conclusion

In Chapter 3, we have explored 12 key best practices for notebook development, which we categorized into 4 groups. For each best practice, we have provided clear explanations and transformed them into detailed quality check steps using Justification Diagrams. These steps aim to enhance the quality of notebooks, with the JD providing solid rationales for each practice.

From this analysis, we can conclude that the Justification Diagram Language can be used to model each best practice into a standalone, comprehensive quality test.

Chapter 4

Quality Check Implementations

As discussed in chapter 3, we have converted 12 notebooks' best practices into 12 individual quality test cases using the Justification Diagram Language. The next important step is implementing those test cases so that users can validate their notebooks.

This chapter chooses practice **notebook** testing (section 3.2.4) under the **High-Quality** Code category as a case study and exhibits its detailed implementation using GitHub workflow in section 4.1, answering the question: “**How can Justification Diagrams be transferred into quality checks in CI/CD pipelines?**”. Then, we will continue the case study and discuss a scenario where one diagram can have separated implementations in section 4.2, demonstrating Justification Diagrams' flexibility and reusability.

4.1 GitHub Workflow Implementations

Recall from the notebook testing practice explained in section 3.2.4 that there are two ways to execute unit tests in notebooks. One is directly running unit tests within the notebook, and the other is indirectly running test scripts, represented respectively in Figures 3.8 and 3.9. The converted quality test steps for both cases are also explained in section 3.2.4. To implement the one running tests within the notebook, we can map those steps into a GitHub workflow and check if each step is valid. A mapping from Figure 3.8 to the actual quality test steps can be found in Figure 4.1. Every evidence (**Su1**) and strategy (**St1**, **St2**, **St3**, **t4**) is transformed into a check and can be performed in order inside a GitHub workflow. Users can trigger the whole process manually or based on events. If any check fails in the middle of the GitHub workflow, it will reject the CI/CD process and state the notebooks' problems.

```
- Su1 "notebook file exists"
  - check notebook file exists
- St1 "Verify notebook has functions defined"
  - run pynblint command to check that the notebook contains functions
- St2 "Verify existence of unit test import modules"
  - run python script to check the existence of unit test modules being imported in notebooks
- St3 is "Verify unit test correctness"
  - convert notebook to Python script using nbconvert
  - run coverage command to check the unit tests result
- St4 is "Verify unit test coverage"
  - run coverage command to check the code coverage of unit test is acceptable.
```

Figure 4.1: Mappings of Justification Diagrams to quality test steps for notebook testing

Figure 4.2 is the YAML file of this quality test, representing a pipeline that checks the usage of Pytest within notebooks. Recall from the background section that the YAML file is used to define the GitHub workflow pipeline. The keyword on line 4, **'workflow_dispatch'**, determines that the workflow can be triggered manually.

Lines 5 to 10 represent the user inputs. Starting from line 12 to the end are the implementations of test steps.

```
1 name: Test pytest modules in notebooks
2
3 on:
4 workflow_dispatch:
5   inputs:
6     notebook_path:
7       description: "Path to the targeting jupyter notebook
8         "
9       required: true
10      default: "./high-quality/notebook-testing/notebook/
11              pytest/examples/pytest_notebook1.ipynb"
12 jobs:
13   check-practice-pytest:
14     runs-on: ubuntu-latest
15
16     steps:
17     - uses: actions/checkout@v4
18
19     - name: Set up Python
20       uses: actions/setup-python@v4
21       with:
22         python-version: 3.8
23
24     - name: Install tools needed
25       run: |
26         python -m pip install --upgrade pip
27         pip install nbconvert coverage pytest pynblint
28
29     - name: Verify notebook exists
30       run: |
31         ./helper-scripts/check_file_exist.sh ${${
32           github.event.inputs.notebook_path }}
33
34     - name: Verify notebook has functions defined
35       run: |
36         pynblint --include '['"']' --output lint-results.
37         json --quiet ${${ github.event.inputs.notebook_path }}
38         ./helper-scripts/check_functions_defined.sh lint-
39         results.json
```



```

36
37 - name: Check existence of pytest used in notebook
38   run: |
39     python3 ./helper-scripts/check_test_modules.py ${
40       github.event.inputs.notebook_path } pytest
41
42 - name: Verify unit test correctness
43   run: |
44     jupyter nbconvert --to script ${
45       github.event.inputs.notebook_path }
46     echo "NOTEBOOK_SCRIPT=$(echo '${
47       github.event.inputs.notebook_path }'| sed 's/\.ipynb$
48       /\.py/')

```

Figure 4.2: YAML file of GitHub workflow testing the usage of Pytest within notebooks

This quality test contains five steps, each checking one notebook condition. A list of scripts and tools used in those five steps is shown in Table 4.1. The first step, **Su1**, is to check if the notebook file exists. We ask users to provide a path of the targeting notebook through GitHub environment inputs and then run a bash script ‘`check_file_exists.sh`’ to check its existence. **St1** verifies whether the notebook has functions defined by utilizing the Pynblint linter to count the number of functions and output the result into a JSON file. Then, check its result with a bash script ‘`check_functions_defined.sh`’. The third step, **St2**, verifies whether the Pytest library is imported statically inside the notebook. We built a Python script ‘`check_test_module.py`’ that employs the Abstract Syntax Tree (AST¹) module, a

¹<https://docs.python.org/3/library/ast.html>

feature that can help programmatically process Python code to check Pytest occurrence. Once the notebook passes **Su1**, **St1** and **St2**, we can confirm the notebook is ready for unit testing. For the fourth and fifth steps, we first convert the notebook file into a Python script by **nbconvert**, a tool that converts notebook files to other formats. Then, we employ coverage, a tool for measuring the code coverage of Python programs, to verify the test results as well as the test coverage of the converted Python script. If the notebook passes all unit tests and the test coverage is over a certain threshold (80% by default), we can conclude the notebook is tested and proven.

Type	Name	Description
Bash scripts	check_file_exist.sh	Check the existence of a file
	check_functions_defined.sh	Check the existence of functions within a given notebook
Python script	check_test_module.py	Check the existence of a given test module within a given notebook
Python tool	pynblint	Jupyter notebooks linter
	coverage	Measuring tool for Python code coverage
	nbconvert	A tool that converts Jupyter notebooks to other formats

Table 4.1: Scripts and tools used in notebook testing

It's vital to note that all these scripts and tools used in this quality test are reusable. They can be used in other quality tests. Moreover, these implementations are not exclusive to GitHub Workflow. It can be implemented freely in other CI/CD pipelines such as Jenkins, CircleCI, Azure Pipeline, etc. We chose GitHub workflow

for the case study due to its ease of use and broad accessibility.

Many unit-testing libraries, such as Unittest², Doctest³, Pytest⁴, and others, are available for Python notebooks. We focus on testing the tool Pytest in this case study. For explanations of other unit testing tools, please see section 4.2. In section 4.1, we briefly summarize the implementations of each step in this quality test. For a more detailed description of the scripts and tools used in the process, please refer to Appendix B or the repository [16].

4.2 Implementation Separation

Following the case study in the above section, many unit testing tools are available, such as Unittest, Doctest, Pytest and so on. In real-world situations, users can choose any testing tool based on personal preference. Consequently, there is a need for a flexible approach that allows for the adaption of existing implementations to suit different testing tools. Luckily, another benefit that the Justification Diagram Language offers is its reusability. The constructed diagram is only an abstract presentation of test steps, while the specific implementations of each step can be tailored as needed. That means no matter which testing tool the user prefers, the core structure of JD (Figure 3.8) and the quality test steps (Figure 4.1) remain the same. For example, we can modify the implementations of Pytest (Figure 4.2) to check the existence of Unittest, as shown in Figure 4.3.

²<https://docs.python.org/3/library/unittest.html>

³<https://docs.python.org/3/library/doctest.html>

⁴<https://docs.pytest.org/en/8.0.x/>

```
1 - name: Verify notebook exists
2   run: |
3     ./helper-scripts/check_file_exist.sh ${
4       github.event.inputs.notebook_path }
5 - name: Verify notebook has functions defined
6   run: |
7     pynblint --include '[' --output lint-results.json
8     --quiet ${
9       github.event.inputs.notebook_path }
10    ./helper-scripts/check_functions_defined.sh lint-
11      results.json
12
13 - name: Check existence of pytest used in notebook
14   run: |
15     python3 ./helper-scripts/check_test_modules.py ${
16       github.event.inputs.notebook_path } unittest
17
18
19 - name: Verify unit test correctness
20   run: |
21     jupyter nbconvert --to script ${
22       github.event.inputs.notebook_path }
23     echo "NOTEBOOK_SCRIPT=$(echo '${
24       github.event.inputs.notebook_path }' sed 's/\.ipynb$/\.py
25       /)'" >> $GITHUB_ENV
26     coverage run -m unittest discover -s $(dirname
27       "$NOTEBOOK_SCRIPT") -p $(basename "$NOTEBOOK_SCRIPT")
28
29
30 - name: Verify unit test coverage
31   run: |
32     coverage report --fail-under=80
```

Figure 4.3: Implementations for Unittest module

The only difference between Unittest and Pytest implementations is the parameters passed to the ‘check_test_modules.py’ script and the ‘coverage’ tool in the third and fourth steps, highlighted in red font.

Thus, one Justification Diagram can have many implementations. Users can adjust the implementations based on their needs. The same theory applies to best

practice: virtual environment, where multiple environments (Pipenv, Conda, Virtualenv, Docker) share the same Justification Diagram. Please refer to the GitHub repository [16] to view its implementations.

4.3 Conclusion

This case study demonstrates the practicability and reusability of Justification Diagrams. It explains how we can transfer JD into quality tests within GitHub workflows and highlights its reusability through an example of having multiple implementations for one diagram. With this analysis, we are confident in asserting that the Justification Diagram Language is a practical and usable solution for users to apply best practices to their notebooks, thus enhancing overall notebooks' quality and fairness.

Chapter 5

Operational Justification Diagram Language

We have demonstrated in the previous chapter that Justification Diagrams are effective in developing quality tests to enhance the overall quality of notebooks. However, the current approach still presents several challenges and inconveniences when applying this technique in real-world applications. This chapter aims to answer the question: **“How can we improve the translation between Justification Diagrams and quality tests?”**. We will first discuss the challenges encountered using the Justification Diagram Language, as detailed in section 5.1. Next, we will introduce a solution - the Operational Justification Diagram Language - and explain how it is designed by explaining its operations in section 5.2 and syntax in section 5.3. Finally, section 5.4 shows a real example of using this new language to create quality tests for notebooks, showcasing its capacity to crack these challenges.

5.1 Introduction

Even though all proposed best practices are represented by Justification Diagrams in Chapter 3 and integrated into the GitHub workflow, there’s still a big learning curve for users to grasp these concepts and be able to create new quality tests using JD themselves. Besides, even for the existing implementations, it takes time for users to understand all the scripts and tools involved so that they can adapt the implementations to suit their specific team needs. For example, suppose a data science team prefers using another tool like Nose¹ instead of Pytest for unit testing their notebooks. In that case, they must fully understand how existing Pytest scripts work and adjust the parameters accordingly.

Moreover, lots of manual work is needed to use those quality tests. In the current setup, each quality test is defined in a ‘.jd’ file and implemented by a YAML file, which demands considerable manual effort from users to maintain these files, thereby adding the possibility of human errors.

To alleviate those difficulties, we need to create an Operational Justification Diagram Language that aims to enrich JD with operational commands. This new language should simplify the conversion of JD into practical GitHub workflow implementations, allowing users to write JD along with the implementation without needing to know all the details of helper scripts and tools, saving time for users. More explanations are provided in the following section.

¹<https://nose.readthedocs.io/en/latest/>

5.2 Eliciting Reusable Operations

To address the challenges mentioned, an Operational Justification Diagram Language is needed to reduce the complexity of converting JD into real implementations in the GitHub workflow. To create such a language, we first need to study the structure of each quality test workflow, extract all the scripts, commands and environment settings from them and identify commonalities. Then, we encapsulate shared ones into simple operations that can be reused easily, ensuring reusability and simplicity. Since our study focuses only on the quality tests for Jupyter Notebooks, every test shares the same workflow structure with only different test steps so that we can look into shared scripts and tools directly. For how to design operational language on different platforms, please refer to Deesha Patel’s work [18]. This section will continue the quality test explained in section 4.1, describing how to define operations of this new language based on the YAML file of that quality test.

Recall the complex GitHub workflow implementation in Figure 4.2, which involves many helper scripts, commands and the environment setup. You can find a thorough description of them in Appendix B.

The script ‘`check_file_exist.sh`’ requires a file path as input to verify whether the file is present. It is a commonly shared script by every quality test. Instead of manually executing this in the GitHub workflow like writing ‘`./helper-scripts/check_file_exist.sh ./notebook.ipynb`’ to check if ‘`notebook.ipynb`’ exists in the current directory. We can simplify this with our language by defining an operation: ‘`file_exist()`’. This function performs the same check when calling ‘`file_exist("./notebook.ipynb")`’.

Similarly, ‘`check_test_modules.py`’ is a script to confirm if a particular test

module is imported into a given notebook file. We can streamline this by creating an operation: `import_exist()`, which requires two inputs. For example, `import_exist("./notebook.ipynb", "pytest")` checks whether Pytest is imported in the `notebook.ipynb`.

Pynblint linter is also a frequently used tool in our quality tests. Each quality test applies different lint rules, and the results are typically saved in a JSON file for further processing [Appendix B - Pynblint]. We can design an operation: `pynblint_check()`, that accepts three parameters: the notebook file path, selected lint rules, and the desired output file name. This operation allows users to easily apply different Pynblint rules on the notebook file and customize the output file name.

The `jupyter notebook -to script` command, followed by a notebook file path, is a standard method to convert a notebook to a Python script. This command is also a candidate for encapsulation into an operation: `convert_to_script()`.

Not every script and command is required for all quality tests, as some tests have unique checks that demand specific ones. For example, the Coverage tool uses different commands for various unit test modules: `coverage run -m pytest` for Pytest, `coverage run -m unittest` for unittest. Therefore, we need a mechanism to allow users to execute customized commands, so we introduce the `run[]` operation. For example, to obtain pytest results of `notebook.ipynb` with Coverage, users can write: `run["coverage", "run", "-m", "pytest", "notebook.ipynb"]`.

Setting up the workflow environment for quality tests usually involves user inputs, the base operating system, and additional configurations. To add user inputs, we designed an operation: `add_input()` takes three parameters: name, description and a default value. Users can write `add_input("notebook_path", "the`

`path to your notebook”, “./notebook.ipynb”)` to create a user input variable: `notebook_path`, storing the value `./notebook.ipynb`. This variable can be accessed by writing `${github.event.inputs.notebook_path}` in the language.

We also need two operations, `run_on()` and `add_step()`, where `run_on(“ubuntu-latest”)` specifies that the GitHub workflow should run on an Ubuntu machine. And `add_step(“actions/setup-python@v4”)[python-version= “3.8”]` configures the workflow to install Python 3.8.

Operation	Description
<code>file_exist()</code>	Check the existence of a file.
<code>import_exist()</code>	Check a provided module is statically imported in a notebook file.
<code>pynblint_check()</code>	Execute Pynblint linter on a notebook file with specific lint rules.
<code>convert_to_script()</code>	Convert a notebook file into a Python program.
<code>run[]</code>	Execute customized bash commands.
<code>add_input()</code>	Define user inputs.
<code>run_on()</code>	Set up base OS system for workflow.
<code>add_step()</code>	Set up environments for workflow.
<code>pynblint_check_results()</code>	Check the Pynblint results contain any recommendation.
<code>relative_path_check()</code>	Check all statically defined paths in a notebook file are relative path.
<code>multiline_comments_check()</code>	Check the existence of any consecutive code comments.

Table 5.1: Operations

We have encapsulated all the commonalities shown in Figure 4.2 into reusable operations. These operations simplify the procedures of writing GitHub workflows, ensuring consistency and ease of understanding for users. A list of all operations we

defined can be found in Table 5.1. It includes all the operations we mentioned above and additional operations used in other quality tests.

5.3 Syntax

To use the operations defined above, we also need to design our language’s syntax. The goal is to minimize the learning curve for users, allowing them to define quality test steps using the Justification Diagram Language while implementing the tests using this new language. This section introduces several keywords we define to complete this new language. Table 5.2 lists all the keywords and their explanations.

Name	Description
evidence	This keyword establishes a link to the corresponding ‘ evidence ’ in the ‘.jd’ file by following the evidence name, such as ‘ evidence Su1 ’. It also defines the check steps involved in this evidence.
strategy	This keyword establishes a link to the corresponding ‘ strategy ’ in the ‘.jd’ file by following the strategy name, such as ‘ strategy St1 ’. It also defines the check steps involved in this strategy.
expectation	This keyword is used to define the expected output of each test step.
is	This keyword assigns the implementation details to ‘ evidence ’ and ‘ strategy ’. It is also used to assign expected value to ‘ expectation ’.
load	This keyword is used to import the ‘.jd’ file for linking.
implementation	It names the implementation of the quality test.
deploy	This keyword indicates the environment that our tests are implemented on.
prologue	This keyword defines the environment settings.

Table 5.2: Keywords

5.4 Example

To build a quality test using the Operational Justification Diagram Language, we first need to load the corresponding ‘.jd’ file and set up the environment. An example of this initial step is shown in Figure 5.1.

```
1 load "../justification.jd"
2
3 deploy python on GitHubAction as "notebook-unittest.yaml"
4   {
5     prologue {
6       runs_on("ubuntu-latest")
7       add_input("notebook_path", "Path to the targeting
8         jupyter notebook", "./high-quality/notebook-testing/
9         notebook/unittest/examples/unittest_notebook1.ipynb"
10      )
11      add_step("actions/setup-python@v4") [python-version="
12        3.8"]
13      run ["python", "-m", "pip", "install", "--upgrade", "pip"]
14      run ["pip", "install", "nbconvert", "coverage", "pynblint"]
15    }
16  }
```

Figure 5.1: Implementation for environment setting

The first line loads the ‘justification.jd’ file, which corresponds to the diagram in Figure 3.8. The instruction ‘**deploy python on GitHubAction**’ indicates we are deploying this test on GitHub Action. The environment is configured using the ‘**prologue**’ keyword, which includes specifying Ubuntu as the operating system, defining the notebook path as a user input, and installing Python 3.8 along with other necessary tools.

Following this setup, we apply the defined operations to implement each piece of evidence and strategy. Figure 5.2 provides a detailed view of these implementations.

Each step ends with an ‘**expectation**’ keyword. ‘**Expectation is True**’ in Su1 means we expect the notebook file to exist. In the other four strategies (St1, St2, St3, St4), ‘**Expectation is EXIT_CODE.SUCCESS**’ implies that successful completion of each step is expected. The entire test will fail if any steps fail in the middle, similar to how the GitHub workflow fails the pipeline when encountering errors.

```
1 implementation unittest of notebook-unittest {
2
3   evidence Su1 is {
4     file_exist("${github.event.inputs.notebook_path}")
5     expectation is True
6   }
7
8   strategy St1 is {
9     pynblint_check("${github.event.inputs.notebook_path}"
10      ,[""],"lint-results.json")
11     run ["/helper-scripts/check_functions_defined.sh",
12      lint-results.json"]
13     expectation is EXIT_CODE.SUCCESS
14   }
15
16  strategy St2 is {
17    import_exist("${github.event.inputs.notebook_path}",
18      unittest")
19    expectation is EXIT_CODE.SUCCESS
20  }
21
22  strategy St3 is {
23    convert_to_script("${github.event.inputs.notebook_path
24      }")
25    add_env("NOTEBOOK_SCRIPT","echo '${github.event.
26      inputs.notebook_path }' sed 's/.ipynb$/.py/'")
27    run ["coverage", "run", "-m", "unittest", "discover", "-s",
28      $(dirname "${NOTEBOOK_SCRIPT}"), "-p", "$(basename "${
29      NOTEBOOK_SCRIPT}")]
30    expectation is EXIT_CODE.SUCCESS
31  }
32
33  strategy St4 is {
34    run ["coverage", "report", "--fail-under=80"]
35  }
36 }
```

```
28     expectation is EXIT_CODE.SUCCESS
29 }
30 }
```

Figure 5.2: Implementations for quality test: notebook-testing

As a result, by comparing our new implementations using the Operational Justification Diagram Language with the GitHub workflow implementation in Figure 4.2, we can see that the new language provides a much cleaner and easily understandable structure, demonstrating its effectiveness.

5.5 Conclusion

The introduction of this new language allows users to incorporate implementations alongside Justification Diagrams. It simplifies the process of setting up complex CI/CD pipelines, leading to a more organized code structure and user-friendly experience, as demonstrated in the example provided in section 5.4. Thus, we can conclude that this operational language can better assist users in utilizing JD, enhancing notebooks' quality efficiently.

Chapter 6

Conclusion and Future work

This chapter provides a summary of the project and discusses potential future works.

6.1 Summary

In conclusion, this report has explored the effectiveness of using the Justification Diagram Language to develop quality tests for Jupyter Notebooks within the CI/CD pipeline. This approach is beneficial for notebook users in maintaining robust coding practices and improving the overall quality of notebooks.

We began by identifying the need for quality checks due to the growing complexity and popularity, alongside the challenges in achieving reproducibility, especially in data science and AI fields. In Chapter 3, we demonstrated the application of the Justification Diagram Language in structuring and visualizing best practices for notebooks. We showed how these diagrams could be transformed into actionable quality test steps, providing researchers with a practical approach to designing proper quality tests. Chapter 4 featured a case study on implementing these quality tests in

GitHub workflow, confirming their practicality. We also emphasized a key advantage of the Justification Diagram Language: Its ability to support multiple implementations from a single diagram, enhancing reusability. Toward the end, we introduced an Operational Justification Diagram Language in Chapter 5. This language gives users a cleaner method to implement quality tests alongside the JD. This approach simplifies the testing process, making it more accessible and less time-consuming for users to apply quality tests to their notebooks.

6.2 Future work

The report has demonstrated the potential of the Justification Diagram Language in creating notebooks' quality tests. However, our research work in this field is incomplete, and we still need further studies in this field.

The Justification Diagrams in Chapter 3 visualize the best practice in a straightforward way. Going forward, we need to add more details to these diagrams to reflect the real-world notebook development process accurately. Specifically, as discussed in section 3.1.1, the only automated check that can be done is to check the file exists; more studies are needed to fulfill the evidence "*notebook file is ready*", "*requirement file is ready*", "*virtual environment configuration files are ready*" and "*python test script is ready*". Also, there are instances where our existing quality tests may not address specific situations. For example, according to the interview conducted in [26], several scientists believe that longer notebook filenames help understand the file content at first glance, which conflicts with the viewpoint of our suggested best practice of keeping filenames simple. Therefore, we need to create more JD and allow users to choose the best fit for their needs.

This report focuses on the best practices that enhance the quality of individual notebook files. There are broader practices related to improving the quality of working many notebooks in a repository, such as dataset reproducibility, which are worth further study.

In this project, we heavily rely on the Pynblint linter to implement the strategies of each best practice JD such as “*Verify notebook has linear execution order*”, “*Verify position of imports*”, and others. Additional experiments are necessary to ensure the accuracy of these implementations.

While our study proposed a practical solution for implementing best practices, we have yet to assess its usability among notebook users. Further works involving conducting experiment usability tests are needed for our tool.

Moreover, our current focus on notebook testing has been limited to tools such as Pytest, Unittest and Doctest. Future research should include a broader range of unit testing tools and other testing methods like mock tests, integration tests, etc.

Lastly, while our study has implemented test cases within the GitHub workflow, there is a need for additional research on incorporating these practices into other CI/CD platforms.

Appendix A

Notebook Best Practices

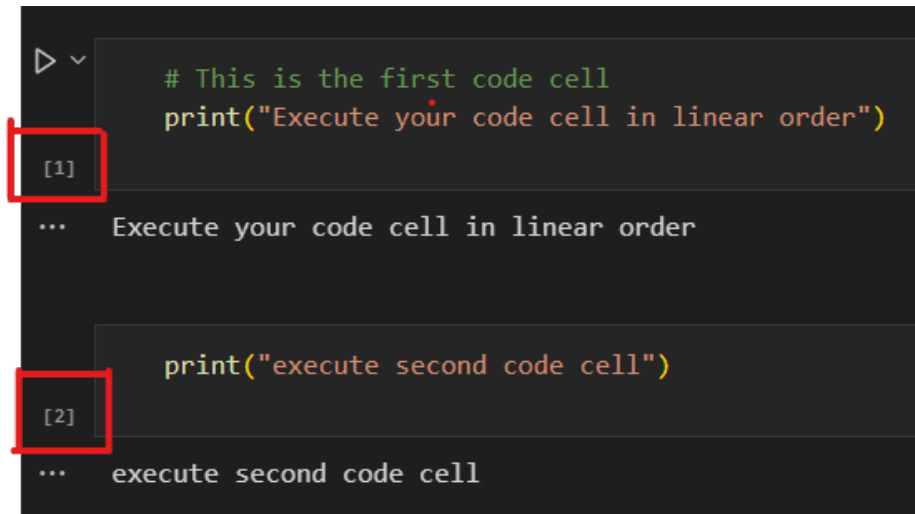
Section 3 briefly introduces 12 notebook best practices without explaining each in detail. In Appendix A, we will explain each best practice’s meaning by showing real examples.

A.1 Linear execution order

In notebooks, there is an execution counter representing the order in which code cells have been run, as shown in Figure A.1. Each time the user clicks on the run button, the execution counter increments by 1, indicating the order. A linear execution order means that the user triggers the code cell one by one from top to bottom.

A.2 Beginning Imports

Put all import statements in one code cell at the beginning of the notebook, just like Figure A.2



```
# This is the first code cell
print("Execute your code cell in linear order")

[1]

... Execute your code cell in linear order

print("execute second code cell")

[2]

... execute second code cell
```

Figure A.1: Example of linear execution order

A.3 Pinned Dependencies

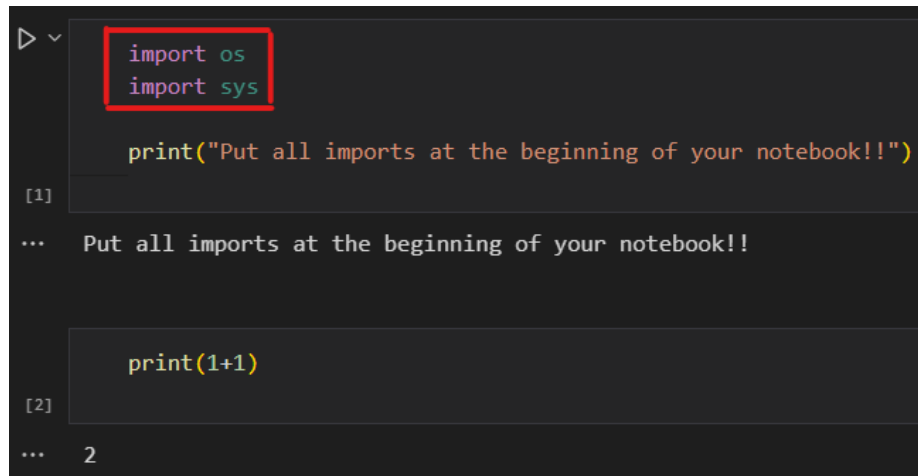
Figure A.3 shows an excellent example of `requirements.txt`, where the package `numpy` has a specific version number: `1.24.0`

A.4 Virtual Environment

Different configuration files are needed for each type of environment. To see a real example, please refer to the repository [16]

A.5 Meaningful Name

A Couple of bad notebook names are shown in Figure A.4. The first one contains non-portable characters. The second one's name is too long. And the third one uses the default notebook name.



```
import os
import sys

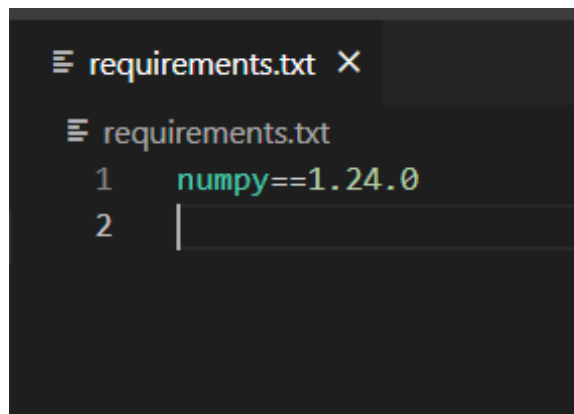
print("Put all imports at the beginning of your notebook!!")

[1]
... Put all imports at the beginning of your notebook!!

print(1+1)

[2]
... 2
```

Figure A.2: Example of beginning imports



```
requirements.txt X
requirements.txt
1 numpy==1.24.0
2 |
```

Figure A.3: Example of requirements.txt

A.6 PEP8 Standard

PEP8 standard contains many rules. One example that does not follow the PEP8 style is shown in Figure A.5, which has too many white spaces.

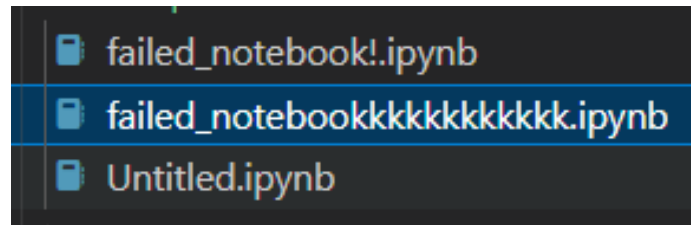


Figure A.4: Example of bad file name

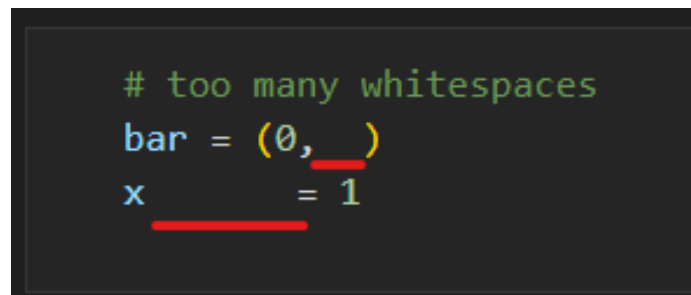


Figure A.5: Example of pep8 standard

A.7 Relative Path

A relative path typically begins with a dot (`.`), like `./data/sample.txt`. This format indicates a path that starts from the current folder (`./`) to the file `sample.txt` file located in the subfolder (`data/`). On the other hand, an absolute path usually starts with a forward slash (`/`), such as `/home/user/data/sample.txt`, indicating a complete path from the root folder of the file system to the `sample.txt` file

A.8 Notebook Testing

As explained in section 3.2.4, there are two ways to run unit tests in notebooks. The first runs directly within the notebook, like in Figure A.6a. We have `test_add()` and `test_subtract()` test functions defined in cell 2. The second runs a Python test

script, like in Figure A.6b. We have test functions written inside a Python script, ‘test_script2.py’, and trigger it by shell command in the notebook.

```
# Cell 1
def add(a, b):
    return a + b

def subtract(a, b):
    return a - b

[3]

# Cell 2
def test_add():
    assert add(1, 2) == 3

def test_subtract():
    assert subtract(4, 2) == 2

[4]

# Function to be tested
def add_numbers(a, b):
    return a + b

[1]

!pytest -v test_script2.py

[4]
```

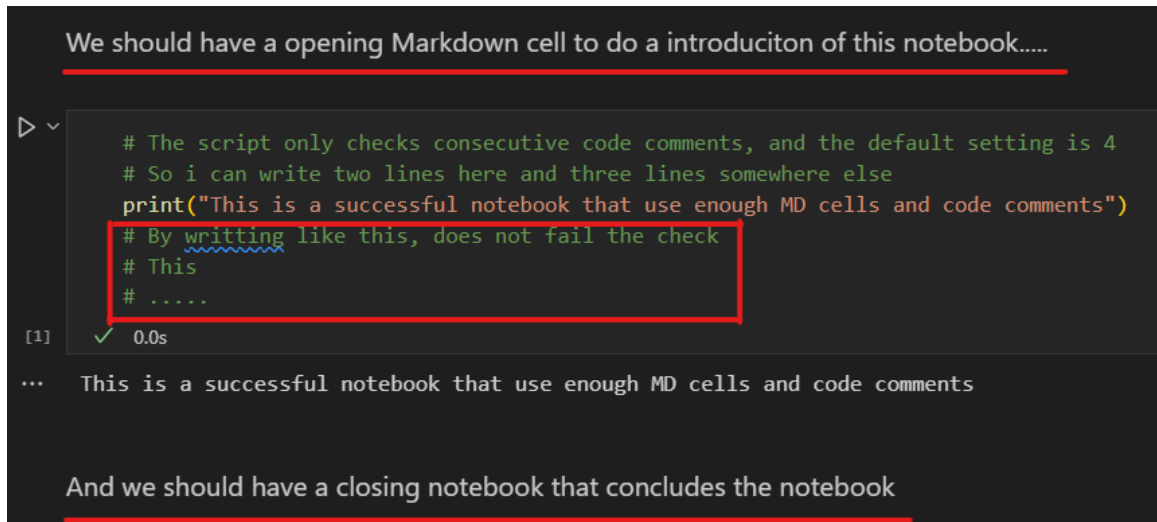
(a) pytest in notebook

(b) pytest in a Python script

Figure A.6: Example of notebook testing

A.9 Code document

As explained in section 3.3.1, it is recommended to have more markdown cells in the notebook. And avoid using more consecutive code comments. For the example in Figure A.7, we put two markdown cells at the start and end of the notebook, respectively. Also, we restrict a code cell to not having more than three consecutive code comments.



```
We should have a opening Markdown cell to do a introducion of this notebook.....  
  
# The script only checks consecutive code comments, and the default setting is 4  
# So i can write two lines here and three lines somewhere else  
print("This is a successful notebook that use enough MD cells and code comments")  
# By writting like this, does not fail the check  
# This  
# .....  
[1] ✓ 0.0s  
... This is a successful notebook that use enough MD cells and code comments  
  
And we should have a closing notebook that concludes the notebook
```

Figure A.7: Example of code document

A.10 Markdown Headings

It is recommended to use markdown headers to organize the notebook for clear structure. In Figure A.8, we use one hashtag (#) for the introduction heading, and in the following content, we use two hashtags (##) to divide them into sections for better readability.

A.11 Concisness

For this practice, it is recommended to have shorter code cell length and fewer code cells in the notebook, aiming for notebooks' conciseness. According to the blog [13], a code cell that contains more than 15 lines is considered lengthy. For real examples and more explanations, please refer to the repository [16].

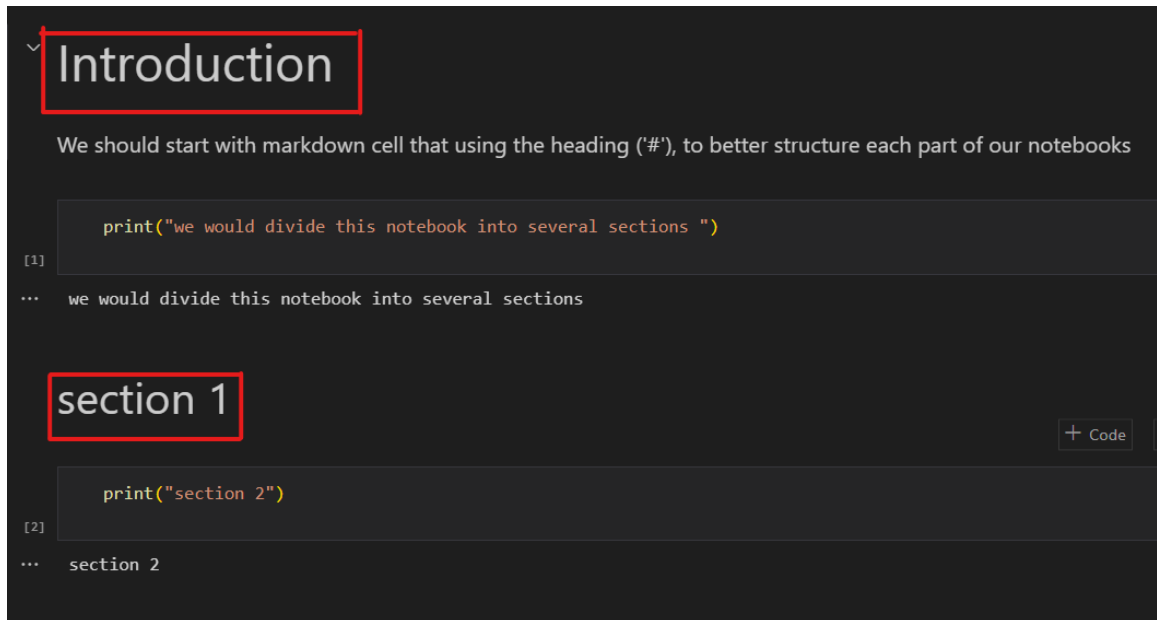
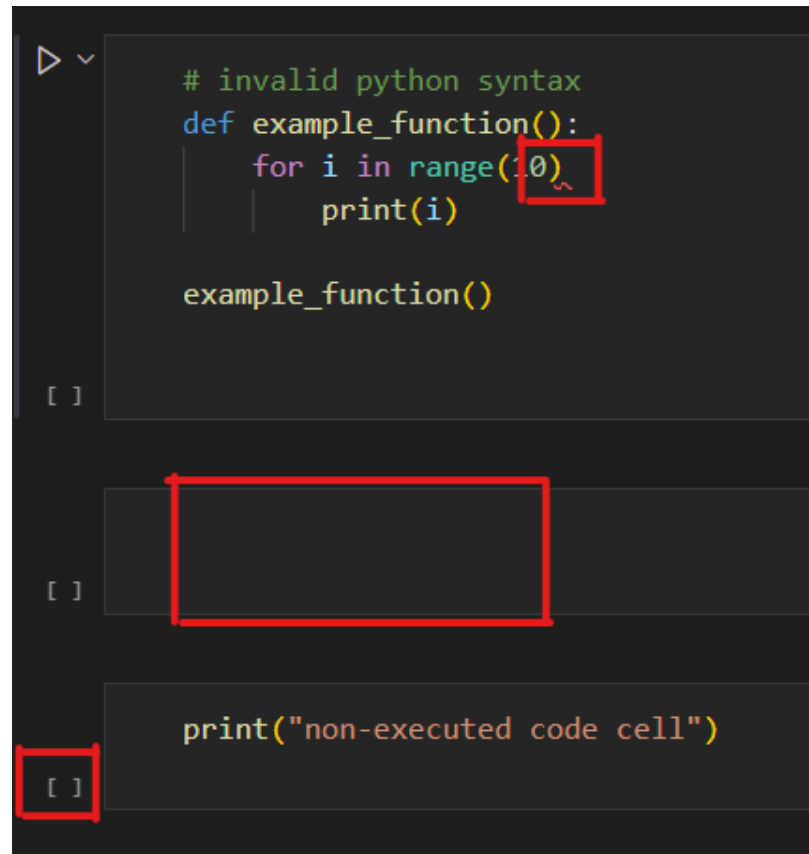


Figure A.8: Example of markdown headings

A.12 Tidiness

Figure A.9 illustrates a notebook that contains unexecuted code cells, empty cells and syntax errors. For notebook tidiness, we should avoid having these problems during notebook development.



```
# invalid python syntax
def example_function():
    for i in range(10):
        print(i)

example_function()

print("non-executed code cell")
```

The image shows a code editor with three code cells. The first cell contains Python code with a red box around the closing parenthesis of the `range(10)` function. The second cell is empty and has a red box around its entire content area. The third cell contains the text `print("non-executed code cell")` and has a red box around its opening bracket.

Figure A.9: Example of tidiness

Appendix B

Implementations

Section 4.1 briefly describes the implementation of quality test: notebook testing, without detailed explanations. In Append B, we explain each helper script and tool used in that quality test in detail. Recall that a list of scripts and tools used is shown in Table 4.1.

B.1 Check_file_exist.sh

'check_file_exist.sh' is a bash script that takes an argument that represents the path of the notebook file and determines whether it exists by using the '-f' flag. If the notebook does not exist, print the error message and exit by 1. Figure B.1 is the screenshot of this script.

```
1 #!/bin/bash
2
3 # Check if a file path is provided as an argument
4 if [ "$#" -ne 1 ]; then
```

```
5     echo "Error: Argument not provided. Usage: $0 path/to/
      file"
6     exit 1
7 fi
8
9 NC='\033[0m' # No Color
10 RED='\033[0;31m' # Red Color
11
12 # Assign the first argument to a variable
13 file_path="$1"
14
15 if [ ! -f "$file_path" ]; then
16     echo -e "${RED}File not found:${NC} $file_path"
17     exit 1
18 else
19     echo "File exists: $file_path"
20 fi
```

Figure B.1: check_file_exist.sh

B.2 Check_functions_defined.sh

‘Check_functions_defined.sh’ (illustrated in Figure B.2) is a bash script designed to verify if functions are defined within the notebook. It operates by analyzing a JSON output file generated by the Pynblint linter. An example of a JSON file is shown in Figure B.3. The script uses the tool ‘jq’, a JSON processor, to process this JSON file and check whether the number of functions is greater than 0.

```
1 #!/bin/bash
2
3 # Check if a file path is provided as an argument
4 if [ "$#" -ne 1 ]; then
5     echo "Error: Argument not provided. Usage:
6         $0 path/to/pynblint output file"
7     exit 1
```

```
8 fi
9
10 # Assign the first argument to a variable
11 file_path="$1"
12
13 # Install jq if not already present
14 sudo apt-get install jq
15
16 GREEN='\033[0;32m' # Green Color
17 RED='\033[0;31m' # Red Color
18 YELLOW='\033[0;33m' # Yellow Color
19 NC='\033[0m' # No Color
20
21 # Check if "notebook_stats.number_of_functions" > 0
22 if [ $(jq '.notebook_stats.number_of_functions' $file_path
23     ) -gt 0 ]; then
24     echo -e "${GREEN} functions defined in notebook${NC}"
25 else
26     echo -e "${RED}No functions found in notebook${NC}"
27     exit 1
28 fi
```

Figure B.2: Check_functions_defined.sh

```
1  {
2    "notebook_metadata": { "notebook_name": "pytest_notebook1.ipynb" },
3    "notebook_stats": {
4      "number_of_cells": 3,
5      "number_of_MD_cells": 0,
6      "number_of_code_cells": 3,
7      "number_of_raw_cells": 0,
8      "number_of_functions": 4,
9      "number_of_classes": 0,
10     "number_of_md_lines": 0,
11     "number_of_md_titles": 0
12   },
13   "lints": []
14 }
15
```

Figure B.3: JSON output of Pynblint

B.3 Check_test_modules.py

The ‘`check_test_modules`’ Python script is designed to verify the presence of specific test modules in a notebook. It requires two inputs: the path to the given notebook and the name of the test module. This script is needed as importing the relevant test module is a prerequisite for running notebooks within the notebook. If the notebook passes the check, we can confirm the notebook is ready for unit testing.

The script utilizes Abstract Syntax Tree (AST), a tool that traverses and interprets different components of Python code. As depicted in Figure B.4, by inputting the notebook path and the targeting test module into the ‘`find_test_modules`’ function at line 60, the script first reads the notebook file using ‘`nbformat`’, then parses it and processes it through an AST node finder, ‘`TestModuleFinder`’. It is instantiated at line 35 and triggered at line 46, processing an ast node structure derived from the notebook’s source code. This node finder contains four methods, each with a specific purpose, as listed in Table B.1. Each method represents a distinct Python code structure which the AST framework processes.

Name	Purpose
<code>visit_Import</code>	visit and process ‘ <code>import</code> ’ statement in the code.
<code>visit_ImportFrom</code>	visit and process ‘ <code>import from</code> ’ statement in the code.
<code>visit_FunctionDef</code>	visit and process ‘ <code>function</code> ’ definitions in the code.
<code>visit_ClassDef</code>	visit and process ‘ <code>class</code> ’ definitions in the code.

Table B.1: List of functions used in the script

```
1 import ast
2 import nbformat
3 import sys
4
5 class TestModuleFinder(ast.NodeVisitor):
6     def __init__(self, target_module):
7         # List of common Python testing modules
8         self.test_module = target_module
9         self.found = None
10
11     def visit_Import(self, node):
12         # Check if any of the imported modules are in the
13         # list of test modules
14         for alias in node.names:
15             if alias.name == self.test_module:
16                 self.found = True
17
18     def visit_ImportFrom(self, node):
19         # Check if the imported module (from 'from x
20         # import y' statement) is a test module
21         if node.module == self.test_module:
22             self.found = True
23
24     def visit_FunctionDef(self, node):
25         # Identify pytest-style test functions by their
26         # name and the presence of assert statements
27         if target_module == "pytest" and node.name.
28             startswith('test_') and any(isinstance(elem,
29             ast.Assert) for elem in ast.walk(node)):
30             self.found = True
31
32     def visit_ClassDef(self, node):
33         # Check for pytest-style test methods within a
34         # class
35         for item in node.body:
36             if isinstance(item, ast.FunctionDef):
37                 self.visit_FunctionDef(item)
38
39 def find_test_modules(notebook_path, target_module):
40     # Initialize the TestModuleFinder
41     finder = TestModuleFinder()
42
43     # Open and read the Jupyter notebook
```

```
38     with open(notebook_path, 'r', encoding='utf-8') as
39         file:
40         nb = nbformat.read(file, as_version=4)
41     # Iterate through each cell of the notebook
42     for cell in nb.cells:
43         if cell.cell_type == 'code':
44             # Parse the code cell and search for test
45             # modules
46             tree = ast.parse(cell.source)
47             finder.visit(tree)
48             # Return the found test module, if any
49             if finder.found:
50                 return finder.found
51     # Return None if no test module is found
52     return None
53
54 if __name__ == "__main__":
55     # Get the notebook path from command line arguments
56     notebook_path = sys.argv[1]
57     target_module = sys.argv[2]
58
59     # Print the result and exit appropriately
60     if find_test_modules(notebook_path, target_module):
61         print(f"{target_module} found in notebook.")
62         sys.exit(0)
63     else:
64         print(f"{target_module} not found in notebook.")
65         sys.exit(1)
```

Figure B.4: check_test_modules.py

The AST framework will traverse the notebook cell by cell from the beginning. When it encounters classes or functions, it delves into the body of `visit_FunctionDef` or `visit_ClassDef` at lines 22 and 27. If the target test module is detected in either `visit_import` or `visit_ImportFrom`, the script sets the found flag as `TRUE`, indicating the presence of the target module within the notebook.

It is important to note that Pytest does not require an explicit import statement

to be used within the notebook. The presence of a function name that starts with `‘test_’` and includes an assert statement is also a sign that Pytest unit tests exist. Therefore, an additional check is performed in the `‘visit_FunctionDef’` method of the script to look for `‘test_’` and assert statements. That ensures Pytest can be recognized even when it is not explicitly imported.

B.4 Pynblint

As introduced in the background section, Pynblint is a linter that can analyze notebook contents and provide recommendations. In our quality test implementations, we utilize Pynblint to analyze the notebook, save the result into a JSON file, and process it, just like how we did in the above `check_functions_defined.sh` sections.

Pynblint provides statistic reports about the notebook, including the number of functions, classes, cells, etc. It also provides recommendations based on the lint rules selected. A partial list of lint rules available in Pynblint is outlined in Table B.2. Users can choose any rules they need and apply them to their notebooks. For a complete list of lint rules, please refer to the Pynblint repository [28].

For example, suppose users want to check whether the notebook has a linear execution order and imports at the beginning. In that case, they can run the command: `‘pynblint –include [“non-linear-execution”, “imports-beyond-first-cell”]’ notebook.ipynb`, and Pynblint will process the notebook file and display the lint results to the console. Users can specify which lint rules to apply by adding them after the `–include` and within the `[‘’]`. If the notebook fails to follow any of the lint rules they choose, Pynblint will report this either in the console output or, if

Name	Description
non-linear-execution	Notebook cells have been executed in a non-linear order.
notebook-too-long	The notebook is too long (we can customize threshold).
untitled-notebook	The notebook still has the default title.
non-portable-chars-in-nbname	The notebook filename contains non-portable characters.
notebook-name-too-long	The notebook filename is too long.
imports-beyond-first-cell	Import statements found beyond the first cell of the notebook

Table B.2: List of Pynblint Lint rules [28]

the results are directed to a JSON file, as specific attributes, like the example in Figure B.5. In most of our quality test implementations, we save the outputs of Pynblint to a JSON file by using a built-in Pynblint flag: ‘**–output lint-results.json**’.

B.5 Coverage

Coverage is a tool for measuring Python programs’ code coverage. We use this tool to execute the unit tests of the notebook and measure the test coverage. To run Pytest tests, run the command: ‘**coverage run -m pytest test_script.py**’, where `test_script` is the file’s name containing Pytest tests. This command displays the test results, revealing how many tests passed or failed, and records the coverage data in

```

1  {
2  "notebook_metadata": { "notebook_name": "failed_notebook.ipynb" },
3  "notebook_stats": {
4    "number_of_cells": 2,
5    "number_of_MD_cells": 0,
6    "number_of_code_cells": 2,
7    "number_of_raw_cells": 0,
8    "number_of_functions": 0,
9    "number_of_classes": 0,
10   "number_of_md_lines": 0,
11   "number_of_md_titles": 0
12  },
13  "lints": [
14    {
15     "slug": "non-linear-execution",
16     "description": "Notebook cells have been executed in a non-linear order.",
17     "recommendation": "Re-run your notebook top to bottom to ensure it is reproducible."
18    }
19  ]
20 }
21

```

Figure B.5: Example of Pynblint JSON lint result

a report file. For instructions on running tests with other tools, please refer to the Coverage repository [3].

To view the test coverage report, run the command: `coverage report`, which outputs the test coverage details. An example of such a report can be seen in Figure B.6. Additionally, the flag `-failed-under=80` can be included to raise an error if the test coverage falls below 80

Name	Stmts	Miss	Cover
test_script1.py	7	0	100%
TOTAL	7	0	100%

Figure B.6: Coverage report

B.6 Nbconvert

Nbconvert is a tool to convert jupyter notebooks into other formats. Given that the Coverage tool is designed to work exclusively with Python programs, we use the command: `jupyter nbconvert --to script notebook.ipynb` to convert the specified `'notebook.ipynb'` file into a `'notebook.py'` file. Once this conversion is complete, the Coverage tool utilizes the generated Python script for further processing.

Bibliography

- [1] Monya Baker. 2016. 1,500 scientists lift the lid on reproducibility. *Nature* 533, 7604 (2016).
- [2] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. 2016. Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software* 33, 3 (2016), 42–52.
- [3] Ned Batchelder. [n. d.]. coveragepy. <https://github.com/nedbat/coveragepy>
- [4] Fabien CY Benureau and Nicolas P Rougier. 2018. Re-run, repeat, reproduce, reuse, replicate: transforming code into scientific contributions. *Frontiers in neuroinformatics* 11 (2018), 69.
- [5] Bestarion. 2022. What is CI/CD and How Does It Work? <https://www.linkedin.com/pulse/what-cicd-how-does-work-bestarion/> Accessed on Dec 22, 2024.
- [6] Andreas Brunnert, André van Hoorn, Felix Willnecker, Alexandru Danciu, Wilhelm Hasselbring, Christoph Heger, Nikolas Herbst, Pooyan Jamshidi, Reiner Jung, Joakim von Kistowski, et al. 2015. Performance-oriented DevOps: A research agenda. *arXiv preprint arXiv:1508.04752* (2015).

- [7] Lianping Chen. 2015. Continuous delivery: Huge benefits, but challenges too. *IEEE software* 32, 2 (2015), 50–54.
- [8] Michael Cheng and Viacheslav Kovalevskiy. 2019. Jupyter Notebook Manifesto: Best practices that can improve the life of any developer using Jupyter notebooks. <https://cloud.google.com/blog/products/ai-machine-learning/best-practices-that-can-improve-the-life-of-any-developer-using-jupyter-notebooks> Accessed on Dec 25, 2023.
- [9] Martin Fowler. 2024. Continuous Integration. <https://martinfowler.com/articles/continuousIntegration.html> Accessed on Jan 20, 2024.
- [10] Agile Guide. n.d.. What Is Definition of Ready? (DoR). <https://www.wrike.com/agile-guide/faq/what-is-definition-of-ready/> Accessed on Mar 10, 2024.
- [11] Matthew Hutson. 2018. Artificial intelligence faces reproducibility crisis. <https://www.science.org/doi/full/10.1126/science.359.6377.725>
- [12] Donald E Knuth. 1984. Literate programming. *Comput. J.* 27, 2 (1984), 97–111. <https://doi.org/10.1093/comjnl/27.2.97>
- [13] Atma Mani. 2018. Coding Standards for Jupyter Notebook. <https://www.esri.com/about/newsroom/arcuser/coding-standards-for-jupyter-notebook/> Accessed on Dec 25, 2023.
- [14] Lj Miranda. 2020. How to use Jupyter Notebooks in 2020 (Part 2: Ecosystem growth). <https://ljvmiranda921.github.io/notebook/2020/03/16/jupyter-notebooks-in-2020-part-2/> Accessed on Dec 25, 2023.

- [15] Sébastien Mosser, Aaron Loh, Deesha Patel, and Nirmal Chaudhari. [n. d.]. `jpipe`.
<https://github.com/ace-design/jpipe>
- [16] Sébastien Mosser and Kai Sun. [n. d.]. `notebook-best-practices`. <https://github.com/ace-design/notebook-best-practices>
- [17] Peter Parente. 2020. `nbestimate`. <https://github.com/parente/nbestimate/tree/master>
- [18] Deesha Patel. 2023. *A STUDY ON JUSTIFYING PLATFORM-INDEPENDENT CI/CD PIPELINES*. Technical Report. <https://macsphere.mcmaster.ca/handle/11375/29326>
- [19] Alexis Perrier. 2020. Tips to make you data analysis easier to share. https://alexisperrier.com/datascience/2020/02/15/jupyter_notebooks_sharing_best_practices.html Accessed on Dec 25, 2023.
- [20] João Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. 2019. A large-scale study about quality and reproducibility of Jupyter notebooks. In *2019 IEEE/ACM 16th international conference on mining software repositories (MSR)*. IEEE, 507–517.
- [21] João Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. 2021. Understanding and improving the quality and reproducibility of Jupyter notebooks. *Empirical Software Engineering* 26, 4 (2021), 65.
- [22] STEN PITTET. n.d.. What is code coverage? <https://www.atlassian.com/>

- `continuous-delivery/software-testing/code-coverage` Accessed on Mar 15, 2024.
- [23] Thomas Polacsek. 2016. Validation, accreditation or certification: a new kind of diagram to provide confidence. In *2016 IEEE Tenth International Conference on Research Challenges in Information Science (RCIS)*. IEEE, 1–8.
- [24] Python Enhancement Proposals. n.d.. PEP8 - Style Guide for Python Code. <https://peps.python.org/pep-0008/#imports> Accessed on Jan 10, 2024.
- [25] Corinne Pulgar. 2022. Eat your own DevOps: a model driven approach to justify continuous integration pipelines. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. 225–228.
- [26] Luigi Quaranta, Fabio Calefato, and Filippo Lanubile. 2022. Eliciting best practices for collaboration with computational notebooks. *Proceedings of the ACM on Human-Computer Interaction* 6, CSCW1 (2022), 1–41.
- [27] ReviewNB. 2019. Reproducible Jupyter Notebooks with Docker. <https://blog.reviewnb.com/reproducible-notebooks/> Accessed on Dec 30, 2023.
- [28] Vincenzo Romito, Luigi Quaranta, Felice Tortorelli, and Filippo Lanubile. [n. d.]. `pynblint`. <https://github.com/collab-uniba/pynblint>
- [29] Adam Rule, Amanda Birmingham, Cristal Zuniga, Ilkay Altintas, Shih-Cheng Huang, Rob Knight, Niema Moshiri, Mai H Nguyen, Sara Brin Rosenthal, Fernando Pérez, et al. 2019. Ten simple rules for writing and sharing computational analyses in Jupyter Notebooks. , e1007007 pages.

- [30] Julia Wagemann, Federico Fierli, Simone Mantovani, Stephan Siemen, Bernhard Seeger, and Jörg Bendix. 2022. Five guiding principles to make Jupyter notebooks fit for earth observation data education. *Remote Sensing* 14, 14 (2022), 3359.
- [31] Mark D Wilkinson, Michel Dumontier, IJsbrand Jan Aalbersberg, Gabrielle Appleton, Myles Axton, Arie Baak, Niklas Blomberg, Jan-Willem Boiten, Luiz Bonino da Silva Santos, Philip E Bourne, et al. 2016. The FAIR Guiding Principles for scientific data management and stewardship. *Scientific data* 3, 1 (2016), 1–9.
- [32] Mark Woodbridge, Daniel Sanz, Daniel Mietchen, and R Mounce. 2017. Jupyter notebooks and reproducible data science. *Retrieved January 24 (2017), 2020.* <https://markwoodbridge.com/2017/03/05/jupyter-reproducible-science.html>