communicated by:
Lehr- und Forschungsgebiet Informatik 9

Prof Dr. Ulrik Schroeder



The present work was submitted to Learning Technologies Research Group
Diese Arbeit wurde vorgelegt am Lehr- und Forschungsgebiet Informatik 9

Redesign and Implementation of nbgrader to Improve the Instructor User Experience

Redesign und Implementierung von nbgrader zur Verbesserung der Benutzerfreundlichkeit für Lehrende

Bachelor-Thesis
Bachelorarbeit

presented by / von

Thape, Felix
422192

Prof. Dr. Ulrik Schroeder

Prof. Dr. Jürgen Giesl

Aachen, February 2, 2024

Contents

List of Figures iii

List of Tables v

List of Listings vi

1 Introduction 2
- 1.1 Motivation . . . 2
- 1.2 Research Questions and Goals . . . 2
- 1.3 Structure . . . 3

2 Related Work 4
- 2.1 Automatic Assessment . . . 4
- 2.2 Automatic Assessment systems . . . 5

3 nbgrader 7
- 3.1 Jupyter Notebook documents in nbgrader . . . 7
- 3.2 Workflow and folder structure . . . 10
  - 3.2.1 Main process folder structure . . . 11
  - 3.2.2 Exchange service . . . 12
- 3.3 User interface . . . 13
- 3.4 Client Server Connection . . . 16
- 3.5 Architecture . . . 17
  - 3.5.1 Notebook Level Components . . . 18
  - 3.5.2 Assignment Level Components . . . 18
  - 3.5.3 High Level Components . . . 18
  - 3.5.4 Other Features . . . 19
  - 3.5.5 Client-side Components . . . 19
- 3.6 Summary . . . 20

4 Requirements Study 21
- 4.1 Study Design . . . 21
- 4.2 Results . . . 21
  - 4.2.1 Summary . . . 23
- 4.3 Threats to validity . . . 24
- 4.4 Requirements . . . 24
  - 4.4.1 Functional Requirements . . . 24
  - 4.4.2 Non-functional Requirements . . . 26

5 Analysis and Design **27**
5.1 Analysis . . . 27
5.1.1 Core requirements . . . 27
5.1.2 Requirements with normal priority . . . 29
5.1.3 Optional Requirements . . . 30
5.1.4 Summary . . . 31
5.2 Design . . . 31

6 Implementation **34**
6.1 Changes to the Autograding Process . . . 34
6.2 Changes to Manual Grading Process . . . 35
6.3 Hiding empty test cells . . . 36
6.4 Error dialog . . . 36
6.5 Validation tests . . . 37
6.6 Self-assessment . . . 37
6.7 Autograde all button . . . 38
6.8 Group Submissions . . . 38
6.9 Duplicate Cell Identifier Warnings . . . 39
6.10 Non-Functional Requirements . . . 40
6.11 Summary . . . 40

7 Validation Study **41**
7.1 Study Design . . . 41
7.1.1 Introduction part . . . 41
7.1.2 Main part . . . 41
7.1.3 Closing part . . . 42
7.2 Results . . . 43
7.3 Summary . . . 48

8 Results **49**

9 Future Work **50**

Appendix **50**

A Preprocessor List **51**

B Converter List **52**

C Bibliography **54**

D Digital Appendix **58**

List of Figures

2.1 check50 components 6

3.1 Example Jupyter Notebook document in JupyterLab editor 8
3.2 nbgrader workflow instructor and student actions 11
3.3 Notebook document editor with validate button and sidebar for metadata 13
3.4 Manage assignments screen of the formgrader extension 14
3.5 Manage submissions screen of the formgrader extension 14
3.6 Screen with submissions ready for manual grading of the formgrader extension 15
3.7 Screen for manual grading of a Notebook document in the formgrader extension 15
3.8 Manage Students screen of the formgrader extension 16
3.9 Assignment list 16
3.10 nbgrader client and server connection 17
3.11 nbgrader architecture layers 17

5.1 Autograding converter change (Sanitization phase) 31
5.2 Self assessment workflow 32

6.1 Adding of missing cells example 34
6.2 Annotation of added missing cells and cells at the wrong position 35
6.3 Command for listing and changing the grades for a cell in a student submission 35
6.4 Manual grading source cell copy 36
6.5 Error message (before version) 37
6.6 Error message (new version) 37
6.7 Self assessment assignments view 38
6.8 Submission screen (older version) 38
6.9 Submission screen (new version) 38
6.10 Duplicate grade identifier warnings 39

7.1 Picture for Student Assignments Comparison 42
7.2 JupyterLab usage 43
7.3 nbgrader usage 43
7.4 Rating of Generation Process 43
7.5 Rating of Generation Error Messages 43
7.6 Number of Errors found by Participants 44

7.7 Autograding Process Rating 44
7.8 Manual Grading Process Rating 44
7.9 Autograding Error Messages Rating 44
7.10 nbgrader version comparison 45
7.11 Self assessment Ratings 45
7.12 Rating of Group submission submitting 46
7.13 Rating of Group submission grading effort 46
7.14 Modified System Usability Scale Rating 46

List of Tables

3.1 nbgradermetadata keys and meaning [nbgh] 8
3.2 Meaning of cell types 9
3.3 Metadata values to nbgrader cell type mapping 9
3.4 Special syntax of nbgrader's assignment creation [nbgc] 9
3.5 nbgrader steps from [nbge] used as a table 11
3.6 nbgrader steps from [nbgd] used as a table 12

6.1 Implemented requirements (functional) 40
6.2 Implemented requirements (non-functional) 40

List of Listings

Abstract

This thesis focuses on the automatic assessment framework nbgrader, its architecture, and the instructor's user experience using nbgrader.
Two qualitative studies were conducted. The first one determined the requirements for the implementation and examined nbgrader's usability for the instructor. These requirements were used in three phases: analysis, design, and implementation. The analysis describes prerequisites for the requirements and looks at implementations already present in nbgrader. The design phase presents different solutions, and the implementation explains the solution used. The second study was used to determine the new nbgrader version's usability and to compare it to the previous version.
The requirements study had one participant and is therefore not representative. The participants were asked to rate their agreement or disagreement with statements regarding nbgrader's usability on a scale and justify their rating. The most relevant result was that errors during assignment generation were hard to attribute to a specific Notebook document or cell and that errors in student submissions caused the autograding to fail, decreasing usability. Manual grading and feedback generation were rated very positively, and assignment creation was rated positively.
The requirements focus on changing the automatic and manual grading process to allow for grading submissions that produced errors in the previous version. They also include simplified error messages and new features like student self assessments and group submissions. These requirements were implemented, and a proof of concept was created for the group submissions.
In the second study, 6 participants were presented with tasks in an unmodified and a modified version of nbgrader and asked to rate statements on a scale to compare the two versions. Functions not available in the unmodified version were subsequently presented and also rated. The study shows clear indications of an improvement in usability but also weaknesses in the validity that should be considered before using the results.
This thesis's architecture description of nbgrader can be used to analyze and improve the nbgrader framework architecture or to compare it to other frameworks. The results of the studies can be used to increase nbgrader's usability.

Chapter 1 Introduction

Instructors and teachers in computer science often use exercises and assignment sheets to let their students practice the contents of the lecture. Many of these exercises involve programming tasks. An example of such an exercise could be a function that should use two parameters to create a specific output. As these exercises are often already written in the source code of a specific programming language, tools were developed to save the instructors the trouble of executing each student submission themselves and writing down the output and differences to the solution. These tools automatically assess student submissions, and systems and frameworks were developed to create and grade assignments.

1.1 Motivation

Despite more than 170 tools and systems for automatic assessment in the system corpus by Strickroth [str], many still have problems when used in a real-world educational context. Ihantola et al. mention that there has been a high interest in creating new tools and frameworks instead of improving existing ones [IAKS10].
For that reason, this thesis focuses on one specific framework, *nbgrader* [JBB+19], which is already used at the RWTH Aachen University. From the experience made by instructors using it, different problems are already known and will be regarded to improve the usability of *nbgrader*. In order to include the perspectives of other instructors, a study is used to find improvements, which will be validated with a second study.
Opposed to the preexisting literature on nbgrader, which often focuses on the student user experience and the integration into the teaching environment, this thesis focuses on the instructor's user experience and *nbgrader's* architecture.

1.2 Research Questions and Goals

The goals of this thesis besides improving *nbgrader*'s user experience are answering the following research questions:

1. How does nbgrader work, and which architecture does it use?
2. What problems do instructors have while using nbgrader?
3. What are the changes that need to be made to solve these issues?
4. Do users prefer the changed version of nbgrader compared to the previous version?

The results to these questions can be found in chapter 8.

1.3 Structure

The thesis is structured as follows: First, related work is described in chapter 2. The instructor's workflow and architecture of nbgrader are looked at in chapter 3. Chapter 4 provides an overview of problems instructors face when using nbgrader, leading to requirements for changes to nbgrader. The analysis and designs for the requirements are described in chapter 5. In chapter 6, the implementations of these requirements are shown. A validation study in chapter 7 is then used to evaluate the changes made.
The chapter 8 summarizes the results of this thesis, and the chapter 9 outlines future work and applications of the thesis.

Chapter 2 Related Work

nbgrader extension in the Jupyter framework

The *nbgrader* is an extension to Jupyter Notebook [jupf] or JupyterLab [jupe] and allows for creating and grading assignments written in Jupyter Notebook documents [JBB+19]. If used in JupyterHub [jupc] or configured correctly for Jupyter Notebook or JupyterLab, nbgrader can also be used to release and fetch assignments and feedback. A command line application for using nbgrader is also provided [nbgg]. In [JBB+19], the authors mention that previously Jupyter Notebook documents were not used frequently for large scale graded assignments and now with nbgrader this is possible.

Jupyter framework and software

Jupyter Notebook documents are JSON-based documents that can be used with editors like the Jupyter Notebook or JupyterLab to create an interactive environment for programming and visualizing data [jupa]. A Jupyter Notebook document contains different cells, including code, Markdown, and plain text. Code cells can be directly executed, and results are shown below the executed cell. JupyterHub is a system for managing and deploying multi-user Jupyter Notebook or JupyterLab instances.

experiences with nbgrader

In [Kle20], the author describes the usage of nbgrader for a university course with JupyterHub and calls the experience positive and encouraging for using the JupyterHub server with nbgrader for more courses. Quantitative data from student feedback in [Lan19a] shows that Jupyter Notebook documents helped the students analyze the data in an engineering class. The author also wants to use nbgrader and Jupyter Notebook documents in more classes.

A detailed description of nbgrader will follow in chapter 3. The rest of this chapter will regard automatic assessment and automatic assessment systems.

2.1 Automatic Assessment

assessment in general

Assessment, as described in [IAKS10], provides the teacher or outside observer with a method of obtaining feedback to see whether learning goals by students are achieved and gives the students guidance and feedback for the learning process. This definition comes from the context of computer science education and teaching programming. However, assessment is a broad term and can be used nearly anywhere, for example, in psychology to try to assess depression based on facial cues automatically [PSM+17] or geology for assessing the landslide susceptibility of an area [SPLC20]. This thesis will only look at the context of computer science education and, more concretely, the teaching of programming to students.

automatic assessment

Automatic assessment specifies that the assessment process is automated and that no intervention by a human is needed. The automation of assessments has the advantage

that more students can participate in the learning experience, as instructors need less time for grading [HNVB21], thus creating a positive benefit for education.

question and responses in assessments

Assessments consist of tasks and questions that can be classified. One classification is to examine whether the question is closed-ended, which means it has a clear answer, or whether it is open-ended, where the answer is given as a freely written text [TRR00]. Closed-ended questions are easier to use for automatic assessment because the answer's correctness can be determined by comparing the solution and the response. Another way is to look at the response format, whether structured like program code, free text, handwriting, or something entirely different [HNVB21]. Each of these formats affects how automatic assessment can be used and how difficult it is to implement.

In general, automatic assessment refers to the automatic process of assessing something and providing feedback, and the next section will describe the systems used to achieve that.

2.2 Automatic Assessment systems

automatic assessment systems and features

Software solutions can implement automatic assessment. The features supported by such automatic assessment systems vary. In [IAKS10], the authors describe these features as the supported programming languages, the integration into a learning management system, the tests used for automatic assessment, the way resubmissions are handled, how and whether manual grading is integrated, and the sandboxing used by the system.

programming languages

Strickroth et al. [str, SS22] list 170 systems supporting programming languages like Java, Python, C/C++, Assembly, and Unix shell scripts. No concrete programming language will be looked at until the examples of automatic assessment systems in paragraph 2.2.

testing strategies

A differentiation of testing strategies for assessing the functionality of programs is given in [Wil16]. The methods mentioned by the author range from output comparison to simulating user inputs and testing for the correct response by the system. A different classification of tests into dynamic, static, and hybrid tests is also possible [TAHE22]. The student code is run as a program for dynamic tests and possibly compiled before its execution. Static tests only look at the without execution, for example, by looking at the source code as a text and checking for the occurrence of a specific word. Hybrid tests combine both dynamic and static tests.

security, sandboxing, container

Automatic assessment software also has to deal with problems like student code not terminating and the threat that the execution of the student code might compromise the assessment system [Wil16]. The authors in [PMC19] compare two solutions, sandboxes and containers, regarding performance and conclude that performance differences are negligible. Jailed Sandboxes are more complex to configure, but perform slightly better than containers.

plagiarism detection and feedback

In a literature review by Tarek et al., the authors mention plagiarism detection and the feedback of automated assessment systems as important aspects [TAHE22]. Susanne Narciss uses different classifications regarding feedback, which can be used to determine how good the feedback given by an automated assessment system is [Nar06].

architectural model for automatic assessment systems

An architectural model for automatic assessment systems using layers and different components is described in [FL14]. These include an interface layer for the connections to various outside systems, a layer for managing the communication of the different components, a layer for grading assignments, and one for distributing work to the grading layer. Outside systems are, for example, the problem set data, a learning management system, or a training interface. Two additional layers are used, one for the database and one for cross-cutting concerns like security, which do not have to be a direct part of the system.
Automatic assessment systems can have many features. The following section will introduce some of these systems.

check50 automatic assessment

Examples An example of a small automatic assessment system is check50 [SvAY+20]. Figure 2.1 shows the process of automatically assessing an assignment with it.

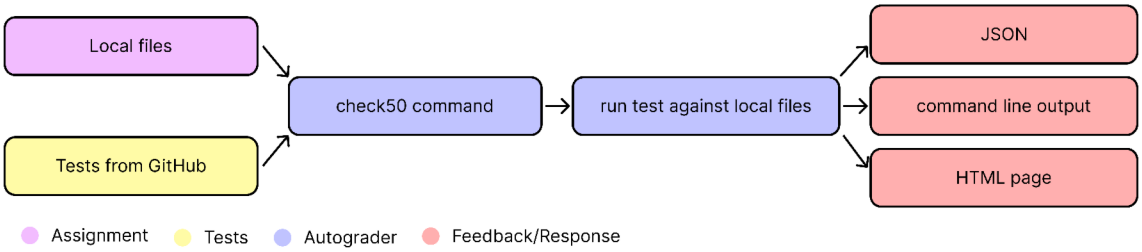


Figure 2.1: check50 components

The check50 command uses the student's local assignment solution and tests from GitHub to create feedback on the console in JSON format or as an HTML page. This shows the possible separation of the tests and the automatic grading, allowing for reusability.

FREJA assessment generation

FREJA is an example of a framework for creating programming assignments in Java [Ber22]. It only requires one artifact, a solution project, which is annotated with Java annotations by the instructor and then used to create the student's start code and assignment description. It does not provide autograding functionality, but the method of using annotations to remove solutions from an assignment and using a separate version of an assignment for students will be found again when looking at nbgrader.

UNCode Notebook auto-grader autmatic assement system

The UNCode Notebook auto-grader is an extension to UNCode [Cri21], an automatic assessment framework that focuses on providing instant feedback to the students. It allows for a complex testing system. Tests can have time and memory limits and are divided into different test cases, with the option to create setup code for a test, allowing for reusability between test cases. Submissions are handled in a submission queue and forwarded to a docker agent to execute each test securely and independently in a sandboxed environment.

Different automatic assessment systems were described, and features of these were presented. The next chapter will focus on nbgrader specifically, showing what features are supported and how nbgrader can be used.

Chapter 3 nbgrader

nbgrader usage

The primary usage of *nbgrader* is creating and automatically grading assignments with Jupyter Notebook documents containing Python code. Other programming languages like R, MatLab, and Java can also be supported when a Jupyter Kernel [jupd] for them is provided. The nbgrader is an extension for Jupyter Notebook and JupyterLab but can also be used on the command line. Releasing assignments and feedback and making student submissions can be made directly with nbgrader when used in a Unix multi-user system and configured correctly [JBB+19, nbgf]. All configurations for nbgrader can be done through a configuration file: *nbgrader_config.py*.

general information

The current version of nbgrader is 0.9.1 [nbgf]. It is mainly written in *Python* [pyt] and a repository maintained by the *Project Jupyter* GitHub organization [pro].

Assignments

Most actions in nbgrader are applied to assignments. *assignments* in the context of nbgrader are folders containing Jupyter Notebook documents and optionally additional files. Jupyter Notebook documents will from now on be shortened to *Notebook documents*.

chapter structure

Notebook documents in nbgrader have a unique schema for metadata for semantic purposes and a special syntax for cells. These will be introduced first in section 3.1. The following section 3.2 will describe the workflow for an assignment from creation to student feedback and how the folder structure of a nbgrader instance mirrors that workflow. The user interface will be described in section 3.3, and section 3.4 regards the communication of different nbgrader extensions. After that, the architecture of nbgrader will be shown and explained in section 3.5.

3.1 Jupyter Notebook documents in nbgrader

A Notebook document is a JSON-based file type [jupa]. Editor programs that can interpret these files provide an interactive development environment.

cells, metadata, usage

Figure 3.1 shows a Notebook document in the JupyterLab editor. Different cells in which text can be inserted can be seen. The interpretation of a cell is dependent on its metadata defined in the Notebook document. For example, the first cell with "Task 1" and the task description is set and interpreted as Markdown, while the cell below is set and interpreted as Python source code. Cells can be executed, and variables and functions of code cells are saved as if they were global variables and thus are usable in other cells. The execution results can be seen in (for Markdown) or below the cell (for code). The result of code cells might be a graphic, a console output, or a variable's value. The execution of a cell can be repeated, allowing for visible and direct changes during development.

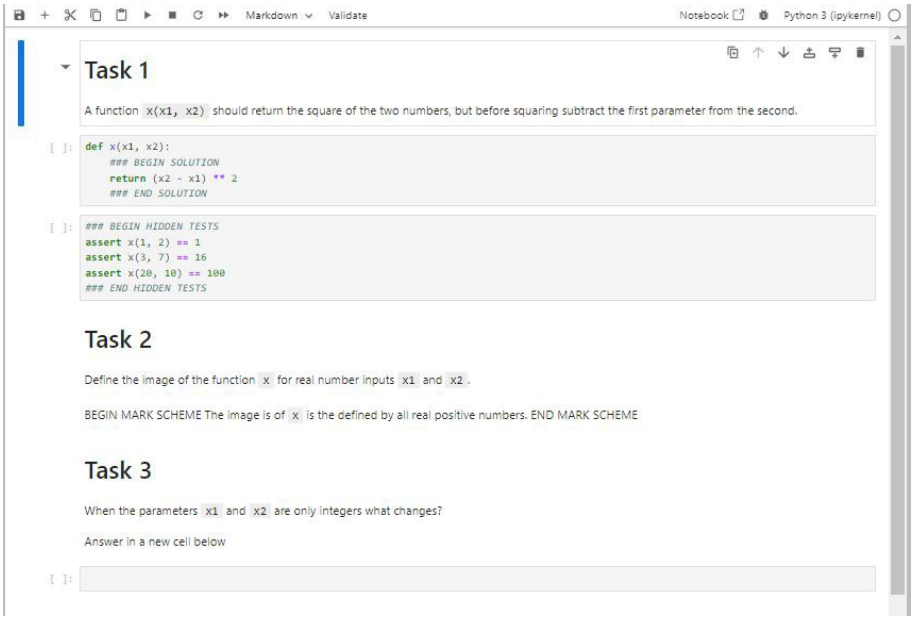


Figure 3.1: Example Jupyter Notebook document in JupyterLab editor

nbgrader cell metadata

Similar to setting the cell type with metadata to Markdown, code, or plaintext, nbgrader uses metadata for each cell to add additional information. This metadata includes information about the nbgrader cell type, a unique identifier, the schema version, points for the cell, a checksum, a locked flag, and a copy of the Notebook document cell type. The Table 3.1 lists all keys of the metadata key-value pairs, their meaning, and whether they are optional or required. These keys form the versioned schema called *nbgraderformat*, which defines the metadata format for all cells using nbgrader metadata. By defining such a schema, it is possible to validate it automatically.

metadata key	meaning	required/optional
grade	whether this cell should be graded	**required**
solution	whether this cell contains a solution	**required**
task	whether this cell contains a task	optional
schema_version	the version of the cell metadata schema	**required**
locked	whether the cell should be editable	**required**
grade_id	a unique identifier of the cell	optional
points	the number of points this cell is worth	optional
checksum	a checksum of the cell contents	optional
cell_type	cell type when nbgrader generate_assignment was run	optional

Table 3.1: nbgradermetadata keys and meaning [nbgh]

nbgrader cell type

From the schema keys in Table 3.1 we see three keys *grade*, *solution*, *task*. These determine the semantic type of the cell. Only *grade* and *solution* are required values, but the combination of all three of them determine the type and purpose of the cell. When looking at nbgrader's client-side implementation and user interface there are

five possible nbgrader cell types for each cell, *Read-only*, *Manually graded task*, *Manually graded test*, *Autograded tests* and *Autograded answer*. An explanation for each of these cell type's meaning can be found in Table 3.2.

cell type	meaning
Read-only	A cell which the student should not edit
Manually graded task	A read-only Markdown for giving a task which the student should solve in multiple cells
Manually graded answer	A cell where the student gives answer, which must be manually graded
Autograded answer	A cell which will be automatically graded and contains the student's answer
Autograded tests	A cell which contains a automatic test and awards full points when executed without error

Table 3.2: Meaning of cell types

In addition to the five nbgrader cell types, a cell can also be of no type, when no nbgrader metadata is used for a cell. The mapping of the three boolean schema values to the cell types is explained by the Table 3.3.

grade	solution	task	nbgrader cell type	allowed Jupyter Notebook document cell types
false	false	false	Read-only	Markdown, Raw, Code
false	true	false	Autograded answer	Code
true	false	false	Autograded tests	Code
false	false	true	Manually graded task	Markdown
true	true	false	Manually graded answer	Markdown, Raw, Code

Table 3.3: Metadata values to nbgrader cell type mapping

It can be seen that 3 out of the 8 possible combinations are not used, and some nbgrader cell types are only usable for specific Jupyter Notebook document cell types.

In addition to the metadata, nbgrader uses a special syntax in certain cells, which can be seen in Table 3.4.

special syntax	meaning	nbgrader cell type
### BEGIN SOLUTION	Starts a solution block	Autograded answer
### END SOLUTION	Ends a solution block	Autograded answer
### BEGIN HIDDEN TESTS	Starts a hidden tests block	Autograded tests
### END HIDDEN TESTS	Ends a hidden tests block	Autograded tests
### AUTOTEST <C>	Visible generated test	Autograded tests
### HASHED AUTOTEST <C>	Visible generated test with hashed values	Autograded tests
BEGIN MARK SCHEME	Starts a mark scheme block	Manually graded task
END MARK SCHEME	Ends a mark scheme block	Manually graded task

<C>: Is an arbitrary code object

Table 3.4: Special syntax of nbgrader's assignment creation [nbgc]

special syntax in cells

For example, the "### BEGIN SOLUTION" and "### END SOLUTION" blocks in the first code cell of Figure 3.1 are used in the source version of an assignment to remove the solution between the two markers from the student version. The source and student versions are explained in section 3.2. Apart from the *Solution* block, which is used for *Autograded answer* cells, similar blocks exist for *Autograded tests* cells with the *Hidden Tests* blocks and for *Manually graded task* cell with the *Mark Scheme* blocks. The *Autotest* marker can be used to automatically generate a test with the solution based on the object or objects given as a parameter. The solutions are visible to the students in the tests. Therefore, a second marker Hashed Autotest, is used to obfuscate the solution in the test by hashing it. Both automatic tests require an *autotests.yml* file for creating a test from the object parameter.

Summary The nbgrader uses a schema for each cell to save additional information about it. Some parts of this information are used to define a cell's purpose, identify it, and limit students' actions while editing a cell.

Furthermore, a special syntax is used to replace or remove marked blocks from the assignment's source version when generating the student version.

The following section will look at the nbgrader workflow, how these notebooks are used in the assignments and the explanation of the source and student versions of an assignment.

3.2 Workflow and folder structure

The nbgrader workflow starts with creating an assignment and ends with generating feedback for the student. Additionally, nbgrader allows for distribution functions, for example, releasing created assignments to students as the instructor and submitting assignments as a student. These extend the main nbgrader workflow. Furthermore, two other functions may be done without the nbgrader extension or application. For many steps of the process, the assignments are saved to a specific folder structure, representing the assignment's state.

nbgrader workflow

The Figure 3.2 shows each step of the nbgrader workflow with student and instructor actions. In the first step, the instructor creates an assignment for which cells and the corresponding metadata are added to the assignment's Notebook documents. The student version is generated from this source version and can be published. The assignment then gets fetched, edited, and submitted by the student. Then, the instructor can collect and automatically grade the assignment. During the automatic grading, the student's submission is processed by making an entry in the database, copying in hidden tests of the source version, and executing the entire notebook. Autograded test cells that produce no error during the execution award all points to the student for that test, and any error leads to zero points. After that, the assignment can be manually graded by adding comments and changing task scores. Then, feedback can be generated and published for the students to fetch and look at. The manual grading can be skipped, and feedback can be generated immediately.

different nbgrader functions

The different colors in Figure 3.2 differentiate the functions nbgrader provides in three categories: the main features, which all describe the state of the assignment,

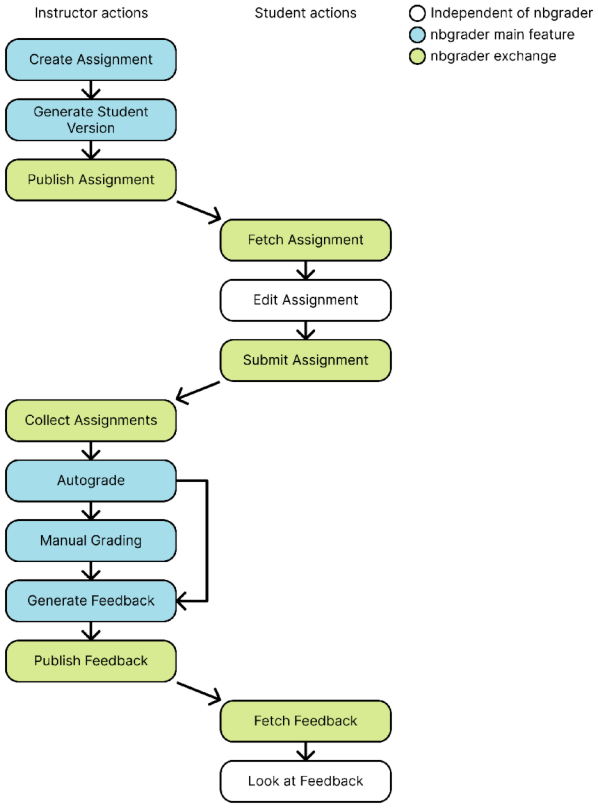


Figure 3.2: nbgrader workflow instructor and student actions

the exchange features for distributing assignments and feedback and the nbgrader independent features like editing an assignment and looking at feedback as the student, which does not have to interfere with nbgrader directly.
This process differentiation can also be found in the folder structure that nbgrader uses, which will be looked at next.

3.2.1 Main process folder structure

The main steps of the nbgrader process, which are creating, generating, autograding, manual grading assignments, and generating feedback, have a folder representation in the nbgrader framework as can be seen in Table 3.5.

Folder	Meaning and Content
source/[1]	created raw assignments
release/[1]	student version of the assignments
submitted/[2]	edited and submitted version by students
autograded/[2]	executed and automatically graded assignments
feedback/[2]	html feedback files with points and errors

[1] the folder structure is {nbgrader_step}/{assignment_id}/{files}
[2] the folder structure is {nbgrader_step}/{student_id}/{assignment_id}/{files}

Table 3.5: nbgrader steps from [nbge] used as a table

The *source* folder contains all raw assignments, which might contain hidden tests or solutions. Assignments are created in this folder and will be called the source version *source*

release

of an assignment from now on.
In the *release* folder, the student version of an assignment without hidden tests and solutions can be found after generating an assignment with nbgrader as mentioned in section 3.1.

submitted and autograded

Next in the main process is autograding. It uses the files from the *submitted* folder, where each student's final submissions lie, creating an autograded assignment in the *autograded* folder. The manual grading step uses these files to show the instructor the autograding results, so the instructor can change the grades and add comments.

feedback

In the last step, the assignment from the *autograded* folder is used to create HTML feedback files for each notebook, which are saved in the *feedback* folder.

inner folder structure

The source and release folder do not store student-specific assignments but a general version. Therefore, they do not have an additional folder in between for each student. However, there is a student folder in between for the submitted, autograded, and feedback folders. This folder structure allows for quickly going through the folder structure to find a specific assignment for a student. For example, Peter's feedback after the generation of feedback for assignment "Ps1" would lie in *feedback/Peter/Ps1/{files}*, while the source version of that assignment would lie in *source/Ps1/{files}*.

3.2.2 Exchange service

general information

For all distribution tasks like publishing, fetching, submitting, and collecting assignments or publishing and fetching feedback, nbgrader provides an exchange service for Unix multi-user systems. A shared exchange folder is used to allow both students and instructors to share files. The Table 3.6 shows the structure of that folder.

Folder	Meaning and Content
outbound/{assignment}	released version of assignments
inbound/{tag}[1]	edited and submitted version by students
feedback/{encoded tag}[2]	released feedback for the students

[1] tag: {student_id}+{assignement_name}+{timestamp}+{random_string}
[2] a hash of the student name, assignment name, notebook name and submission time

Table 3.6: nbgrader steps from [nbgd] used as a table

outbound folder

The *outbound* folder contains all assignments released to the students. The students fetch their assignments from here.

inbound folder

After editing, the students submit to the *inbound* folder, using a unique name for the folder containing all the submission files. The folder name consists of the student identifier, the assignment name, the timestamp of the submission, and a random string to identify each submission uniquely.
When the instructor collects the assignments from this folder, they take the newest submission within the submission time window for each student and copy it to the *submitted folder*, where it is processed as mentioned in subsection 3.2.1.

feedback folder

After the feedback generation of the main process, the feedback can be published to the *feedback* folder of the exchange, where each feedback file for a notebook is stored with a hash as the file name. Only the student who has submitted the assignment can know the hash for their feedback files.

Summary The workflow of nbgrader consists of five main steps: creating, generating, autograding, manually grading assignments, and generating feedback. These steps can change the state of the assignment, and the folder structure reflects this state. For distribution purposes, a public exchange folder can be used to release assignments to students and for students to submit their solutions. This exchange folder uses an inbound, an outbound, and a feedback folder to accomplish this.

3.3 User interface

The nbgrader can be used through the command line and a Jupyter client like Jupyter Notebook or JupyterLab. Both methods of interacting with nbgrader will be described.

nbgrader command line

The command line application of nbgrader consists of a single command called *nbgrader*, which uses different sub-commands and options for executing all commands of the workflow described in section 3.2. A list of all sub-commands and their options can be found in the nbgrader documentation [nbgb]. For example, the command for generating the assignment "Assignment2" is *nbgrader generate_assignment Assignment2*.

The following paragraphs describe the different graphical interfaces used by nbgrader. Each interface is provided by a *JupyterFrontEndPlugin* [jupb], extending the Jupyter client.

Validate Assignment Extension The *validate_assignment* extension adds a button for validating the nbgrader metadata of a notebook in the Notebook document editor of the Jupyter client. This button can be seen in Figure 3.3 in the menubar at the top with the label "Validate".

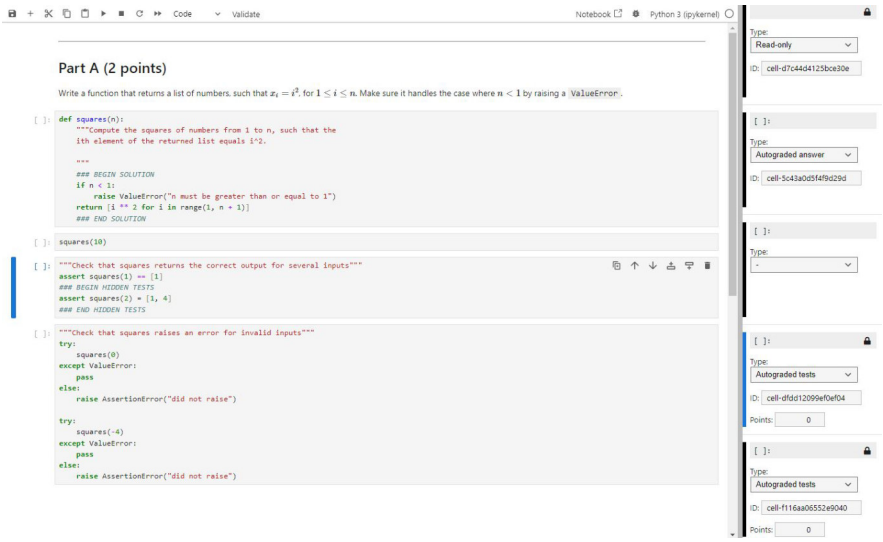


Figure 3.3: Notebook document editor with validate button and sidebar for metadata

Upon pressing the button, the notebook is validated, and a response dialog will be displayed.

Create Assignment Extension The *create_assignment* extension adds the right sidebar visible in Figure 3.3. It is used to set the metadata of the cells, including the purpose of the cell through nbgrader cell type, the grade identifier, and the points for the cell. The grade identifier is only available when a nbgrader cell type is set, and points are only usable for cells which have the nbgrader cell type *Autograded tests*, *Manually graded task* and *Manually graded answer*.

Course List Extension The *course_list* extension provides a window in which all courses an instructor is part of are shown and allows them to use the *formgrader* of that course.

managing assignments

Formgrader Extension The *formgrader* extension provides an instructor with one window with multiple screens. The manage assignments screen seen in Figure 3.4 displays the state of an assignment and starts various functions of the nbgrader workflow for an assignment.

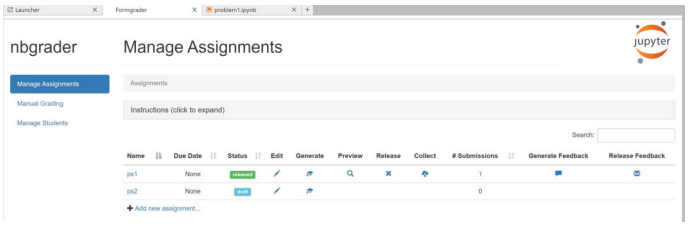


Figure 3.4: Manage assignments screen of the formgrader extension

manage submissions

Distribution functions like releasing and collecting assignments are also done through this screen when the exchange service is configured.

The manage submissions screen is reachable from the manage assignments screen, where student submissions can be autograded as seen in Figure 3.5.

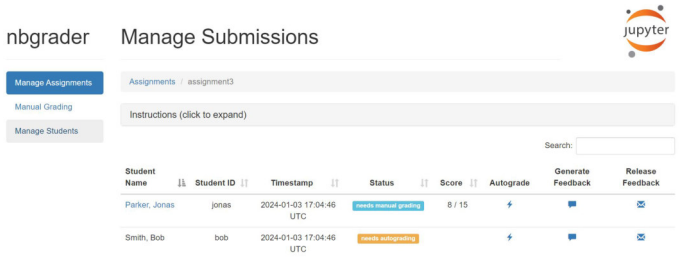


Figure 3.5: Manage submissions screen of the formgrader extension

manual grading submissions

The state of each student's submissions is indicated. Furthermore, the functionality to autograde, generate, and release feedback is given.

After autograding an assignment, it is visible in the list of submissions and ready for manual grading as seen in Figure 3.6.

manual grading

The student who submitted the assignment is hidden, and the distribution of the points to the nbgrader cell types is shown.

For the manual grading of the assignment, another screen is used. It can be seen in Figure 3.7.

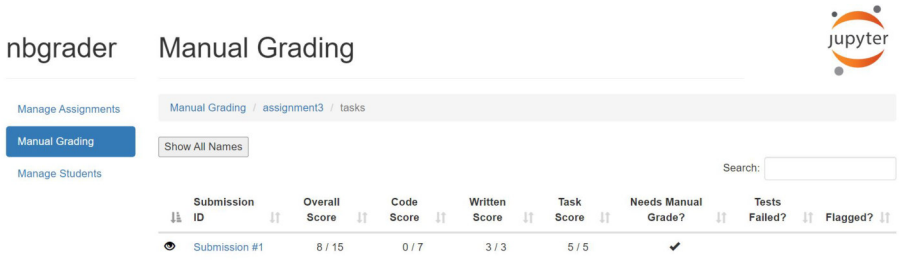


Figure 3.6: Screen with submissions ready for manual grading of the formgrader extension

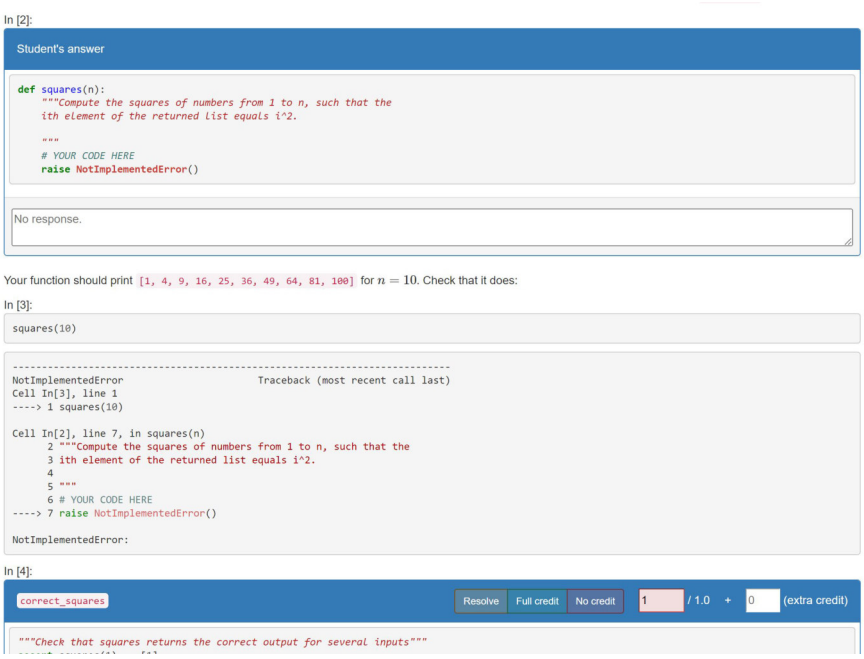


Figure 3.7: Screen for manual grading of a Notebook document in the formgrader extension

A box surrounds the graded cells of the submitted Notebook document. If the nbgrader cell type is *Autograded answer*, *Manually graded answer*, or *Manually graded task*, a text field for comments is provided. For cells with the nbgrader cell type *Autograded tests*, *Manually graded answer*, or *Manually graded task*, a text field for giving the student points can be used.

manage students

The manage assignments screen seen in Figure 3.8 is used to manage the students for all courses.

The students can be edited to show their names and email addresses. Additionally, it shows how many points they earned for all assignments.

Assignment List The *assignment_list* extension provides instructors and students with a window for seeing assignments and feedback in the exchange. The Figure 3.9 shows this.

Assignments are grouped in lists by their released, downloaded, or submitted status. Additionally, a dropdown for selecting the course is provided. A label shows to which course an assignment belongs. The exchange functions of fetching and submitting

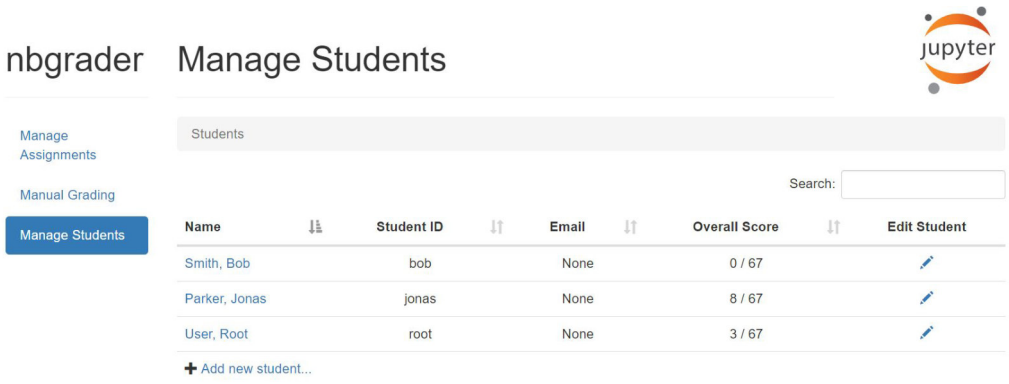


Figure 3.8: Manage Students screen of the formgrader extension

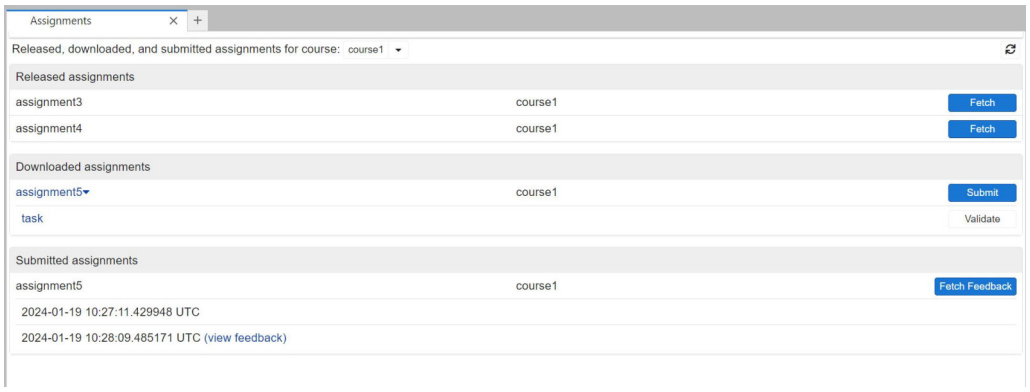


Figure 3.9: Assignment list

assignments and fetching feedback are implemented through buttons at the end of each row for an assignment.

Summary The graphical user interface of nbgrader is implemented through extensions to Jupyter clients. Five different extensions are used to implement the complete workflow of nbgrader. The *formgrader* extension covers nearly the entire nbgrader process except for the student actions, which are in the *assignment_list* extension. The *create_assignment* extension allows for setting the metadata, and the other two extensions cover switching between courses as an instructor and validating the Notebook document metadata.

3.4 Client Server Connection

client-side and server-side extensions

In section 3.3, the client-side extensions are described, and these extensions use server-sided extensions called server extensions [juph]. Server extensions extend the Jupyter Server [jupg] and start alongside a Jupyter Client like Jupyter Notebook or JupyterLab. The connection between the client-side and server-side extensions of nbgrader will be explained in this section, and more detailed information on the server extensions is given in section 3.5.

connection of extensions

For each of the five *JupyterFrontEndPlugins* except the *create_assignment* extension, there exists an equally named server extension that provides an *REST* API. The Figure 3.10 shows how the client-side extensions use the server-side extensions.

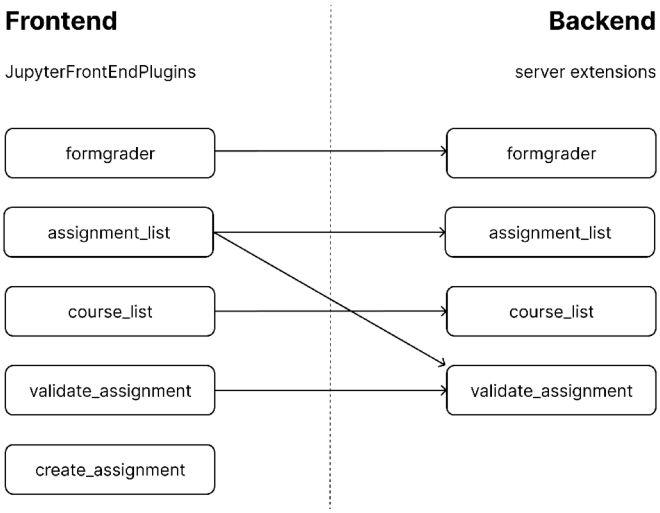


Figure 3.10: nbgrader client and server connection

For all *JupyterFrontEndPlugin* at most one corresponding server extension is used, except for the *assignment_list* client-side extension, which also uses the *validate_assignment* server-side extension.

3.5 Architecture

The architecture of nbgrader's mainly revolves around the server-side and command line application. Thus, the primary focus lies here, but the client-side components are also described in subsection 3.5.5. Components from now on refer to classes and functions. Figure 3.11 shows the command line application and server-side components in a layered architecture. Some components are grouped, for example, all server extensions. This architecture description was created by looking at the usage and import of components in the nbgrader GitHub repository [nbgf].

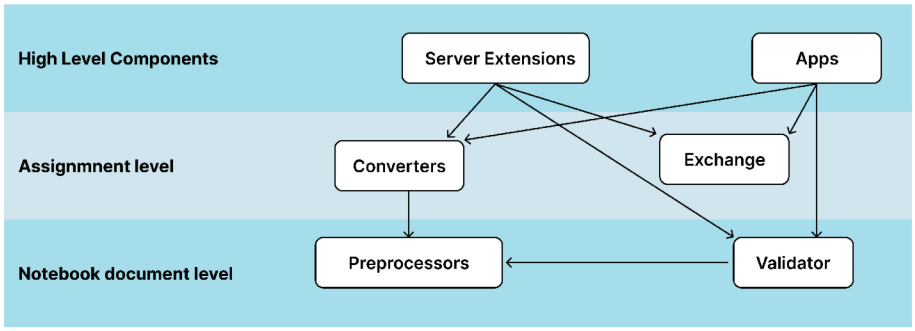


Figure 3.11: nbgrader architecture layers

architecture layers

On the Notebook document level, components, like the preprocessors or the validator, examine a single Notebook document at a time and transform or analyze its cells.

Components at the assignment level transform or move an assignment. The Converters, the exchange service, and several plugins belong to this layer.

The Jupyter server extensions that form the API and the applications are on the highest level of the architecture. The main nbgrader command line application uses the applications as sub-commands.
Because the layered architecture does not cover all features of the architecture, missing components and features will be shown in subsection 3.5.4.

3.5.1 Notebook Level Components

preprocessors

The scope of notebook level components is a single notebook. Preprocessors are used by nbgrader to allow for stepwise changes and information extraction on Notebook documents. A full list of the preprocessors and their usage is given in Appendix A. For example, the *ClearHiddenTests* preprocessor removes all lines within a hidden test block as mentioned in section 3.1. The *Preprocessor* class, which all preprocessors of nbgrader use as a base class, is part of the *nbconvert* library, which abstracts transformations to Jupyter Notebook documents [nbc]. This class has two important methods, *preprocess* for processing the entire Notebook document and *preprocess_cell* for processing every cell individually. All Notebook document transformations of nbgrader are implemented by using these two methods.

validator

The *Validator* class is used to validate the metadata of a Notebook document and uses several preprocessors after each other to accomplish this.
This principle of applying several preprocessors to transform or validate a Notebook document can be seen again when looking at the assignment level components.

3.5.2 Assignment Level Components

Assignment level components always scope their actions to one or more assignments and alter their contents while copying them.

converter

For example, the converter for autograding looks at the selected assignment or assignments and automatically grades them while coping them to the autograded folder as mentioned in subsection 3.2.1. Such a converter exists for every step of the main nbgrader process, except the manual grading. There are two additional converters. One is for generating the solution of an assignment from the source version, and another is for replacing only the autotest markers in an assignment.
All converters extend the *BaseConverter* class, which provides functions for copying an assignment and applying a list of preprocessors to a Notebook document. Converters overwrite this list of preprocessors to apply the preprocessors needed. In Appendix B, it is shown which preprocessors are used by which converter.

exchange

The exchange service consists of several classes. For every step of the exchange process, a class implements it by extending the *Exchange* class, which provides functions and configuration parameters to copy an assignment to and from the exchange folder. A class *ExchangeFactory* makes all other *Exchange* classes available through it.

3.5.3 High Level Components

server extensions

The high level components are used to give the user access to the assignment level components and the validator. The server extensions do this through their API endpoints, where through an endpoint, a *Converter* or class of the exchange service is created and used.

commandline applications

The command line application sub-commands do something similar. The sub-command defines the *Converter* or class of the exchange service to use and then forwards the parameters given to it to the configuration of the Converter or class of the exchange service.

workflow steps

All steps of the nbgrader workflow and validation of a Notebook document can be done through the server extension API or the command line application. However, the command line application provides additional sub-commands for other features.

zip-collect

These sub-commands include access to the *zip-collect* plugin used for moving zip files with student submissions to the submitted folder in the correct nbgrader folder structure.

export

Furthermore, a sub-command for exporting student grades to comma-separated list files (.csv) exists.

database

The command line application also allows changes to the database, but only to the assignment and student tables.

3.5.4 Other Features

configuration file

Other features of the nbgrader architecture include the configuration system, which uses the *traitlets* [tra] python package. This framework allows classes to have attributes with default values and, in nbgrader's case, to use a single configuration file called *nbgrader_config.py* and command line application options to change the attributes of classes.

CourseDirectory

The *CourseDirectory* class uses these attributes from the traitlets package to be configurable and hold the information of the current course, student, and assignment during the runtime of nbgrader. Several converters and the exchange service use this class.

Gradebook

The *Gradebook* class is the database interface and allows for interaction with the database, like creating assignments or reading the grades of a student's submission.

3.5.5 Client-side Components

Jupyter Notebook and JupyterLab compatibility

The client-side architecture consists of one index file in which the *JupyterFrontEndPlugins* are created and exported for usage as an extension for a Jupyter client.

Due to the usage of *JupyterFrontEndPlugins* for the extensions, they can also be used in Jupyter Notebook since version 7 of Jupyter Notebook. For some features, there is a case distinction between the usage of JupyterLab and Jupyter Notebook within the plugin to deal with differences not covered by the *JupyterFrontEndPlugin* class [nbgf].

formgrader iFrame

Requests to the server extensions are made using the Jupyter ServerConnection class, which communicates with the Jupyter Server.

The *formgrader* client-side extension is an exception because it uses an iFrame. The *formgrader* server-side extension provides and constructs the HTML, CSS, and JavaScript for the iFrame with a web server. Here, navigation and requests are either handled internally or via post requests.

learning management system integration

step learning curve

complexity

security

Disadvantages of nbgrader In [Lan19b], awkward submissions with the learning management system Blackboard [bla] are mentioned. Students downloaded and uploaded assignments from Blackboard, and then an instructor downloaded them again for usage with nbgrader. Despite nbgrader having distribution functions, this shows a lack or issues with integrating outside learning management systems.
A similar issue with uploading grades to a learning management system is mentioned in [EM23]. The authors also describe nbgrader as having a complex process and steep learning curve for instructors with configuring test cells and creating assignments.
The authors in [SvAY+20] call their solution, check50, light-weighted compared to nbgrader as nbgrader needs the complete setup of a course to autograde assignments with tests. Although it is not nbgrader's intended use case to automatically grade a single assignment as it is built for large-scale automatic assessment [JBB+19], it can be seen as a disadvantage of the framework not to allow flexibility.
Regarding security, González-Carrillo et al. compare UNcode to other frameworks, including nbgrader, and say that a secured sandbox is not provided or used by default and could interfere with the grading [Cri21].

3.6 Summary

The nbgrader is an automatic assessment system for grading assignments written in Notebook documents. It can be an extension to a Jupyter client or used as a command line application.
It has a unique metadata format for Notebook documents to add information to each notebook cell. Combined with markers in the cells, it allows for creating and generating a wide variety of assignments, which can be automatically graded and manually graded.
The clear workflow of nbgrader is reflected by its folder structure, where folders represent the state of an assignment. This system is supported by an additional exchange folder used to distribute assignments.
The nbgrader architecture is a mostly server-side extension to the Jupyter framework with many components. Most of these components are part of a three-layer architecture consisting of high level components made available to the user through a command line application and a REST API used by the client-side components. These client-side components are *JupyterFrontEndPlugins*, allowing compatibility with Jupyter Notebook and JupyterLab as the JupyterClient. The *formgrader* client-side and server-side extensions form an exception to the other extensions because they use an iFrame and a web server to display the content for the instructor's actions.
This chapter answers Research question 1.

Chapter 4 Requirements Study

study motivation

In Reseach Question 2, the problems instructors have when using nbgrader and their location in the nbgrader process should be determined. For this reason, a qualitative study is used to interview instructors who have used nbgrader. The results of this study then can be used to answer Reseach Question 3 and to define requirements for an improved version of nbgrader.
Threats to the validity of the study will are regarded in section 4.3.

4.1 Study Design

participation

The study is conducted as a voluntary interview, where the participants got the 13 questions in advance and may or may not look at these to prepare for the interview. The participation requirement is the usage of nbgrader as an instructor.

question design

For the questions, where applicable, a scale to measure the participant's agreement is used first to rate the question and then to ask follow-up questions to get an elaborate reasoning for this rating. The scale is constructed so that 1 always means complete disagreement or not easy to use, and 10 means complete agreement or flawless.

question categories

The questions are split into three sections. The first asks preliminary questions about how the participants use nbgrader, how easy they thought it was to learn, and how much the documentation helped them.
The following questions are about the nbgrader process. For each step of the main nbgrader process, as seen in Figure 3.2, it is asked how easy this step is to use. A follow-up question lets them justify this rating to obtain more information.
At last, the participants are asked more open questions regarding what problems they face while using nbgrader and which features they would like to have.

4.2 Results

The number of participants for the study is 1 **(n=1)**, which makes the results not representative.

Preliminary

How do you use nbgrader?
The possible answers were the command line, Jupyter Notebook, and JupyterLab. The participant's answer was the command line and Jupyter Notebook, but they also said they had used a self-written extension and scripts with nbgrader.

Did you find it easy to learn how to use nbgrader?
The participant gave it a score of 7 and said it was hard to learn to use nbgrader at the beginning, where they would have given this question a score of 2 or 3, but after understanding the folder structure, it was easier to learn. According to them, it also depends on the experience with Python and Jupyter Notebooks.
Is the documentation helpful?
The documentation was rated with a 6. It was also said that the documentation was primarily used to learn about the command line application options of nbgrader. Other channels and documentation, like tutorial videos or the Python and Jupyter Notebook documentation, were used for the initial learning.

Main Part

Is the assignment creation good and easy to use?
This part of nbgrader was rated with a 7. It was said that the assignment creation must be understood once to use it effectively. Learning the five nbgrader cell types was also mentioned as necessary, as well as that partial points are only possible by using multiple test cells. Negatively mentioned were that hidden test cells are visible to the student when empty and that each hidden test block's replacement code or comment cannot be changed individually.
Is the assignment generation good and easy to use?
A rating of 4 was given to the assignment generation. The participant justified this with hard-to-identify errors, a dialog window that is too small to show the error, and error messages or stack traces insufficient to find the source of the error in the Notebook document. The overall usability of this step was seen as dependent on whether an error occurred. If no errors occurred, the participant would rate this step with a score of 9.
Is the assignment publication good and easy to use?
The assignment publication by nbgrader was not used, instead assignments were inserted into a learning management system.
Is the assignment collection good and easy to use?
The participant did not use the assignment collection. Assignments were collected with a custom script instead.
Is the assignment autograding good and easy to use?
The participant gave the autograding a score of 4 because crashes during the autograding cause a submission to not be manually gradable. It was said that endless loops or recursions stopped executing the following cells in the submission's Notebook document. Another mentioned aspect was the possible side effects of having a variable declared before a test and then being declared inside the test. The participant's solution to this issue was resetting the Notebook document interpreter kernel. Positively mentioned was the possibility of autograding a submission again, but it would be helpful to be asked whether manual grades should be overwritten.
Is the assignment manual grading good and easy to use?
The manual grading rated with a 9 is described as easy to use because grades can be entered for each test.
Is the feedback generation good and easy to use?
The feedback generation was rated with a 9 and described as having no problems.

Is the feedback publication good and easy to use?
The participant did not use the feedback publication, and the generated feedback was published with additional scripts not provided by nbgrader.

Open and follow-up Questions

Are there other problems when using nbgrader?
Problems mentioned for this question are that when a student deletes a test cell, this test will always be graded with 0 points during autograding, although it might have passed. Cells with the same cell identifier cause the autograding process to not finish, leaving the Notebook document not manually gradable, like when endless loops or recursions occur. Feedback files for students show errors with their full trace back, but this is not very helpful for students just learning programming, as they might not understand what to do when a maximum recursion depth error or some other error occurs.

What features are missing in nbgrader?
According to the participant, nbgrader should provide more methods for error resolution, for example, when duplicate identifiers are used. These include the possibility of manual grading, even if the automatic grading fails. Cells moved in or removed from a student submission are not executed properly, and a resolution for this issue is missing. A proposed solution by the participant was to use the source version of an assignment and rebuild it with the student's submission to contain all cells.
Feedback should be exportable to JSON or Python format, but this feature should have a low priority, according to the participant.
There should be a feature for self assessment assignments, where the student submits their assignment and receives direct, automatically graded feedback without requiring an instructor to grade the assignment manually.
It was also stated that students sometimes make submissions that do not run or produce errors when all cells are run. For this reason, a feature to validate the student submission regarding execution should be added, with additional tests.
Group submissions should be possible. The submission should be made for multiple students.

error resolution
feedback
self assessment
submission validation
group submissions

4.2.1 Summary

In summary, the main problems lie with the generation and autograding of assignments, as their rating was 4, the lowest score of all questions. Error messages do not help to resolve errors occurring in these two process steps. The many situations where the autograding process does not finish are problematic, as it disallows manual grading. All other problems, except the feedback format, can also be attributed to the generation or autograding of assignments. Thus, errors in these two parts of the nbgrader process should be the focus of the requirements. This answers Reseach Question 2.
For all other parts of the process, either no score was given, or it was positive with a score higher or equal to 7.
The participant wants better error resolutions, more varied feedback formats, self-assessment functionality, and group submission functionality.
These results answer Research Question 2.

4.3 Threats to validity

not representative

Regarding threats to validity, as only one person participated in the study, the results are not representative and should not be seen as a complete overview of the problems of nbgrader.

participants used an older version of nbgrader

The study participation requirement was to have used nbgrader regardless of the version. This an issue for the results, as some problems named by the participant, like duplicate cell identifiers in the student submission or specific runtime errors when executing a student submission during autograding, are already resolved in the current version of nbgrader.

participants did not all features of nbgrader

Another problem is that participants did not necessarily use all the features of nbgrader. As seen in the study results, some questions are thus left unanswered. Furthermore, due to a possible lack of a fully holistic view of nbgrader's feature set by the participants, some issues they encountered, or features they thought were missing or wanted could have already been implemented. The study design did not foresee this and included answers and scores regardless, which could account for a skewed view of nbgrader.

These threats to the validity should be considered when examining and using the study results.

4.4 Requirements

process of defining requirements

The requirements consist of functional and non-functional requirements.

Functional requirements are defined by looking at the study's main problem areas, which are autograding and assignment generation. Autograding is considered more important because it is the core feature of an automatic assessment system and, thus, should be the core requirement with its connection to manual grading. A similar process of examining a problem and making a requirement is used for all other requirements.

Due to the bias in the study data and low participation, some requirements that do not directly follow the study results were added. These include Requirement 7, Requirement 9, Requirement 13 and most of the non-functional requirements in subsection 4.4.2.

4.4.1 Functional Requirements

The functional requirements will be divided into three categories: core requirements, which are essential to be implemented; requirements with normal priority; and optional requirements.

Core requirements

1. The autograding process must be changed.
    a) The student submission must contain all cell of the source version of that assignment before the execution of cells during autograding takes place.
    b) A incorrect cell order in a submission with regard to the student version is annotated or reversed.

   c) Ensure the autograding process continues, if cells produce any error during the notebook execution (including timeouts or maximal recursion depth errors).
   d) Ensure the autograding process continues, if cells with duplicate identifiers for grading (grade ids) exist in the metadata of the cells in the submission.

2. The manual grading process must always be possible.
   a) Grades for a student submission can always be entered and changed by the instructor.
   b) Manual grading must be possible, even when the autograding process doesn't finish without an error.

Requirements with normal priority

3. Test cells, which are empty, should not be visible to the student.
4. When an error in the nbgrader process is produced by a Notebook document cell, the error message should contain and show this cell's content.
5. Update validation tests for a student submission to test whether the metadata is correct and whether a notebook runs without errors.
6. Self assessment assignments should be possible.
   a) For a self assessment assignment, students submit their assignment and get instant feedback for it without feedback or grading by the instructor.
7. The screen for seeing all submissions for an assignment should afford autograding all submissions for that assignment with one action.
8. When autograding an assignment again, give the option to overwrite or not to overwrite the manual grading.

Optional Requirements

9. Add an export application, which allows saving the grades for each student and assignment in a JSON file.
10. Allow for group submissions.
    a) Groups are a collection of students where every student of the group can see and change the submissions of all group members.
    b) Groups are formed per assignment and not per course.
11. (addition to Requirement 1d) There should be an indicator for cells with duplicate identifiers.
12. (addition to Requirement 5) Add Smoke tests that execute additional tests that go beyond metadata validation and running all cells.
    a) The exact tests are invisible to the student.
13. (addition to Requirement 7) Grade multiple notebooks simultaneously.

4.4.2 Non-functional Requirements

14. Usability
    a) Instructors can easily find where errors were produced.
15. Maintainability
    a) Any changes should be well documented, in a way that other developers can easily understand the code.
    b) Configurable parameters use the same framework as other components.
16. Portability
    a) The solution should work on JupyterLab and on Jupyter Notebook.
17. Compatibility
    a) Notebooks written in the unchanged version of nbgrader must be compatible with the changed version of nbgrader

Now that requirements are defined for a changed version of nbgrader they are analyzed and implemented. These requirements are the answer to Reseach Question 3.

Chapter 5 Analysis and Design

This chapter analyzes the requirements by analyzing prerequisites for implementing each requirement and provides designs and changes to the architecture.

5.1 Analysis

The requirements will be analyzed in the same order as defined in the requirements.

5.1.1 Core requirements

prerequisites

Requirement 1a Having all cells in the student submissions that were in the source version requires three things: the source version of the assignment, the student submission, and an algorithm for identifying and adding the missing cells in the Notebook document. The source Notebook document can be obtained from the source folder as mentioned in subsection 3.2.1 or by using the database entry. The database entry only contains cells with a grade identifier. The submission Notebook document can be obtained from the submitted folder.

identifying cells

The algorithm for adding the missing cells must first identify which ones are missing. For that, cells need to be identifiable. Two identifiers exist: the grade identifier in the nbgrader metadata or the cell identifier used in the Notebook document metadata since version 4.5 and higher. Notebook document cell identifiers are added to every cell in a Notebook document, and the nbgrader grade identifier is only used for selected cells during the assignment creation. The nbgrader processes rely on the grade identifier, not the cell identifier. Changing this would require many changes or would introduce inconsistencies in the design. Another method would be looking at the cell's content, for example, the text in a cell, but this is no solution because any changes to a cell's content by the student would make identifying impossible or significantly harder.

existing solutions in nbgrader

An algorithm to add all task and grade cells not occurring in the submission already exists in the preprocessor *OverwriteCells*. A configuration parameter can enable its usage. It only adds cells that are in the database, which does not necessarily include all cells of the source version. Furthermore, task cells are appended at the end of the submission and not their original position. This algorithm also sometimes fails with an error. Thus, it would require changes to be reliably usable.

Requirement 1b For annotating or reversing changes to the cell order, it is required to have access to the source version and student submission and to identify cells similar to the analysis of Requirement 1a.

reversing the cell order

Changing the cell order in a Notebook document has effects on its execution. For example, having a variable defined in the first cell and using it in the second cell creates a dependency between the two cells. Now, if the cell order of this Notebook document is wrong and the two cells should be in the reverse order according to the source Notebook document, the execution can cause an error or produce a different result than the student had seen. Any implementation for reversing the cell order has to account for this.

Requirement 1c Continuing the autograding process, even if errors occur, requires handling the error or using an execution environment that can deal with such errors. The current version of nbgrader already handles all errors occurring during the execution, so no changes are required.

Requirement 1d This requirement concerns automatically grading submissions with duplicate grade identifiers. One option is to make all converters and preprocessors compatible with duplicate grade identifiers, making cells unidentifiable through this identifier. Another option would be to remove the duplicate identifiers already implemented in nbgrader's current version. Only the last cell with a grade identifier is used in that implementation. As this feature exists, no implementation is necessary.

Requirement 2a Grades for an assignment are saved in the database. Thus, changes are always possible by changing the database. The current implementation of nbgrader also allows for using text input fields for changing grades, as seen in Figure 3.7, when the autograding is completed before, and all grades cells are contained in the autograded version. This leaves cases open where directly changing the database or executing the autograding is not possible.

Requirement 2b In the current nbgrader version, manual grading is only possible after autograding because two prerequisites must be fulfilled. The prerequisites are the saved submission in the database and the automatically graded submission in the autograded folder. To make manual grading possible even when the autograding finishes with an error, there are two solutions to approach this problem. One would be keeping the workflow with both prerequisites for manual grading, copying the submission to the autograded folder, and writing the database entries. A second solution would be to change the prerequisites for manual grading, for example, by making the database entries during the manual grading or using the student submission directly from the submitted folder.

The first solution is more straightforward to integrate into the nbgrader process because it does not change the workflow. The current version of nbgrader offers an option not to execute cells. However, in case of incorrect metadata in the Notebook document, it fails and does not copy the assignment to the autograded folder or write the necessary information to the database. Thus, such a solution must be safe for metadata errors.

The second solution would make the manual grading step independent of the autograding step.

5.1.2 Requirements with normal priority

Requirement 3 The requirement demands that empty test cells be hidden from the student. Each Notebook document cell can be checked to determine whether it is a test cell or empty by examining its metadata and content.
A Cell in a Notebook document can be made invisible to a student by removing it from the Notebook document or hidden by altering the appearance of a Notebook document in the editor. Removing the cell has the benefit of working independently of the editor the student uses to do the assignment but requires adding the cell back to the Notebook document when it is graded.

Requirement 4 A prerequisite for adding the error-causing cell to the error message is having one cell responsible for the error. The API provides the information for the error message. Thus, the API must make the error and the cell information available. In the current version, nbgrader handles errors caused by preprocessors within converters by catching and then raising them again or saving the information and then raising a different error after all Notebook documents of the assignment were run. The information about the cell causing the error is not given in the error or not used.

Requirement 5 Validation tests check whether the metadata is correct and whether all cells of a Notebook document can be executed without an error. The current nbgrader version offers a validate button that, by default, checks the metadata of the Notebook document but can also run all cells to see if errors occur during the execution of cells in the Notebook document when a configuration option is enabled. Thus, this requirement can be solved by changing the configuration.

Requirement 6 Self assessments are assignments where students receive instant feedback when they submit their solutions without requiring an instructor to grade them. Therefore, self assessments can only be automatically graded. *distribution and processing*
A prerequisite is that instructors can publish the self assessment assignments, and students can access them. The exchange folder of nbgrader already provides this functionality for regular assignments. Releasing and fetching assignments through a graphical interface could also be adapted for self assessments. *autograding*
Instant feedback for students requires a new workflow because the nbgrader process relies on the instructor to grade the assignments and generate feedback. For self assessments, these steps must be combined and usable by a student. Thus, a new converter or changes to the existing converters are required to grade a self assessment assignment. *data access*
The *autograding* converter requires access to the database or source version of an assignment to save the grades and add hidden tests before the grading. The *generate_feedback* converter needs to access the database to use the grades for the feedback generation. For the self assessment, students should not have full access to the database or source version of an assignment in the nbgrader folder structure because this would introduce security issues.

Requirement 7 A prerequisite for autograding all submissions for an assignment is the assignment name or identifier and having a method to do the autograding. The nbgrader provides a command *nbgrader autograde AssignmentName* for doing the autograding, and on the screen where the submissions are listed, the assignment name is known. Thus, the prerequisites are fulfilled and can be used by an implementation for this requirement.

Requirement 8 Displaying a dialogue when autograding a student submission for a second time requires knowledge about the existence of a previous grading. This can be done by looking into the database or the autograded folder as discussed in subsection 3.2.1. This information must be available to the template and its connected JavaScript file to display the page in the *fromgrader* extension. Then, the dialog can show the different actions that trigger the desired mode of overwriting.

5.1.3 Optional Requirements

Requirement 9 Nbgrader already allows exporting students and their grades for assignments to comma-separated value files (.csv). The database is accessed to obtain this information, and the same technique can be used to export to the JSON format. It would be possible to export to a comma-separated value file and then use that file to create the JSON, but using the information of the database directly is more efficient.

groups

Requirement 10 The groups required for group submissions need to be saved to allow for checking a student's membership in a group. Options for this are Unix user groups or a database that holds a table for the groups.

slicing the nbgrader process

For the main nbgrader process, there are several ways to split the assignment from the group to the individual students, as each student should get their feedback individually. Making the submission for each student during the group submissions would mean that an instructor would grade the same assignment for each group member although it is the same assignment. This would be easier to integrate into the nbgrader process but comes at the expense of effort for the instructor. An alternative is to split the group again at the feedback step to give each student the feedback. Then, during the feedback generation, the converter needs to know whether an assignment is a group assignment and who belongs to the group.

Requirement 11 Identifying cells with a duplicate grade identifier in a Notebook document is necessary to show the student the visual hint. The grade identifier can be found in a cell's metadata explained in section 3.1. This information is obtainable both on the client and server sides. When comparing both options, the client side is the better choice as any change to the Notebook document can be directly registered and used to update the visualization of the Notebook document. Transferring this information on every change to the client side would be necessary for a server side solution.

Requirement 12 Adding smoke tests, which validate the student's submission before submitting, requires the student's client to access these smoke tests, which results in a similar issue as having access to the source version of an assignment for the self-assessment in the analysis of Requirement 6. During the validation, these smoke tests must be executed to test for errors, for example, by adding the smoke test cells to the Notebook document.

Requirement 13 Grading multiple Notebook documents simultaneously can be done by creating a list of Notebook documents that need to be graded and starting a new thread for each autograding execution. Important here is that each execution has a different *Jupyter KernelManager* for each Notebook document so that the variable names stored by the Kernel cannot be accessed or overwritten by a different Notebook document's execution.

5.1.4 Summary

The analysis shows that changes to the autograding and manual grading process need changes to the process and preprocessors of nbgrader. Self assessments also require a new converter or process for evaluating the self assessment assignments.
Some requirements like Requirement 1c, Requirement 1d, and Requirement 5 do not need any changes as they are already fulfilled or can be solved by changing the configuration.
Group submissions offer different implementations, depending on how the groups are implemented and how the process changes after a student submits for a group.
The other requirements need changes or additions of preprocessors or client-side extensions.

5.2 Design

This section covers the changes to the architecture and workflow of nbgrader to implement the analyzed requirements.

Changes to the Autograding process The autograding process needs changes to its design to have a separation of concerns. In the analysis of Requirement 1, it is said that a preprocessor for adding missing cells already exists. This preprocessor's primary concern is validating that a student did not change the metadata of a cell, thus not primarily for adding missing cells. A new preprocessor *AddMissingCells* is used for that reason.

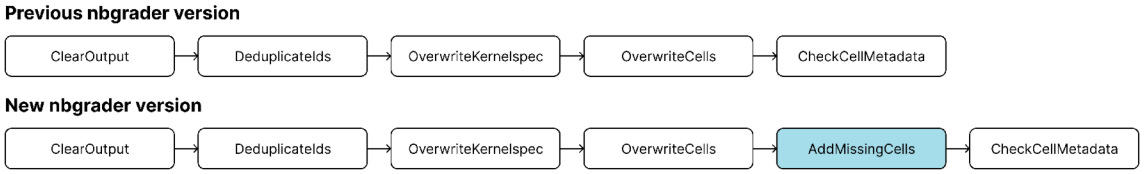


Figure 5.1: Autograding converter change (Sanitization phase)

Figure 5.1 shows the new preprocessor in the autograding converter's sanitization phase. It is used after the *OverwriteCells* preprocessor is executed. Additionally,

annotations regarding the cell order are made in the new preprocessor *AddMissingCells*. A design for these annotations is to add a text in the cells that are in the wrong position in regard to the source version of the Notebook document.

Changes to Manual grading A design for changing the grades as mentioned in the analysis of Requirement 2a is to use a command line application.
The other changes to the manual grading process require a redesign of the grading process. A method mentioned in the analysis of Requirement 2b is bypassing the autograding and making changes to the database and folder structure to have all prerequisites for manual grading fulfilled. An option for the *autograde* command can be used for this.

removing cells

hiding cells

Hiding empty test cells The analysis of Requirement 3 gives two solutions to hide tests from students. An architectural design for removing the cells from the Notebook document would be to add or change a preprocessor to remove the empty test cells while generating the student version of an assignment in the *GenerateAssignment* converter.
A *JupyterFrontEndPlugin* can be used to hide cells, which alters the appearance of a Notebook document in a Jupyter client editor.

Self assessment Adding self assessment requires a new workflow because the instructor does not act after releasing the self assessment assignments to the students.

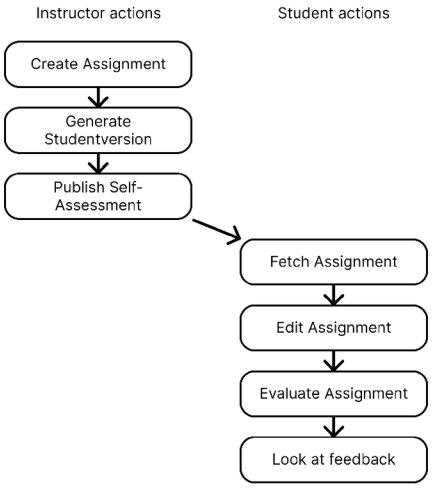


Figure 5.2: Self assessment workflow

The design for the workflow can be seen in Figure 5.2. The First three steps are similar to releasing a regular assignment, but instead of submitting the assignment, the student evaluates it. The converter for the evaluation combines the process of autograding and generating feedback.

data access

Two possible design approaches for accessing the required data exist. One option is to use the exchange to provide the student with the information. Another would be to use an API endpoint to let the student receive the required data.

Furthermore, new client-side and server-side extensions that allow the student to use self-assessments and evaluations in the graphical user interface of nbgrader are part of the design.

Autograde all submissions The analysis shows that all prerequisites are fulfilled. A possible design is a button and additions to the *formgrader* server-side extensions to invoke the nbgrader command for autograding all submissions for an assignment.

Group Submissions The design for the group submissions is to use Unix user groups to form the groups so that no changes to the database are needed. The standard autograding process can be used with a student which represents the entire group. All submissions of the group members are submitted for this student representing the entire group.

The following chapter will show the implementation of these designs in detail.

Chapter 6 Implementation

The implementation of the analysis and design results will be described next, with pictures and details on the implementation.

6.1 Changes to the Autograding Process

The changes to the autograding process are made as proposed in the design section for it.

algorithm for adding missing cells

As mentioned in the analysis of Requirement 1a, a previous version of adding missing cells existed and is now enabled by default. In addition to that, a new algorithm was added. This algorithm looks at all cells of the source version and adds all those back with the grade identifier in its metadata. This allows adding other missing cells that are not solution or task cells. Furthermore, task cells are added after the preceding cell in the source Notebook document rather than at the end. The resulting cell order is more similar to the source notebook. An example can be seen in Figure 6.1.

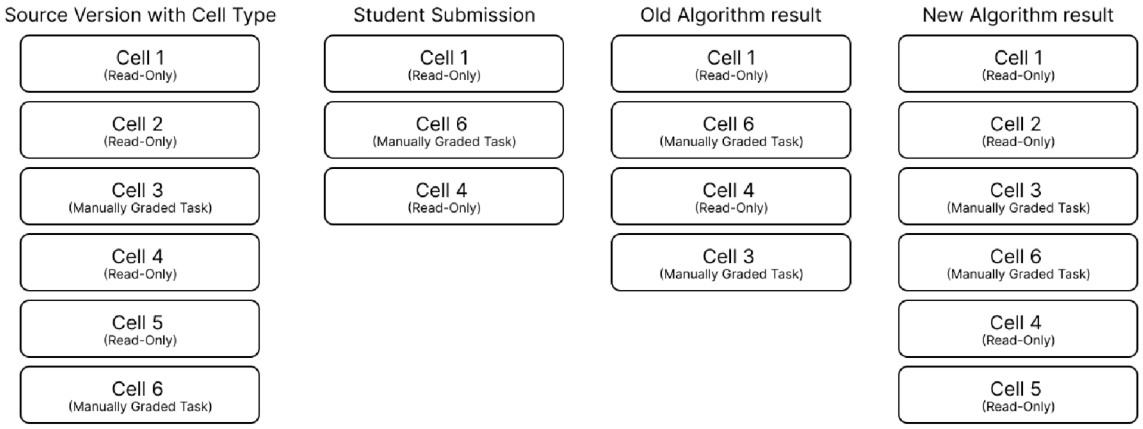


Figure 6.1: Adding of missing cells example

The student submission removes three cells and changes the order of cells 6 and 4. In the old algorithm, only the missing task cell 3 was added at the end. All missing cells are added with the new algorithm, and the order is always ascending except for cell 6.

The new algorithm can only detect a cell with a grade identifier. Other cells are not added, as their position and occurrence cannot be determined as stated in the analysis.

marking incorrect cell order

The analysis of Requirement 1b names potential problems of reversing the order of cells. Therefore, the implementation only marks wrongly positioned cells that use a

grade identifier. The implementation is within the *AddMissingCells* preprocessor to avoid loading the Notebook document of the source version multiple times.

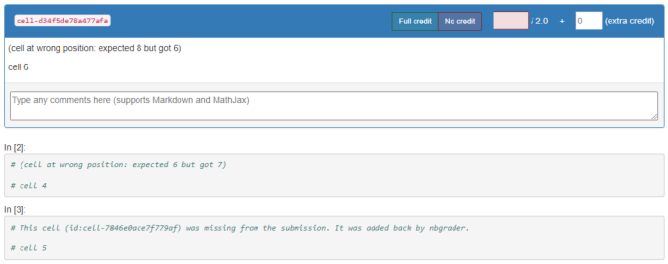


Figure 6.2: Annotation of added missing cells and cells at the wrong position

The annotation for a cell at a wrong position can be seen in Figure 6.2. The solution is as proposed in the design for this requirement.

other autograding requirements

Both analyses of Requirement 1c and Requirement 1d show that the requirement is already fulfilled in the newest version of nbgrader. Therefore, no changes are made.

6.2 Changes to Manual Grading Process

always changing grades

An additional sub-command to the nbgrader command line application's sub-command *db* is added to allow for always changeable grades. In the new version of nbgrader the instructor can use the commands shown in the Figure 6.3 to list the grades for the cells in a specified Notebook document in a student submission.

```
# nbgrader db submission list assignment3 tasks bob
cell-24ae395a7913f479: 7.0/7.0 manual: 7.0, auto: 0.0
cell-6270d7a851bedf36: 3.0/3.0 manual: 3.0, auto: 0.0
cell-753683422f785b46: 5.0/5.0 manual: 5.0, auto: 0.0
# nbgrader db submission update assignment3 tasks bob cell-24ae395a7913f479 5
In the submission of bob for assignment3/tasks changed the score of cell cell-24ae395a7913f479 to 5.0
# nbgrader db submission list assignment3 tasks bob
cell-24ae395a7913f479: 5.0/7.0 manual: 5.0, auto: 0.0
cell-6270d7a851bedf36: 3.0/3.0 manual: 3.0, auto: 0.0
cell-753683422f785b46: 5.0/5.0 manual: 5.0, auto: 0.0
```

Figure 6.3: Command for listing and changing the grades for a cell in a student submission

The second command shows how to change the grade of the cell with its grade identifier to a score of 5. The additional parameter bob is the student identifier, assignment3 is the name of the assignment and tasks is the name of the Notebook document.

bypassing autograding

The solution to keep the workflow is implemented as making the manual grading independent of the autograding step is considered to be not beneficial when manual grading is only used at one specific location in the nbgrader workflow. To make errors in the metadata handleable, a new preprocessor that removes the entire nbgrader metadata is added, which can be activated with the autograde command and the option *–without-metadata*. The *SaveAutoGrades* preprocessor is updated to ensure that all grades are added to the database when the metadata of all cells is removed. It now has a boolean configuration option, which, when set to true, adds an automatic grade with a score of 0 for each autograded test and demands manual grading for all of those cells. This is sufficient to use the before-mentioned command to change grades and thus accomplishes the goal of making manual grading always possible.

graphical user interface

To also enter the scores in the graphical user interface in Jupyter Notebook or JupyterLab, a copy of the source Notebook document cells which are graded is added to the manual grading screen. The score and comments can be entered through these cells.

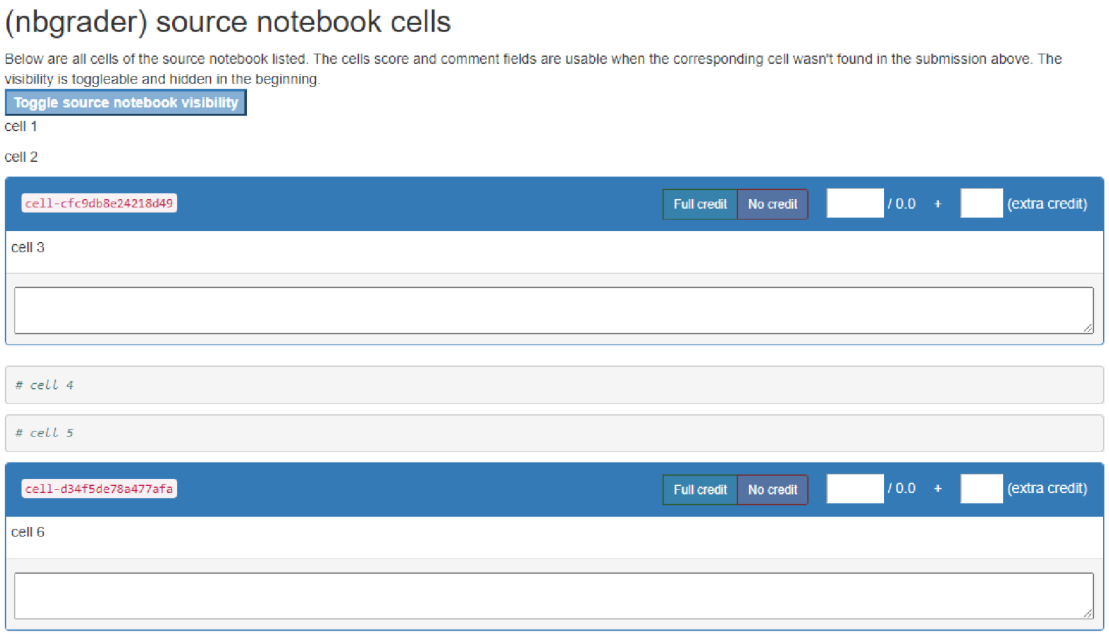


Figure 6.4: Manual grading source cell copy

In Figure 6.4, the additional cells of the source version and text boxes for assigning grades and adding comments for each gradable source cell can be seen. The text fields are only usable when the cell is not found in the student submission, and this is the case when the metadata for each cell is removed. Thus, the copies of the source version cells allow the user to assign grades.

6.3 Hiding empty test cells

The implementation of Requirement 3 is done with a preprocessor according to the analysis and design. This option is chosen because it is easy to integrate into the nbgrader framework and works even if students edit the Notebook document outside a Jupyter client using nbgrader. The new preprocessor *RemoveEmptyTests* is added to the generate assignment converter right after the *ClearHiddenTests* preprocessor. Therefore, the hidden tests are first replaced and then removed if the entire cell is empty.

6.4 Error dialog

The error dialog needs to be given the information about the cell causing the error through the API, as mentioned in the analysis of Requirement 4. For this purpose, a new Python exception called *NbGraderPreprocessingException* is added, with an attribute for the cell identifier and cell's content. The converters can specifically catch these exceptions and save the cell information with the notebook and assignment name in which the error occurred. The converter then raises and saves another

exception called *NbGraderConverterException* with necessary information. This information is converted to a JSON format for the API endpoints and can be used in the error message to display the cell content.

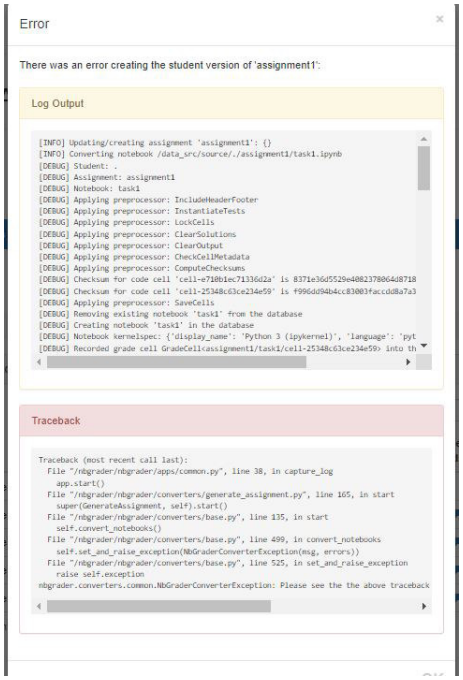


Figure 6.5: Error message (before version)

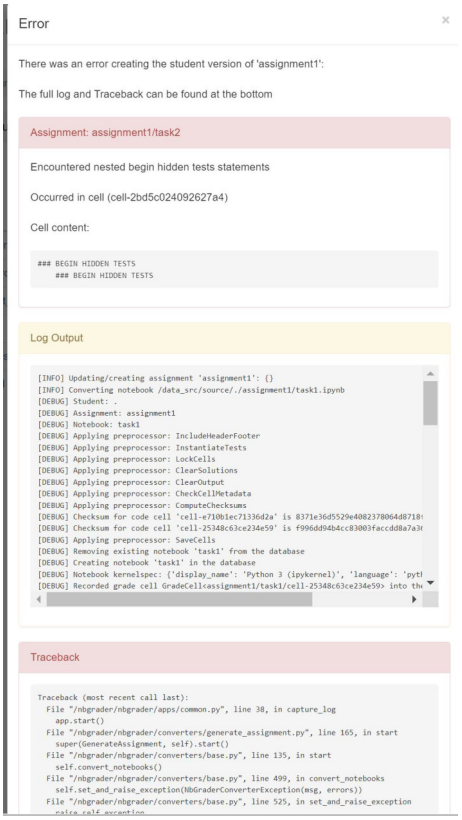


Figure 6.6: Error message (new version)

The previous error message can be seen in Figure 6.5, and the error message after the implementation can be seen in Figure 6.6. The new version uses an additional box to show the cell information, assignment name, and Notebook document name.

6.5 Validation tests

Validation tests in nbgrader already exist to check the cell metadata and can also be used to check whether a notebook's execution runs without an error. In the analysis, it is determined that changing a configuration parameter is sufficient to implement Requirement 5. This parameter's default value was changed.

6.6 Self-assessment

The self assessment is implemented according to the design. For the data access, the exchange is used where the student and source version of the self assessment are accessible to the student. When fetching the self assessment, the student version is used. The source version of the assignment is loaded for the evaluation to add missing cells and hidden tests to the Notebook document.

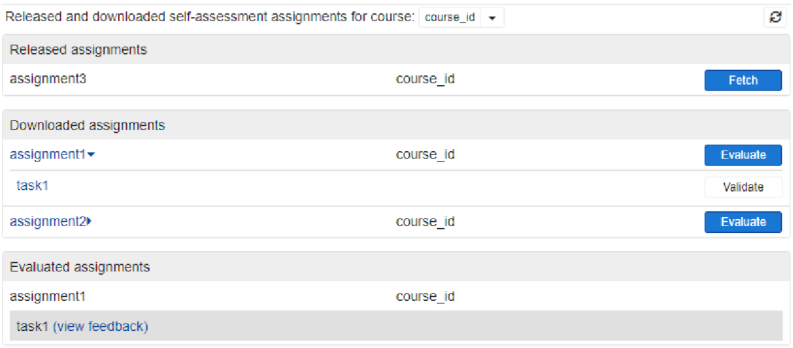


Figure 6.7: Self assessment assignments view

Sub commands, a client-side extension, and a server-side extension are used to let the student use the self assessment functions.
The client-side extension *self_assessment_list* can be seen in Figure 6.7. It shows all self assessments available to the student and can be used to fetch and evaluate the assignments. After the evaluation, the student can look at the feedback.
Releasing a self assignment is only possible through the command line with the command *nbgrader release_self_assessment AssignmentName*.
The added converter *EvaluateSelfAssessment* uses the same preprocessors as the converters for autograding and generating feedback. For some of these preprocessors, a configuration parameter is used to distinguish the usage in self assessments and regular assignments to avoid using the database and use otherwise provided data.

6.7 Autograde all button

A button to autograde all submissions for an assignment was added on the submissions screen, as described in the design for Requirement 8. The button uses a new API endpoint that executes the autograding and returns a JSON response with the result and possible errors displayed in a dialog.

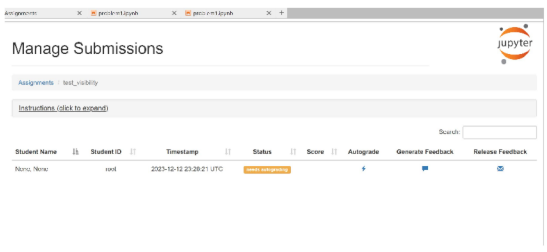


Figure 6.8: Submission screen (older version)

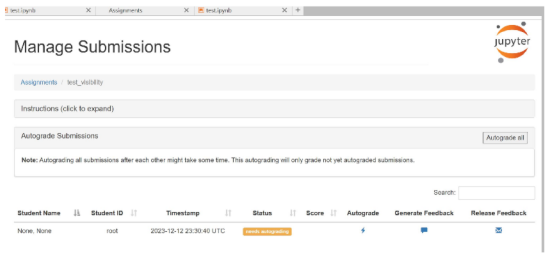


Figure 6.9: Submission screen (new version)

The addition of the button and its description text can be seen in Figure 6.9 and Figure 6.8 shows the screen in the older version.

6.8 Group Submissions

workflow

The proof of concept for group submissions works like the regular assignments. An instructor creates and generates the assignment and publishes it to the students. The

instructor needs to add all students of a group to the same Unix group. Then, the students can submit the assignment under the name of their group and the instructors. A prefix distinguishes submissions for a group and a single student. For example, a group named *group1* would submit under the name *group-group1*, which is treated by nbgrader like any other student. Then, the steps of collecting, autograding, and manual grading are made like for regular assignments. Then, at the generation of feedback step, the members of the Unix group are retrieved, and the feedback is created for each of them.

submitting and fetching

The submission can be made from any group member as they submit under the same name, and they can also look at the submission of their group member and download the last submission. Downloading the last submission is necessary to retrieve the feedback, because it has the timestamp of the submission, which is needed to find the feedback.

compatibility issues

Managing groups this way would require modifications to work on a non-Unix system. Additionally, groups are not created through nbgrader but through the Unix system, meaning a possible loss of control as changes to the groups might not be noticed. Future work can look at a solution that solves these issues, for example, using the JupyterHub roles.

6.9 Duplicate Cell Identifier Warnings

In Figure 6.10, warning and error messages can be seen for cells with duplicate grade identifiers in a Notebook document.

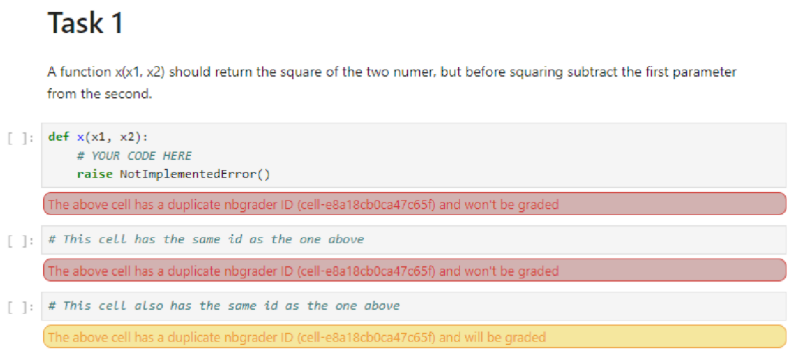


Figure 6.10: Duplicate grade identifier warnings

Red banners mean the cell will not be graded, and yellow means it will be graded. Only the last occurrence of a grade identifier will be graded in nbgrader. Thus, the banners follow the same principle. The text on the banner additionally gives information about the grade identifier, which is duplicated. This feature is implemented as a new *JupyterFrontendPlugin duplicate_id_indicator*, so it can be enabled for instructors and students. The plugin listens to state changes in the notebook and then adds the banners accordingly. These banners are shown and updated when the student saves their work or the first change after saving occurs. The second change will not be registered until the notebook is saved. This detection should be updated in future work.

6.10 Non-Functional Requirements

The non-functional Requirement 14 was implemented through the Requirement 4. Requirement 15 is implemented through comments on the newly implemented classes and functions. For the configuration, the *traitlets* package is used, which is already used in nbgrader. To make the solution work in Jupyter Notebook and JupyterLab, according to Requirement 16, *JupyterFrontEndPlugins* and command line applications are implemented. Requirement 17 is fulfilled because no changes to the nbgrader Notebook document metadata format are made, and the Notebooks documents of the previous version can be used.

6.11 Summary

changes to grading

All core requirements are implemented. Requirement 1a and Requirement 1b are solved with a new method for recreating a Notebook document based on the source version instead of only the cells saved in the database. During this, cells at the wrong position are marked.

The grading process was changed by allowing for the removal of nbgrader metadata in a Notebook document. Then, copies of the cells in the source version of an assignment can be used to grade the Notebook document, even when the nbgrader metadata is corrupted. Furthermore, a command to directly change the score for a cell in the student's submission is added.

Error messages show the cell causing the error, the assignment name, and the Notebook document name.

Validation tests now also execute all cells in a Notebook document by default.

new features

A button for autograding all submissions for an assignment and visual hints for finding cells with duplicate identifiers are added.

Two new features are introduced to nbgrader. Self assessment assignments are added, where students can automatically grade assignments and receive instant feedback. The other feature is a proof of concept for group submission, where students submit their assignment for a group, and all group members receive feedback, while instructors only grade the latest submission of the group.

Requirement	1	2	3	4	5	6	7	8	9	10	11	12	13
Implemented	x	x	x	x	x	x	x			x	x		
Not Implemented								x	x			x	x

Table 6.1: Implemented requirements (functional)

Requirement	14	15	16	17
Implemented	x	x	x	x
Not Implemented				

Table 6.2: Implemented requirements (non-functional)

The Table 6.1 and Table 6.2 show that all functional and non-functional requirements are implemented except Requirement 8, Requirement 9, Requirement 12 and Requirement 13, which are not implemented due to a higher prioritization of the other requirements. All non-functional requirements are implemented.

Chapter 7 Validation Study

This study aims to determine whether the changes improved the nbgrader instructor user experience as stated in Research Question 4, and where it improved. Additionally, it is used to find remaining or not mentioned issues while using nbgrader. The questions and results are presented in section 7.2.

7.1 Study Design

The study consists of three sections, an introduction, a main part with multiple tasks in two versions of nbgrader and a comparison, and a closing part where participants are asked to rate several statements.

nbgrader versions

The first version of nbgrader used in the study is the main branch of version 0.9.1 [nbgi] with the default configuration. The second version is the modified version of nbgrader with the implementations of chapter 6 with its default configuration.

participation requirements

The participation requirements were having experience with programming in general.

7.1.1 Introduction part

introduction part

The first part asks the participants about their experience with Jupyter Notebook documents and with nbgrader. Followed by a guided tour through the nbgrader process from assignment creation to looking at feedback as a student in the changed version of nbgrader. Participants are given stepwise instructions like "Create a new assignment called assignment1". The participants are asked to describe their actions and thoughts similarly to the think aloud method [VSBS+94] and to name anything they find unintuitive.

7.1.2 Main part

main part

The main part consists of two tasks done in two versions of nbgrader and a comparison of two pictures. After each task, the participants are asked to rate statements on a scale of 1 to 10 where 1 means full disagreement and 10 means full approval. Tasks are performed first in the unchanged and are repeated in the changed version of nbgrader with the same files. Tasks also have a time limit of 5 minutes, which was not communicated to the participants. In case the time limit is exceeded, task-solving is stopped. After the picture comparison, the participants are asked a question.

first task

In the first task, the participants are asked to generate an assignment, which causes several errors. The task is to find and solve these problems in the Notebook documents of the assignment and to generate the student version successfully. There are

3 errors in 3 Notebook documents and one additional Notebook document without an error. The types of errors are a missing "### END HIDDEN TESTS" statement, a cell configured to use the wrong Notebook document cell type for a specific nbgrader cell type, and cells that use the same grade identifier.

second task

In the second task, participants are asked to grade two submissions with full points. The first submission used incorrect nbgrader metadata, and all test cells were removed in the second submission. Depending on the version, errors can be solved differently.

If participants are stuck on a task, further hints or instructions are given to help them. Due to the scope of the study, no clear protocol or method is used regarding how these hints are presented nor when to intervene. Consequences of this will be discussed in paragraph 7.2.

student assignment comparision

For the comparison of student assignments, a picture is used. The picture can be seen in Figure 7.1.



Figure 7.1: Picture for Student Assignments Comparison

The figure shows two student versions, one with the test cells and another with test cells removed. The participants are asked which version they would prefer as students.

showing of new features

Afterward, participants are shown how to use the self assessment assignments and group submissions and are asked to rate several statements for these features. They do not perform any actions themselves.

Lastly, participants are asked to pick a version of nbgrader they prefer and comment on using and seeing it.

7.1.3 Closing part

In the closing part of the study, participants are asked to rate the system according to the system usability scale statements [Bro96]. The statements are given one by one and are answered with a score from 1 to 10, where 1 is full disagreement and 10 full agreement.

7.2 Results

The study had 6 participants (**n=6**), and the results will be presented in the order of the study, and then additional participant comments will be shown. All participants choose to speak German. Therefore, all quotes are translations.

Introduction part Figure 7.2 and Figure 7.3 show the results of the introductory questions.

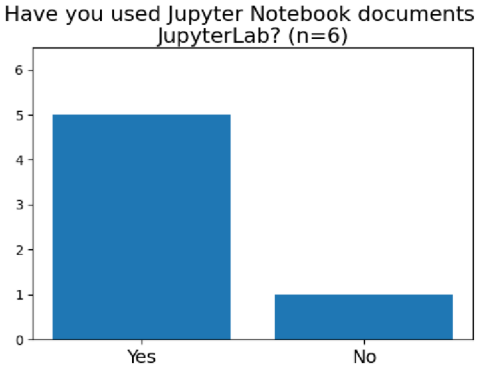


Figure 7.2: JupyterLab usage

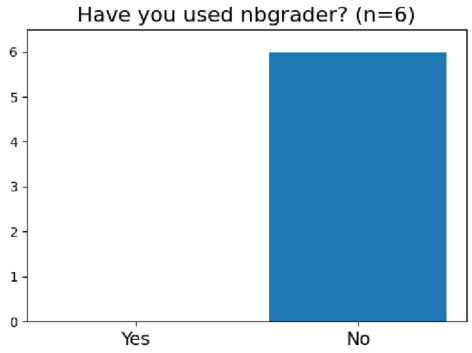


Figure 7.3: nbgrader usage

Only one of the six participants has not used Notebook documents in JupyterLab, and none have used nbgrader. This means that the study results have to be seen in the context of users without prior experience, which, on the one hand, has the advantage that the results have no bias from previous usages of nbgrader. On the other hand, this also means comments made by the participants cannot speak to the effect any change would have on the in-classroom nbgrader experience.

First task

Main part Three things were measured for the first task, where the participants had to find errors in an assignment. The results can be seen in Figure 7.4 and Figure 7.5.

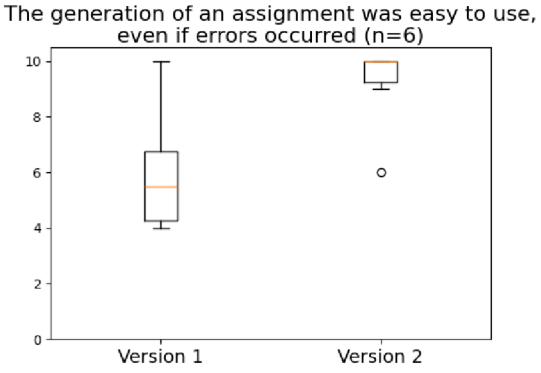


Figure 7.4: Rating of Generation Process

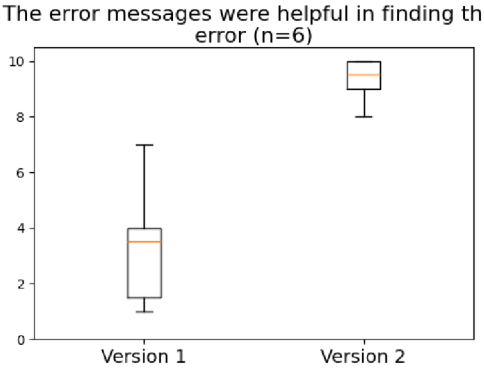


Figure 7.5: Rating of Generation Error Messages

Both box plots show that the second version has significantly better ratings for the usability of the generation process and the helpfulness of error messages. The median for the second version, for both statements, is higher than 9, meaning nearly full approval for the easy usability and helpfulness of error messages in the second version. When seeing the error dialog of the second version, one participant even said,

"Oh, that's much better". The number of found errors in both versions was counted and can be seen in Figure 7.6.

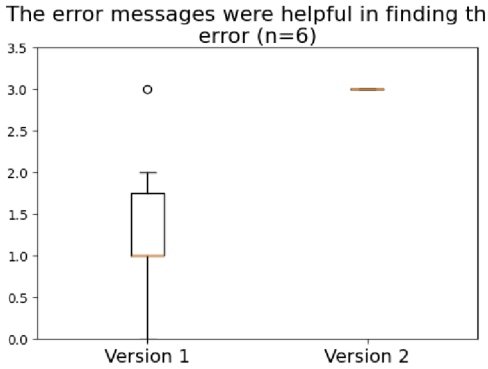


Figure 7.6: Number of Errors found by Participants

It shows that in the first version, most participants found one or two errors, while in the second version, all participants found all three errors and could generate the assignment. This indicates an improvement in the generation process.

Second task

In the second task, ratings for the usability of the autograding, manual grading process, and the helpfulness of error messages for the autograding process were measured. These ratings can be seen in Figure 7.7, Figure 7.8 and Figure 7.9.

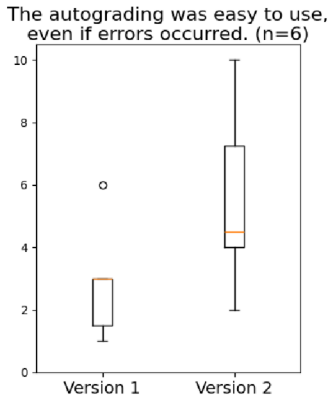


Figure 7.7: Autograding Process Rating

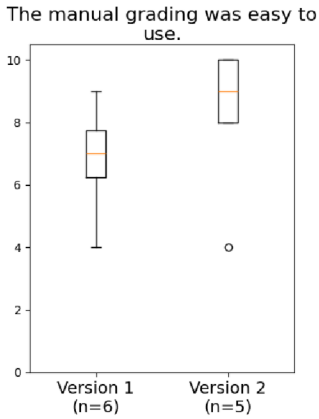


Figure 7.8: Manual Grading Process Rating

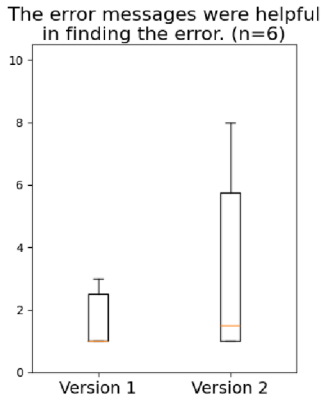


Figure 7.9: Autograding Error Messages Rating

From the results, it can be seen that both the autograding and the manual grading process are rated higher in the median. For the autograding process, the second version's values are spread more, which could indicate conflicting opinions regarding the autograding process of the second version. For the manual grading, the opposite is true. The first version's rating is high, with a median of 7, but spread from 4 to 9. In the second version, most ratings are in a range from 9 to 10, showing a higher consensus among the participants regarding this rating. The median for the helpfulness of error messages for the autograding process only improved slightly, showing that the changes made here were ineffective.

After these ratings, participants were also shown how to use the autograding without metadata command and the source cell copies in the manual grading screen of a

Notebook document. The source cell copies were remarked positively, but the button to show them was seen as being labeled incorrectly or not shown noticeably enough. The command was criticized. Some participants wanted a button, and others said the command should afford to be copied directly to the command line instead of manually entering the student identifier.

For the manual grading process of the second version, one rating was removed from the results because it was answered twice after a comment by the study conductor. When comparing the two pictures of the student version of an assignment, all participants preferred the second version without the empty cells for hidden tests.

student version comparison

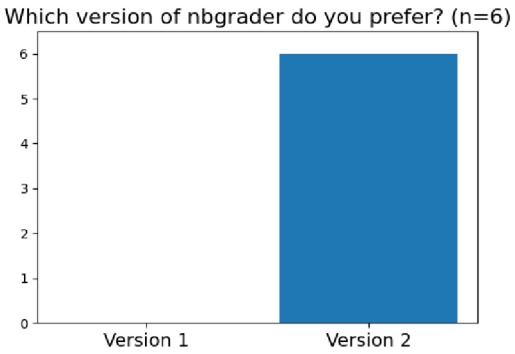


Figure 7.10: nbgrader version comparison

nbgrader versions

Figure 7.10 shows the answers to comparing both nbgrader versions throughout the study, clearly showing a preference for the second version of nbgrader.

For the self assessment, it was asked whether releasing, evaluating, and seeing feedback for a self assessment is easy to use.

self assessment



Figure 7.11: Self assessment Ratings

The results in Figure 7.11 show that evaluating and seeing feedback is seen as easy to use, with a median of 7 for evaluating and 8 for seeing feedback. The release of a self assessment was rated mostly positive, with a median of 6 indicating potential for improvements. The number of participants for evaluating and seeing feedback is 5, because the question was not asked correctly for one participant. It combined the

evaluating and seeing feedback in one question instead of separating it, leading to removing this result from the analysis.

group submissions

The group submissions are rated on whether submitting for a group is easy and whether autograding takes less, equal, or more effort.

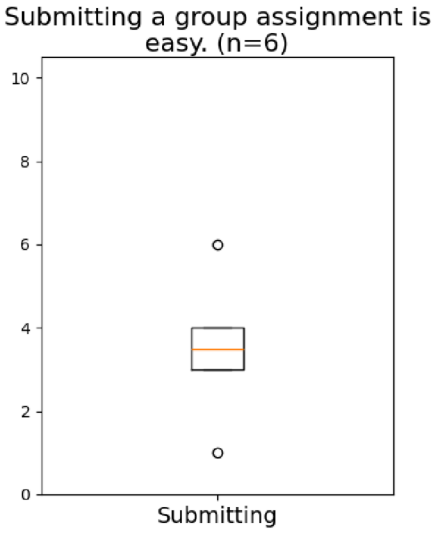


Figure 7.12: Rating of Group submission submitting

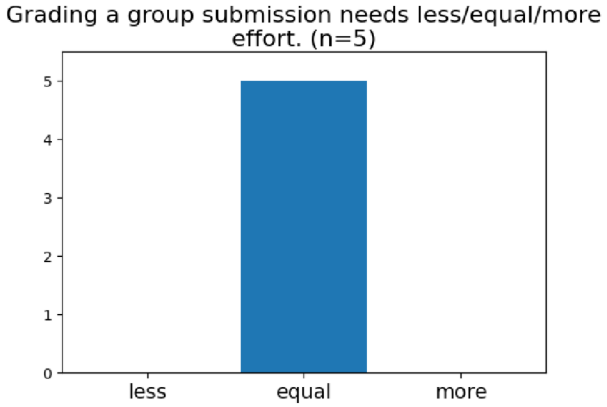


Figure 7.13: Rating of Group submission grading effort

The results shown in Figure 7.12 with a median of 3.5 and the highest rating of 6 indicate that submitting a group assignment is not easy and should be improved. The rating in Figure 7.13 captures the effort relation of the normal grading process and the grading process of group submissions. According to the participant's rating, both autograding processes need equal effort. One participant did not give an answer for this statement. Thus, the results are only for 5 participants.

system usability scale

Closing part The results of a modified system usability scale results are shown in Figure 7.14.

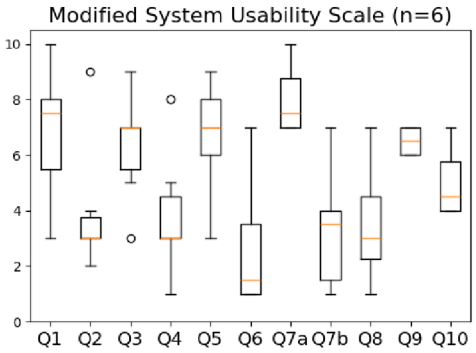


Figure 7.14: Modified System Usability Scale Rating

The statements to rate are the following:
Q1: I think that I would like to use this system frequently.
Q2: I found the system unnecessarily complex.

Q3: I thought the system was easy to use.
Q4: I think that I would need the support of a technical person to be able to use this system.
Q5: I found the various functions in this system were well integrated.
Q6: I thought there was too much inconsistency in this system.
Q7a: I would imagine that most people with a computer science background would learn to use this system very quickly.
Q7b: I would imagine that most people without any specific background would learn to use this system very quickly.
Q8: I found the system very cumbersome to use.
Q9: I felt very confident using the system.
Q10: I needed to learn a lot of things before I could get going with this system.
The ratings of Q1 have a median of 7.5 and a range from 3 to 10. This indicates that most participants would like to use the system frequently.
For Q2, the system's complexity, the median is 3 with a range of 2 to 4 for most participants, which means it is not unnecessarily complex. One participant gave this a rating of 9, meaning unnecessarily complex, which is an exception for the other participants.
The system is seen as mostly easy to use, with a rating of 7 as the median for Q3.
The participants disagreed mostly with the need for the support of a technical person regarding Q4 with a median of 3 and ratings less or equal to 5, except one participant.
For Q5, most participants agreed that the functions of nbgrader were well integrated, with a median of 7 and all ratings higher than 6 except one.
Most participants disagree that nbgrader is too inconsistent. The results for Q6 show a median of 1.5 and only one rating higher than 4.
The rating of Q7a and Q7b shows that participants agree that using nbgrader can be learned quickly with a computer science background, with all ratings being higher than 7. Most participants disagreed with the statement for users without any specific background, with a median of 3.5.
Most participants slightly disagree with the statement of Q8, but one participant also slightly agrees with a rating of 7. This rating indicates minor obstacles when using nbgrader.
There was a slight agreement for being confident while using the system, with all ratings being either a 6 or 7 for Q9.
The participants neither agree nor disagree with Q10 because the median lies at 4.5, and the ratings range from 4 to 7. This says that nbgrader has some requirements that need to be learned before.

Comments by Participants The participants were encouraged to think aloud and to comment on anything they thought unintuitive. Nearly all participants commented that changes to the current directory were not noticeable. For example, when creating an assignment, participants did not notice that the directory changed and that they had to open and create notebooks through the current directory menu. One observation is that some participants attempted to open the assignment with the edit button instead. Similar observations occurred when the participants tried to look at feedback. One participant even pointed out the inconsistency with opening Notebook

current directory unnoticed

documents through the assignment list, which opens the Notebook document directly in the editor instead of changing the current directory.

error message

The dialog for error messages was criticized as "too small" when looking at the log and the log was criticized as containing too much information. The traceback in the dialog was said to be "not useful".

menu navigation

Several participants needed help with the menu navigation when trying to reach the manage submissions screen after manually grading an assignment and being in a submenu of the manual grading screen.

button design

The button for generating an assignment was disliked for "not saying much", because of its appearance as a graduate hat. The button for opening the source cell copies in the manual grading screen was criticized similarly and should be renamed according to a participant. The buttons that are not usable yet should indicate which previous step is needed. The release feedback button should show how many feedback files will be released.

commands

All three times a command needed to be used, it was disliked, and a solution with a button was wanted. This was especially remarked for the group submissions.

user interface

One participant summarized their comments for the study in two words, "More UI". Another wanted more user interface highlighting for frequently used actions. Tooltips were also mentioned as a possible addition.

no clear protocol for hints

Threats to validity During the study's conduction, two issues threatened its validity. There was no clear protocol for conductor intervention and responses. This includes no precise method of determining when to give participants hints and not necessarily equal hints. Both of these can change the participant's rating of the system, as they may have more or less information and success at solving the tasks.

learing effects

Learning effects from the task in the first version can be used for the task in the second.

7.3 Summary

The study results show a clear improvement from the first to the second version, which answers Research Question 4. This improvement is more significant for the generation process than the grading process.

The self assessment was seen positively, but releasing assignments was criticized for using a command line application.

The group submissions are also criticized for being made through the command line interface. The effort involved in grading is equal to that required to grade regular assignments.

The answers for the modified version show nbgrader as slightly positive. It is seen as more accessible to learn with a computer science background and not so easy to learn without it. According to the results, it also requires previous knowledge before using it.

The parts of nbgrader that were most relevant to participants' comments were the user interface, especially missing signifiers that guided them through the process and indicated what they should do. A button should be used instead of the command line interface for newly implemented features like autograding without metadata, releasing submissions, and submitting in a group.

Chapter 8 Results

workflow

To conclude, nbgrader has a workflow of five main steps for the instructor and six optional steps for the distribution of assignments. A specific folder structure is used to mirror this workflow and can be used to retrieve an assignment's state through the file system.

architecture

Architecture-wise, nbgrader is versatile with possible usages as a command line application or integrated into the Jupyter framework. Using *JupyterFrontendPlugins* with several compatibility improvements allows it to be used in Jupyter Notebook and JupyterLab. A three-layer design describes nbgrader's mostly server-side architecture consisting of high level components like the API and command line applications, which use the assignment level components which transform assignments by copying and changing them. These assignment level components use Notebook document level components, which change the Notebook documents in an assignment and scope all changes to a single Notebook document and its cells. This answers Research Question 1 of how nbgrader works and what architecture it uses.

problems with nbgrader

Research Question 2 and Research Question 3 ask for the problems instructors have while using nbgrader and the required changes needed. These are answered through a study that concluded that the most pressing issues lie in the generation of assignments and the automatic grading of student submissions. The error messages and methods to cope with occurring errors are seen as insufficient. Additionally, wanted features are found, like the possibility of self assessments and group submissions. These results are threatened due to stemming from only one participant.

analysis and design

The analysis and design showed that all the requirements could be implemented within nbgrader.

implementation

All core requirements and most other requirements were implemented. These include error dialogs with more information, changes to the grading process to make Notebook documents that have corrupted metadata gradable, self assessments assignments and a proof of concept for group submissions. A button to grade all assignments of a submission was added, as well as hints for students and instructors to see cells with duplicate grade identifiers.

validation study

The implementation was validated with a second study in which six participants used the previous and changed nbgrader version of nbgrader and were presented with the newly added features. The results show a clear preference for the changed version, signifying successful changes, which answers Research Question 4. The generation process was seen as more usable. Most negative comments made by the participants regarded the user interface and problems while using it. Clear signifiers of which step can or should be taken next are missing or not noticeable. Furthermore, not all participants noticed directory changes when looking at an assignment or feedback.

Chapter 9 Future Work

The requirements defined in chapter 4 and analyzed in chapter 5 can be used to implement the missing or not fully implemented features.
The validation study in chapter 7 showed several issues users had with nbgrader's graphical user interface, which can be solved in future work.
The new features added to nbgrader should be tested and in a test classroom environment to get further feedback for these features and to improve their usability.

nbgrader architecture

The description of the nbgrader framework's architecture can be used to compare it to other automatic assessment frameworks and analyze it further to identify its architectural problems.

nbgrader security

One aspect this thesis does not cover is security, and nbgrader is used in teaching environments where grades are used to evaluate students, which makes it paramount that nbgrader is secure. If students had the opportunity to cheat by any means or even, in the worst case, manipulate their scores or those of other students, it would be very problematic.

Appendix A Preprocessor List

The list of all preprocessors and their usage is given in Table A.1. Unless said otherwise, the usage will refer to all cells of the Notebook document the preprocessor is applied to.

Preprocessor	Usage
CheckCellMetadata	Validates the (nbgrader) metadata
ClearHiddenTests	Removes content between begin and end hidden tests blocks
ClearMarkScheme	Removes content between begin and end mark scheme blocks
ClearOutput	Removes the execution output below cells
ClearSolutions	Removes content between begin and end solution blocks
ComputeChecksums	Adds or updates the checksum of a cell
DeduplicateIds	Checks cells for duplicate identifiers and removes it from all but the last cell in the Notebook document
Execute	Executes the cells with options to ignore errors, use a timeout, and retry the execution on error
GetGrades	Saves scores from the database to the notebook
IncludeHeaderFooter	Adds cells as a header and footer to the Notebook document
InstantiateTests	Replaces (hashed) *Autotest* markers with tests
AssignLatePenalties	Assigns penalties for late submissions to the database
LimitOutput	Limits the number of lines for the cell output and traceback
LockCells	Makes cells not deletable and not editable
OverwriteCells	Overwrites metadata and content, when a cell was changed and can also be configured to add missing cells in a student submission
OverwriteKernelspec	Checks and overwrites the kernel specification metadata attribute of the Notebook document
SaveAutoGrades	Saves autograding grades to the database
SaveCells	Saves cell information about grade and solution cells to the database

Table A.1: List of preprocessors (from [nbgj])

In total, there are 18 preprocessors.

Appendix B Converter List

The following paragraphs will briefly describe the converters used by nbgrader and use figures to show which preprocessors are used. The information about the converters was gained by looking at the GitHub repository [nbga]

GenerateAssignment For generating an assignment's student version, two converters exist. One is named *GenerateAssignment* and the other *Assign*. The latter exists for compatibility only and inherits its functionality from the *GenerateAssignment* converter.

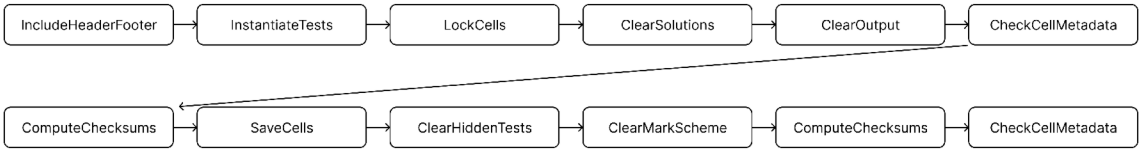


Figure B.1: Generate assignment converter

In Figure B.1, the preprocessors used by the can be *GenerateAssignment* converter can be seen.

GenerateSourceWithTests The *Autotest* marker mentioned in section 3.1 can also be replaced by the *GenerateSourceWithTests* converter to create an assignment with the automatically generated tests, which then can be used as the source version of an assignment.

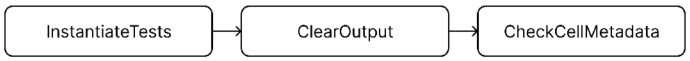


Figure B.2: Generate Source with Tests converter

The converter uses the *InstantiateTests*, *ClearOutput* and *CheckCellMetadata* preprocessor as seen in Figure B.2.

GenerateSolution The *GenerateSolution* converter creates a version of an assignment that students can look at as a reference for solving the assignment. This version includes solutions to programming tasks and the output and content of cells.

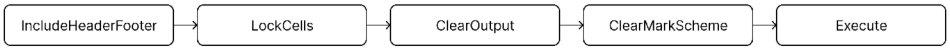


Figure B.3: Generate Solution converter

The five preprocessors used for this converter are shown in Figure B.2.

Autograde The *Autograde* converter uses two phases for processing assignments. The first is the sanitization phase, which is shown in Figure B.4.

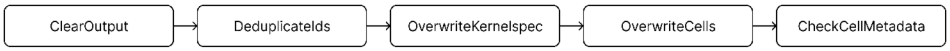


Figure B.4: Autograde converter (sanitization phase)

After the sanitization phase, the autograding phase starts, which executes all cells in the Notebook document.

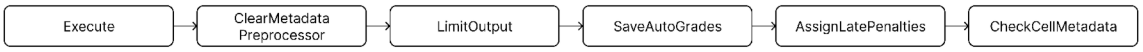


Figure B.5: Autograde converter (autograde phase)

With the autograding phase show in Figure B.5, it can be seen that the complete autograding process uses 11 preprocessors to grade an assignment.

GenerateFeedback Two converters can be used to generate feedback for a student. These are the *GenerateFeedback* and *Feedback* converter. Like the autograding converters, the *Feedback* converter is only used for compatibility.



Figure B.6: Autograde converter (autograde phase)

The generation of feedback is done with the six preprocessors in Figure B.6.

Appendix C Bibliography

[Ber22] Erlend Berntsen. Freja: A framework for effectively creating programming assignments based on code transformations, 2022.

[bla] `https://www.blackboard.com/` (last accessed on 27.01.2024).

[Bro96] John Brooke. Sus: a "quick and dirty'usability. *Usability evaluation in industry*, 189(3):189–194, 1996.

[Cri21] Cristian D. González-Carrillo, Felipe Restrepo-Calle, Jhon J. Ramírez-Echeverry, Fabio A. González. Automatic grading tool for jupyter notebooks in artificial intelligence courses. *Sustainability*, 2021.

[EM23] Mohamed Elhayany and Christoph Meinel. Towards automated code assessment with openjupyter in moocs. In *Proceedings of the Tenth ACM Conference on Learning@ Scale*, pages 321–325, 2023.

[FL14] Jordan FERNANDO and MM LIEM. Components and architectural design of an autograder system family. *Olympiads in Informatics*, 8, 2014.

[HNVB21] Marcelo Guerra Hahn, Silvia Margarita Baldiris Navarro, Luis De La Fuente Valentín, and Daniel Burgos. A systematic review of the effects of automatic scoring and automatic feedback in educational settings. *IEEE Access*, 9:108190–108198, 2021.

[IAKS10] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli calling international conference on computing education research*, pages 86–93, 2010.

[JBB+19] Project Jupyter, Douglas Blank, David Bourgin, Alexander Brown, Matthias Bussonnier, Jonathan Frederic, Brian Granger, Thomas Griffiths, Jessica Hamrick, Kyle Kelley, M Pacer, Logan Page, Fernando Pérez, Benjamin Ragan-Kelley, Jordan Suchow, and Carol Willing. nbgrader: A tool for creating and grading assignments in the jupyter notebook. *Journal of Open Source Education*, 2(16):32, 2019.

[jupa] `https://jupyter.org/` (last accessed on 10.01.2024).

[jupb] `https://jupyterlab.readthedocs.io/en/stable/extension/extension_dev.html#application-plugins` (last accessed 19.01.2024).

[jupc] https://jupyter.org/hub (last accessed on 18.01.2024).

[jupd] https://docs.jupyter.org/en/latest/projects/kernels.html (last accessed 18.01.2024).

[jupe] https://jupyterlab.readthedocs.io/en/latest/ (last accessed on 25.01.2024).

[jupf] https://jupyter-notebook.readthedocs.io/en/stable/ (last accessed on 25.01.2024).

[jupg] https://jupyter-server.readthedocs.io/en/latest/ (last accessed 19.01.2024).

[juph] https://jupyter-server.readthedocs.io/en/latest/developers/extensions.html (last accessed 19.01.2024).

[Kle20] Nik Klever. Jupyter notebook, jupyterhub and nbgrader. *Becoming Greener—Digitalization in My Work; The Publication Series of LAB University of Applied Sciences*, pages 37–43, 2020.

[Lan19a] DEREK Land. Automatic grading in engineering classes. In *Proceedings of the 10th International Conference on Physics Teaching in Engineering Education PTEE*, 2019.

[Lan19b] DEREK Land. Automatic grading in engineering classes. In *Proceedings of the 10th International Conference on Physics Teaching in Engineering Education PTEE*, 2019.

[Nar06] Susanne Narciss. *Informatives tutorielles Feedback: Entwicklungs-und Evaluationsprinzipien auf der Basis instruktionspsychologischer Erkenntnisse*. Waxmann, 2006.

[nbc] https://nbconvert.readthedocs.io/en/latest/nbconvert_library.html#Custom-Preprocessors (last accessed 19.01.2024).

[nbga] https://github.com/jupyter/nbgrader/tree/main/nbgrader/converters (last accessed on 27.01.2024).

[nbgb] https://nbgrader.readthedocs.io/en/stable/command_line_tools/index.html (last accessed 19.01.2024).

[nbgc] https://nbgrader.readthedocs.io/en/stable/user_guide/creating_and_grading_assignments.html (last accessed 18.01.2024).

[nbgd] https://nbgrader.readthedocs.io/en/stable/user_guide/managing_assignment_files.html (last accessed on 02.02.2023).

[nbge] https://nbgrader.readthedocs.io/en/stable/user_guide/philosophy.html (last accessed on 02.02.2023).

[nbgf] https://github.com/jupyter/nbgrader (last accessed on 18.01.2024).

[nbgg] https://github.com/jupyter/nbgrader/blob/ab5512d44869bf8c44c5a9921ddfa40fe941672e/nbgrader/apps/nbgraderapp.py (last accessed on 18.01.2024).

[nbgh] https://github.com/jupyter/nbgrader/blob/main/nbgrader/nbgraderformat/v3.json (last accessed on 02.02.2023).

[nbgi] https://github.com/jupyter/nbgrader/tree/12136c2eccde44ca7a86b7805e240a406704cb74 (last accessed 21.01.2024).

[nbgj] https://github.com/jupyter/nbgrader/tree/main/nbgrader/preprocessors (last accessed on 27.01.2024).

[PMC19] Matthew Peveler, Evan Maicus, and Barbara Cutler. Comparing jailed sandboxes vs containers within an autograding system. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, pages 139–145, 2019.

[pro] https://github.com/jupyter (last accessed on 11.01.2024).

[PSM+17] Anastasia Pampouchidou, Panagiotis G Simos, Kostas Marias, Fabrice Meriaudeau, Fan Yang, Matthew Pediaditis, and Manolis Tsiknakis. Automatic assessment of depression based on visual cues: A systematic review. *IEEE Transactions on Affective Computing*, 10(4):445–470, 2017.

[pyt] https://www.python.org/ (last accessed on 11.01.2024).

[SPLC20] Samuele Segoni, Giulio Pappafico, Tania Luti, and Filippo Catani. Landslide susceptibility assessment in complex geological settings: Sensitivity to geological information and insights on its parameterization. *Landslides*, 17:2443–2453, 2020.

[SS22] Sven Strickroth and Michael Striewe. Building a corpus of task-based grading and feedback systems for learning and teaching programming. *The International Journal of Engineering Pedagogy*, 2022.

[str] https://systemscorpus.strickroth.net/ (last accessed on 18.09.2023).

[SvAY+20] Chad Sharp, Jelle van Assema, Brian Yu, Kareem Zidane, and David J. Malan. An open-source, api-based framework for assessing the correctness of code in cs50. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '20, page 487–492, New York, NY, USA, 2020. Association for Computing Machinery.

[TAHE22] Mohamed Tarek, Ahmed Ashraf, Mahmoud Heidar, and Essam Eliwa. Review of programming assignments automated assessment systems. In *2022 2nd International Mobile, Intelligent, and Ubiquitous Computing Conference (MIUCC)*, pages 230–237. IEEE, 2022.

[tra] `https://traitlets.readthedocs.io/en/stable/` (last accessed 19.01.2024).

[TRR00] Roger Tourangeau, Lance J Rips, and Kenneth Rasinski. The psychology of survey response. 2000.

[VSBS+94] Maarten W Van Someren, Yvonne F Barnard, Jacobijn AC Sandberg, et al. The think aloud method: a practical approach to modelling cognitive processes. *London: AcademicPress*, 11:29–41, 1994.

[Wil16] Chris Wilcox. Testing strategies for the automated grading of student programs. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, pages 437–442, 2016.

Appendix D Digital Appendix

The digital appendix is submitted on a SD card. To get access to this material, they shall contact the author or LuFG i9.

Eidesstattliche Versicherung

Thape, Felix	422192
Name, Vorname	Matrikelnummer (freiwillige Angabe)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/Masterarbeit* mit dem Titel

Redesign and Implementation of nbgrader to Improve the Instructor User Experience

ohne unzulässige fremde Hilfe (insbes. akademisches Ghostwriting) erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Aachen , February 2, 2024	
Ort, Datum	Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

§ 156 StGB: Falsche Versicherung an Eides Statt
Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt
(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.
(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

Aachen , February 2, 2024	
Ort, Datum	Unterschrift