WIRTSCHAFTS
UNIVERSITÄT
WIEN VIENNA
UNIVERSITY OF
ECONOMICS
AND BUSINESS

Bachelor Thesis

Towards Automated Grading of Jupyter Notebooks for University Programming Assignments

Fabian Ebner

Date of Birth: 19.04.2001
Student ID: 12024077

Subject Area: Information Business

Studienkennzahl: UJ033561

Supervisor: Prof. Dr. Axel Polleres

Date of Submission: 13. August 2024

Department of Information Systems & Operations Management, Vienna University of Economics and Business, Welthandelsplatz 1, 1020 Vienna, Austria

DEPARTMENT FÜR WIRTSCHAFTSINFORMATIK
UND OPERATIONS MANAGEMENT DEPARTMENT
OF INFORMATION SYSTEMS AND OPERATIONS
MANAGEMENT

Contents

List of Figures

List of Tables

Listings

Abstract

The use of software to automatically grade programming assignments has been shown to improve the quality of education. While there are several different tools available, many come with a series of flaws, which limit the trust instructors can place in them. As a result, many courses still need teaching staff to manually inspect all commissions. A popular choice for grading notebook-formatted assignments is nbgrader. The primary objective of this work is to improve the current implementation of the software, by overcoming some of the main challenges. This includes the integration of a solution that allows to assign points to individual tests. The tests for a task are then executed conjointly, and the achieved points get automatically assigned to the total score of the assignment. In addition, functions to improve the quality of the feedback are presented, together with a synopsis of best practice techniques for the creation and auto-grading of programming assignments.

1 Introduction

Technologies to automate the grading process of programming assignments have been researched for decades. Professors at Stanford University published papers on the topic as early as 1965. The need to evaluate the code of many students served as motivation for the development of a software solution. Back then, ALGOL programs were used to grade the students' assignments written in the same language [9]. Since then, a lot has changed. A multitude of advancements in both hardware and software have been made, but the need for auto-graders still remains. Furthermore, the motivation also stays unchanged: grading code assignments is a tedious task. As manual grading is prone to errors, great advantages can be gained by automating said process [6]. Recent developments in education expand the need for an effective auto-grading solution. An increased number of university courses now require students to write code. *Computational thinking* , the ability to solve problems using computer science techniques, is nowadays a requirement for students in almost all STEM fields [22]. This results in an increasing number of programming assignments, which all need to be graded by someone, or something. Computational notebooks offer an attractive way to create such programming assignments. In the last few years, Jupyter notebooks have become a popular tool in education. Offering an easy-to-use environment, they are well-suited to be used in teaching. Especially when it comes to learning Python, many universities use Jupyter notebooks to create assignments [21]. Using the Jupyter environment in an educational context brings many benefits. Thanks to their simplicity, they can be used by instructors to implement modern teaching methods [4]. Additionally, they allow users to write code in almost all popular programming languages [34]. Notebooks can also be edited in a web-based application which eliminates the need for students to install special software. Furthermore, the process of creating and running code in notebooks is simple [1]. Thus, they can be used in both introductory and advanced courses. Should the teaching staff decide to use notebooks as the basis for assignments, they must find a way to create and grade them. One tool that can achieve this is called *nbgrader*. The workflow of the software allows the creation and publishing of assignments. It is also used to grade the notebooks and to create feedback files for the students [16]. This solution is also employed in the *Data Processing 1* (DP1) course at the Vienna University of Economics and Business. Due to my involvement with the course as a tutor, I came to notice that it was not possible to completely auto-grade the assignments. While nbgrader is certainly very useful, it still has some flaws. This is also supported by other teaching staff, who voice their complaints via forums like GitHub [25]. The problems include running

multiple tests at once and efficiently assigning partial points, in the common case that a student completes some but not all tests.

The goal of this thesis is to develop and describe an approach to auto-grading that eliminates those flaws. As previously hinted, this approach will build upon the nbgrader tool, and will thus be applicable for grading Jupyter notebooks. To achieve this goal, elements of *unit testing* get merged with the existing approach for grading with nbgrader. But solving the previously discussed problems is not the only objective. The new approach aims to be both simple and flexible. Due to its simplicity, the testing code can be easily maintained and modified. The flexibility allows many different tasks to be graded with the same code blueprint. Another problem with auto-grading lies within the provided feedback. To be able to include long and complex outputs in the feedback files, while still keeping the files tidy, I constructed a specialized function. It processes the output in a way that keeps it minimal but still enables fast comparisons for either the instructors or the students. This way, the approach does not only improve the existing auto-grading techniques but can also assist in cases where manual grading is unavoidable.

I structured the thesis as follows. Chapter 2 provides an introduction to the Project Jupyter environment, laying a solid foundation for those interested in implementing this approach. The focus lies on three important subprojects of Jupyter, namely *Jupyter Notebooks*, *JupyterLab*, and *JupyterHub*. This foundation is further enhanced in chapter 3, as the standard process of creating and grading assignments with nbgrader is explained. Finally, chapter 4 dives into the new and improved approach. The included checklist on best practices for the new approach can help teaching staff to minimize the time spent on manual grading.

1.1 Methodology for Replication

The code presented in this thesis was developed and tested in an Anaconda environment, where all necessary software was installed. I used the graphical interface called *Anaconda Navigator*. The official documentation offers a detailed explanation of how to install and use the software [2]. The installation of packages and launching of JupyterLab were all conducted through the *Anaconda Navigator*. This setup includes *Python*, *JupyterLab*, *nbgrader*, and various modules and libraries. The auto-grading approach builds upon the *unittest* module, which is part of the *Python Standard Library* [11]. Therefore, the replication of the results heavily relies on the Python version. Equally important is the *JupyterLab* extension *nbgrader*. The official documentation of *nbgrader* provides easy-to-understand instructions on how to set it up, including steps to install it via Anaconda [31]. These instruc-

tions were followed during the creation of this thesis. Although I expect the proposed solution to work with most nbgrader installations, I will include the software versions used, to ensure that my results can be replicated. The following table lists the important software and their corresponding version numbers used in the creation of this thesis.

Name	**Version**
Anaconda Navigator	2.6.0
Python	3.10.9
JupyterLab	3.6.7
nbgrader	0.9.1
Pandas	1.5.3
IPython	8.10.0

Table 1: Software Versions

2 Jupyter

The auto-grading approach for programming assignments will be based on the software provided by Project Jupyter. It is important to be familiar with the key parts and main features of this software in order to understand why it was chosen for this approach. Project Jupyter was created in 2014, out of the iPython Project. The software and standards it provides are open-source and can be used together with more than 40 programming languages [33]. Jupyter is widely used in data science, but also in physics, chemistry, economics, and education. The more than 10 million Jupyter notebooks available on GitHub are proof of the widespread use and acceptance of this document type as a way to create and share computational work and knowledge. One reason for the notebook's popularity is its ability to tell a story with code and data. In this context, the label *interactive computing* is often associated with the Jupyter environment, as it enables users to solve tasks with the help of computers and data [12].

2.1 Jupyter Sub-projects

Project Jupyter is split into various sub-projects, which are used to perform different tasks. In the following, three sub-projects will be described in detail: Jupyter Notebooks, JupyterLab, and JupyterHub.

2.1.1 Jupyter Notebooks

In this thesis, the term *Jupyter notebook(s)* will be used to address the document type which is saved with the *.ipynb* extension and not the web editor with the same name.

Jupyter Notebooks can be classified as *computational notebooks*, which allow the editor to embed and execute code. Easily editable through a web application, they create an interactive system for beginners and experts to write and test their code, as results are displayed within the notebook rather than within a console. If the embedded code is used for plotting data, then the resulting plot can also be displayed in the notebook. In addition to code, it is also possible to include text and images in the notebook. Those can be formatted with markup languages like the *Hyper Text Markup Language* (HTML) [8] and *Markdown* [7], or by using *LaTeX* [36]. The notebooks can be easily shared, partly because their internal format is based on the *JavaScript Object Notation* (JSON) [14]. The base structure of the notebook consists of *cells*, which are either code cells, raw cells, or markup cells [28].

Thanks to the different cell types available, it is possible to construct a document that combines code, images, equations, and text. Using those elements, a Jupyter notebook can be transformed into an interactive application that tells a story. This stems from the fact that the embedded code can be edited, executed, and documented using text and visualizations [4].

Users have many options when it comes to sharing their notebooks. As it is possible to view and execute them online, many types of documents can be found in the form of a Jupyter notebook, including research papers, blogs, and textbooks. Many online courses and curricula at universities use notebooks as teaching materials [12].

With this information, conclusions can already be drawn as to why Jupyter notebooks are a valuable asset for teaching data science courses. An in-depth analysis of the applications of notebooks in education is provided later in this chapter.

2.1.2 JupyterLab

JupyterLab is the sub-project focused on interacting with computational notebooks. According to the official website, JupyterLab is described as *"[...] the latest web-based interactive development environment for notebooks, code, and data. Its flexible interface allows users to configure and arrange workflows in data science, scientific computing, computational journalism, and machine learning. A modular design invites extensions to expand and enrich functionality."*[34]

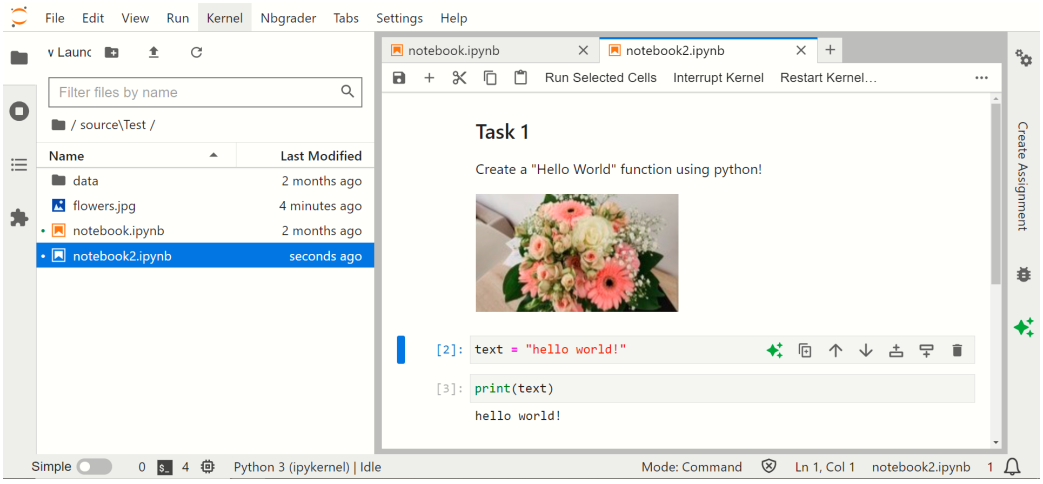


Figure 1: JupyterLab displaying a Jupyter Notebook

The JupyterLab interface shares some similarities with traditional inte-

grated development environments (IDEs), but has additional functionality. It features text editors and terminals and can be used to view files in multiple formats like CSV and JSON. Overall, it is a more advanced authoring application compared to the interface called *Jupyter Notebook*, which was the original application for interacting with *.ipynb* files. A similar version to run notebooks locally is also available, which is called Jupyter Desktop [29].

JupyterLab offers full support for Jupyter notebooks. Many functions are available for working with the *.ipynb* files. Cells can be dragged between notebooks and code can be collapsed or expanded. Multiple Views can be created from a single notebook to create dashboard-like applications [32].

Figure 1 displays a notebook opened in JupyterLab. On the top, you can see a menu bar with multiple options like *Edit*, *Run*, and *View*. The left sidebar can be used to browse and upload files. Two notebooks, one image, and one folder labeled "data" are visible. It is also possible to display a sidebar on the right side of the screen [30]. The Figure also shows what a notebook can look like. Multiple types of cells are visible. The first cell is a markdown cell displaying formatted text as well as an image. The other two cells are code cells, used to implement a simple "hello world" program.

Figure 2: The Raw Cell Content of the Notebook

In Figure 2 the same notebook is visible, but this time the raw content of the cells is shown. In the first cell, Markdown elements have been used to create the headline and implement an image.

2.1.3 JupyterHub

JupyterHub is the gateway for using Jupyter Notebooks and JupyterLab with a large group of participants. Employing this application allows corporate or educational groups to create a customized JupyterLab environment. Users get access to an individual, pre-configured workspace without the need to install anything. It also provides a way for the administrator to share data and documents with the users, for example, instructions or data sets. For institutes with good hardware, it is possible to run JupyterHub on their servers. If not enough resources are available, the service can also be hosted in the cloud. [35].

The application consists of multiple subsystems. An HTTP proxy forwards the requests to the *hub*, which is the heart of the application. The *hub* then confirms the user login and creates a single-user server. One *hub* can spawn multiple servers so that JupyterHub can be used by more than 100 users. Depending on the number of users, two main distributions are available [33]. The distribution *The Littlest JupyterHub* (TLJH) can be used to give a group of up to 100 users access to Jupyter Notebooks, and requires almost no knowledge in system administration [17]. On the other hand, *Zero to JupyterHub with Kubernetes* can be deployed by big organizations, to grant many users access to Jupyter Notebooks. As the name suggests, *Kubernetes* are used to run JupyterHub on a cloud [18].

2.2 Educational Applications

Using Project Jupyter in an educational context has many benefits. One of the reasons why it is used by teaching staff is the ability to provide resources to the students without the need to install software on their client's machine. Whether it is computing resources or lecture materials, the Jupyter environment allows the students to access both, as JupyterHub can also be used as a file management system [5]. The Jupyter environment has many characteristics that make it well-suited for educational use. It grants access to multiple programming languages through a single web interface. Students can easily switch between languages like Java, C++, and Python. Furthermore, there are multiple features that help to simplify the process of programming. First off, users can start to code as soon as they open a Jupyter Notebook. It is not required to create a project folder with multiple files. Additionally, users can program in Python without switching from *.py* files to the terminal. This stems from the fact that the output, results, and possible error messages are displayed right under the executed code cell. The possible fields of application are another benefit of utilizing Jupyter notebooks in the classroom.

They can be used to teach basic programming, as well as complex concepts in fields like computer and data science [1].

Regarding Jupyter notebooks, there are numerous ways to implement them in a lecture. A rather underutilized way to use notebooks would be a linear textbook. Here, the document is more of a reading exercise than an interactive experience. Workbooks are another use case, which employs more of the available features by implementing active elements like code cells. This way, students can modify included code snippets and create new outputs, which boosts engagement with the lecture's content. Students might also use notebooks to take notes to capture the content of a lesson. Additionally, notebooks are excellent for creating assignments [4].

With the help of extensions, Jupyter Notebooks can easily be turned into presentation slides. One of the extensions that add this functionality is *RISE*. With an additional toolbar, the metadata of cells can be changed to make them suitable for presentations [3].

This chapter has highlighted numerous features of Jupyter Notebooks. Based on these features, several benefits of using Jupyter Notebooks for assignments have been identified:

1. As notebooks are both the application to execute code, as well as the documentation of said code and its results, the whole assignment can be created in one document. This eliminates the need for separate question-and-answer files. Students download a notebook with instructions, and hand in the same file after the code cells have been filled in. The notebooks can be modified with ease, thanks to their modular structure. If only small changes need to be made, only some cells have to be modified. This allows to update assignments with minimal effort.

2. The result of the executed code is displayed below the respective cell. This seems to be especially useful for programming beginners because they can immediately see the result and troubleshoot potential errors. In addition, the code can be split up between multiple cells, so it is easy to solve a more complex problem step-by-step.

3. Due to in-browser editing, students do not have to install any software for the course. They can access their files from multiple devices, which enables them to work on devices provided by the university as well as personal machines.

4. Using Jupyter-compatible software, the assignments can be created, collected, and graded. Many of those processes can be automated. For the grading part, the software *nbgrader* can be used [16].

2.2.1 Challenges of Using Notebooks in the Classroom

Employing the Jupyter environment in the classroom comes with a few challenges. It is important to give a quick overview of the drawbacks and pitfalls of using the popular software in the classroom.

While there are some technical challenges with the setup of JupyterHub like managing user permissions, the concerning challenges are those that arise when students use JupyterLab and Jupyter notebooks. Due to the cell-based structure, beginner students can have problems with understanding the execution flow. As code cells can be executed in a non-linear manner, bugs can be harder to find. Jumping between different code cells and the possibility to edit and re-run previous cells can confuse inexperienced users, as working with a non-linear execution flow can result in unexpected outcomes. Especially if the code is fragmented between multiple cells, the overall logic can be hard to understand. In that case, an inexperienced user might not see what cells depend on each other, causing confusion if cells are executed multiple times in different orders. Another potential issue lies in the difference between traditional IDEs and JupyterLab. Using Jupyter only, students might not get prepared to manage other environments and could miss out on gaining a proper understanding of system states and execution flows [26].

The issue with the execution order of cells goes hand in hand with another challenging aspect of Jupyter notebooks. As all cells in the notebook share a hidden state. Variables and functions that have been defined once can be used across multiple cells. For example, a function defined and executed in one code cell can be used in all subsequently executed code cells. While this certainly comes with benefits, it can also be difficult for inexperienced students to understand. If a variable name gets used twice in different cells, unexpected results can occur. If a snippet of code results in an error, the actual source of that error may be located in a completely different cell. Although there is a number displayed next to each cell to keep track of the execution order, there is no such tool for checking the state of functions and variables [15].

An example can be seen in Figure 1. Here, the number on the left side of the code cells displays the execution order. As shown, the last cell in the notebook has the number "3" beside it, indicating that it was the third cell that was executed. Using this information, we can conclude that the cell with the number "2" was executed previously, so the variable *text* includes the string *"hello world!"*. While this is fairly logical, let's assume another cell is added, which sets the variable *text* to the string *"hello mars!"*. Now, depending on the execution flow, the cell with the print statement will display a different output.

3 Grading Jupyter Notebooks

Manually grading assignments can quickly become a time-consuming task. A large number of students combined with complex tasks to grade can even make it an impossible task. One possible solution to assist the teaching staff is *nbgrader* [20]. This tool based on Jupyter offers features needed to grade notebooks. For the Python programming language, there is even some functionality for auto-grading. Among other things, additional types of cells like solution and test cells are introduced [19]. Although this thesis is focused only on nbgrader, it should be noted that there are other methods available for the task at hand. One example would be Web-CAT, another web-based software used for testing and grading. This solution is compatible with the learning management system Canvas [21].

3.1 nbgrader

The tool nbgrader is an extension for Jupyter and can handle both the creation and grading of assignments. Instructors can create a master copy of an assignment, complete with tests and solutions, which can be turned into a student version without the solutions. The nbgrader can be used together with JupyterHub, in which case it is also possible to facilitate the distribution and collection of assignments through the tool. [16]. While the combination of nbgrader and JupyterHub is certainly handy, the functionality for sharing notebooks can also be outsourced to a learning management system or a digital learning environment [20].

3.1.1 nbgrader for Creating and Grading Assignments

This sub-chapter extensively references and utilizes information from the official Nbgrader documentation [31].

The functionality that nbgrader offers can be used with the "formgrader" extension or with the command line. To run commands in the terminal, the instructors need to navigate to the correct directory first (in most cases, this is the *source* directory). In this example, the focus is on the formgrader. Here, a new assignment can be created by navigating to the formgrader tab "Manage Assignments" and clicking on "Add new assignment...". This will create a new folder in the *source* directory, where all the necessary files (like data or scripts) can be placed. Also, the master copy of the assignment notebook will be located here.

When it comes to creating the actual notebook, most of the tools needed are available through the "Create assignment" toolbar, as shown in Figure

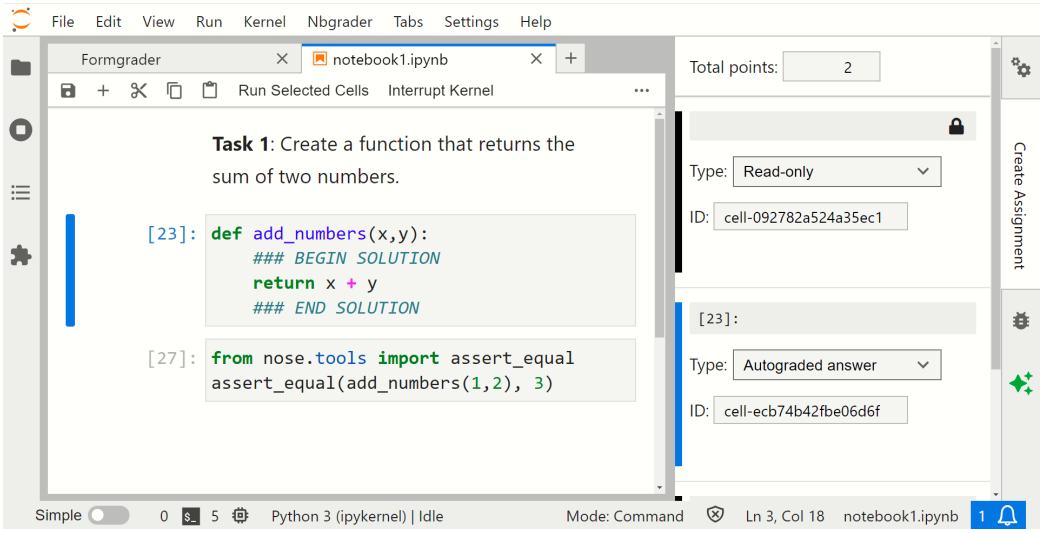


Figure 3: An Assignment in nbgrader

3 on the right-hand side. Through this toolbar, each cell gets a certain type assigned. All cells with a nbgrader type assigned to them need to have a unique ID. A random ID is automatically applied (as seen in Figure 3), but the teaching staff can change those values. Also, certain types of cells can have points assigned to them. During the auto-grading process, all cells are executed. The points of those cells that executed without any exceptions are then added to the total score of the assignment.

When an assignment is created, the first few cells are likely to be instructions. For this case, "Read-only" cells can be used. Those cells cannot be edited by the students, and thus are well-suited to display information.

For cells that need manual grading, two versions are available: the "Manually graded answer" and the "Manually graded task" cells. The answer-type cell is mostly used for non-code answers, like describing a function or written answers to questions. If a cell has this type assigned to it, the student's version of the assignment will display a placeholder, indicating that an answer is expected here. When selecting this cell type, the instructors need to select the number of points that can be achieved in this cell. The cells are graded by manually specifying the achieved points.

In comparison, the "Manually graded task" cells are used for the description of a task. This is mostly used for tasks where the answer can be spread across multiple cells. Instead of assigning points to each cell used for the solution, only the cell with the task description is graded.

Auto-grade cells are the most important type of cell for the auto-grading approach presented in this thesis. Similarly to the manually graded cells,

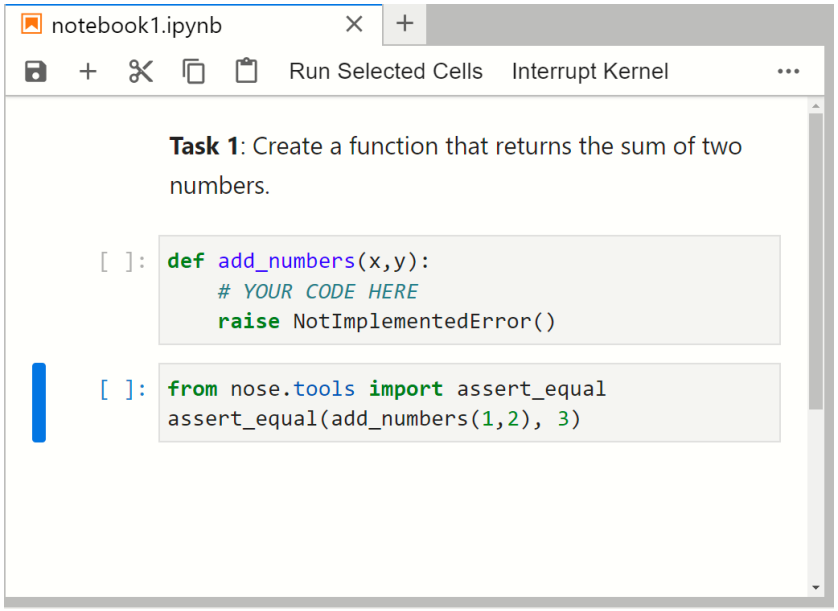


Figure 4: The Student Version of an Assignment

there are two different types available: "Autograded answer" and "Auto-graded test" cells. The answer type is used for cells in which the student writes his answer to a task. No points can be assigned to this cell, but they have a special feature. The special syntax shown in the second cell in Figure 3 hides the solution in the student's version of the notebook. Instead, the student will see a comment like "Your Code here". This can be seen in Figure 4, which displays the student version of the example assignment.

Following the answer cells, the auto-graded test cells are used to test and grade the student's answer. Usually, this cell type is used in combination with assert statements. Contrary to the answer cell type, the grading cells are worth a certain amount of points that can be specified in the toolbar. If the cell passes (no errors occur), then the points get counted towards the total achieved score of the assignment. Otherwise, no points are awarded. This type of test cell can be seen in Figure 3. The last cell in the notebook uses the *assert_ equal()* function, which is recommended by the Jupyter Development Team. In the presented example, the test will complete only if the function *add_ numbers(1,2)* returns the sum of its parameters. To hide the content of test cells, the instructors can use a modified version of the syntax shown in Figure 3. To achieve the desired result of hiding the cell content, the text "### BEGIN/END HIDDEN TESTS" needs to be inserted into the cell, which hides all the code lines in between.

The Jupyter Development Team, responsible for the nbgrader extension,

says that tasks, where the student is required to write functions, can be auto-graded more easily. Ideally, all the features and functions should be tested individually, especially if the solution consists of multiple tasks. Another goal is to cover all possible edge cases. This is done to ensure that a function is indeed working as intended. Emphasis should be put on this goal, as it is essential to ensure that the auto-grading works as intended.

Navigating back to the "Manage Assignments" tab in formgrader allows the instructor to generate the student version. The resulting notebook will look different from the master copy and can be previewed using the formgrader. This generated notebook can be found in the *release* folder. The notebook can then be released to the students. Here, multiple options are available. Nbgrader has a built-in method, that can be used only if the student interface is available in the student's JupyterLab interface. Otherwise, the file can be uploaded to a Learning Management System (LMS), or distributed via email. Depending on the choice made to release the assignment, an appropriate way to collect the submitted assignments is needed. Again, multiple options are available, like collecting them via the built-in method or uploading files using the command line.

After the submitted files have been uploaded to nbgrader and are accessible in the formgrader, the grading process can be started. Two options are available to choose from. Using the formgrader, individual submissions can be auto-graded one by one. For a class with many members, it is advisable to use the command line interface though. With the command "nbgrader autograde 'AssignmentName' –force" all submissions get graded simultaneously. The option *–force* overwrites the existing grades and notebooks. The resulting auto-graded notebook(s) can be viewed in the formgrader, or by navigating to the directory *autograded*. That is also where the manual grading happens. This is required for cells that are labeled with the type "Manual graded answer" or "Manual graded task". Those cells will not get graded automatically, and thus instructors need to manually grade them. For auto-graded cells, the scores can be altered. Furthermore, comments can be added to cells which have been modified by students, to give additional feedback.

Nbgrader also has a feature to generate feedback files. Again, the feature can be used with formgrader or from the command line. After the process is started, HTML files will be created that contain the executed notebook together with the scores achieved and potential comments from the instructors. An example can be seen in Figure 5. Hidden tests are also visible in those files. The files are located in their own directory called *feedback*. Like the student's version of the assignment, the feedback files can be distributed (released) via nbgrader, if the necessary prerequisites are in place. Otherwise, the files can be downloaded and shared using an LMS or email for example.

Student's answer (Top)

```
def add_numbers(liste, x):
    ### BEGIN SOLUTION
    if type(liste) != list:
        return "Wrong Input"

    if type(x) != int:
        return "Wrong Input"

    output_list = []

    for i in range(0, len(liste)):
        output_list.append(liste[i] + x)

    return output_list
```

Comments:
good solution!

In [3]:

Grade cell: `cell-97b259daa6189f6b` Score: 1.0 / 1.0 (Top)

```
from nose.tools import assert_equal

assert_equal(type(add_numbers(exampleList, exampleNum)), list)
assert_equal(add_numbers(exampleList, exampleNum), [6,7,8,9,10])

#(This is a "Read-only" cell)
```

Figure 5: Answer and Test Cells in a Feedback File

3.1.2 Pitfalls of nbgrader

The previous sub-chapter showcases what the nbgrader workflow might look like. While it is certainly streamlined and more efficient than not using any software at all for the creation and grading of assignments, it still has its flaws. Especially when it comes to auto-grading, nbgrader lacks some functionality. In the previous description of nbgrader's features it is stated that if an auto-grade cell fails, no points will be added to a notebook's total score. Assuming that the student's solution passes five out of six assert statements contained in a cell, nbgrader will grade this cell with zero points. This stems from the fact that the executed test cell results in an error as soon as one assertion is not fulfilled. This is a major flaw, which is supported by multiple issues and discussions found on the official nbgrader GitHub [25]. A possible quick fix for this problem would be to create multiple test cells. If each cell with the type "Autograded Test" only contains one assert statement, then it is possible to assign points to each individual test. While this is helpful for auto-grading on multiple tests, it is only feasible if the number of tests is low. Otherwise, the file would get too confusing. On the official GitHub, issues like #1445 recommend using try-except statements and custom functions to overcome this problem. Those workarounds can quickly get confusing too, as maintenance of this code could be a problem as the code grows.

During the research for this thesis, no satisfying solution was found that could handle running multiple tests, as well as automatically grading the student's submission on those tests. While this is the main problem with most solutions, other issues do exist. Those include a lack of flexibility in regard to the kind of tasks that can be tested, as well as overly complicated code, causing difficulty to adapt the test cases to modified tasks. Furthermore, basic implementations of nbgrader fail to deliver meaningful feedback to the students.

4 Improved Auto-Grading Approach

This chapter presents the developed approach for auto-grading assignments. The approach builds upon the nbgrader framework and tries to eliminate some of its flaws.

4.1 Requirements

Before the previously discussed problems can be tackled, the requirements for a new solution should be stated first. The improved approach to auto-grading should have the following characteristics:

1. **Simplicity**: The solution should be simple and minimal. The research for this thesis showed that the solutions proposed on the nbgrader GitHub lack simplicity and tend to be overly complex [25]. Especially when it comes to implementing a partial scoring mechanism, the solutions on GitHub are unsatisfactory, as they often make use of sub-par techniques like multiple levels of nested *try* - *except* blocks.

    - **Implementation**: The improved approach uses an easy-to-read structure. As the tests are written as class methods, they can be easily identified. The structure of the test methods is also simple, due to using a framework specialized for software testing. The scoring mechanism is connected to the execution of tests and works automatically.

    - **Example (see Appendix A)**: The hidden tests for Task 1 are visible in cell (6). Due to the structuring of tests as class methods, existing tests can be found quickly, and new tests can be added with ease. This helps to keep the assignment tidy, which results in improved readability of the cell. Having multiple tests requires a solution to calculate a score from those tests. The achieved points are calculated as the tests are executed, as each test has a point value assigned to it. If the test completes, the points get assigned to the score. This way, instructors can quickly change how many points a task is worth.

2. **Flexibility**: The solution should be flexible enough to handle various tasks. Instructors should not be restricted by the auto-grading approach when creating an assignment. Flexibility is a requirement commonly found in publications regarding auto-grader systems [23].

Additionally, the approach should be flexible enough to handle correct code even when it is very different from the expected code.

- **Implementation**: The framework used in the approach includes a broad range of assertion methods, which can be used to create a vast amount of different tests. Furthermore, additional assertion methods can be imported or created. This enables the teaching staff to write tests for numerous different tasks. The quality of the tests is controlled by the instructors that create them.

- **Example (see Appendix A)**: Task 1 and Task 2 are very different in regard to the code and output that has to be tested. The improved auto-grading approach can handle both. The structure of the testing code remains the same, only the asserts change. This can be seen when cell (6) and cell (11) are compared.

3. **Scalability**: The solution should be able to test complex functions using multiple tests. This feature has already been identified in the previous chapter. The ability to run multiple tests in one test cell is crucial to achieving the desired outcome. Without this feature, the first and second characteristics cannot be satisfied. In this sense, *Scalability* means that the solution has the capacity to support a growing number of tests for each task. While instructors might start off with only a few tests, it should be possible to include as many tests as they want, without sacrificing a great amount of simplicity or flexibility.

   - **Implementation**: This is one of the main features of the new approach. Multiple tests can be pooled into one test suite, which then gets executed. The code needed to do so can be included in a single cell. If the tests have points assigned to them, the achieved score gets calculated.

   - **Example (see Appendix A)**: While there are not many tests included in the example notebook, adding more would be as simple as writing a new class method. The test loader automatically detects the new tests and incorporates them into the test suite. The class *Test_Function* in cell (6) could be extended with an additional method *test_floats()*, to test how the solution handles floating point numbers. The new test method would then be executed automatically by the test runner. It is possible to include a large number of tests while maintaining the simple structure of the test class.

4. **Adaptability**: The solution should be easy to adapt. If individual tests cannot be changed easily, this would make maintenance and reuse of test code complicated. A good approach has the ability for tests to be added, deleted, or modified with ease.

    - **Implementation**: Due to the modular structure used in the approach, it is easy to edit the tests. This includes changing the data used in the tests, adding new tests, and deleting old tests. The value of the tests, measured in points, can also be changed with ease.

    - **Example (see Appendix A)**: The tests in cell (6) can be modified with ease. To discard a test case, the code for the class method has to be deleted. This can be done without worrying about changes in the behavior of the test runner. Due to the high readability of the test cases, testing parameters can also be changed without much effort. For example, if the instructors want to change the message that students have to return, they only have to change the string in the *test_ input_ control()* function.

5. **Utility**: The solution should be useful for both the instructor and the student. This means that the grading approach should not only be used to calculate points but also to give feedback to the students.

    - **Implementation**: The improved approach provides a detailed test output, which can help the students to identify their errors. Furthermore, a custom function can be used to clearly display and compare the output of the student function with the corresponding solution. The value generated for the instructors is described in the previous bullet points.

    - **Example**: The utility provided to the students can be seen in Figure 6. Here, a clear output is provided to show the results of the tests. This can be further optimized by including messages that get printed when a test fails. An example of this can be seen in cell (6) in Appendix A. Here, the first assert in the *test_ input_ control()* function specifies the parameter *msg* to return a string in case the test is not successful.

4.2 The Solution

The proposed solution has all of the five presented characteristics. Thus, an enhancement to the standard approach with nbgrader has been achieved. The new solution utilizes the functionality of the *unittest* Python framework. The documentation of this package is heavily referenced throughout this chapter [11]. To showcase the improved approach to auto-grading assignments, an example will be used. To enhance clarity, it is assumed that the reader has reviewed the assignment provided in the appendix of this thesis (see Appendix A).

Let's assume that we are creating a simple task for an "Introduction to Data Science" class. We want the students to implement a function called add_numbers(x, y), that takes as input a list x and an integer y. The task is to return a new list, where each element is obtained by adding y to each element of x. For example, for x = [1,2,3] and y = 1, the function should return [2,3,4]. Additionally, if the input values are not a list and an integer, the function should return the message "Wrong Input".

4.2.1 Cell Structure

In the example assignment created, the first cell is a read-only markdown cell. This is used to present the task description. Following the first cell, another read-only cell is used to define example input parameters. In our case this is a list and an integer, the parameters used by *add_ number()*. Those can be used by the student to test his or her solution. Unlike the first cell, this is not a markdown cell but a code cell. The variables or functions defined there will be usable in the subsequent cells of the notebook. Cell (4) is of type "Autograded Answer". Here, the student is expected to write his or her code. As already explained in Chapter 3.1.1, the special syntax seen in this cell replaces the enclosed code with a predefined text like "# Your answer here".

Cell (5) is the first cell used for testing. It implements a visible test, which means that the test code is displayed in the student's version of the assignment. Adding visible tests to assignments is highly recommended, as it provides a way for the students to check their code, and also gain an understanding of how their code is being graded. In the presented approach, the visible test is used to test the basics of the function, like the output type. Additionally, there is an assertion that tests the student's code using the previously defined input parameters. For this cell, the type "Autograded Test" is used, and one point is assigned to it. This means, that if the cell executes without any errors, the total score of the student's assignment will

be increased by one point. The following two cells (6) and (7) implement the in this thesis developed method of auto-grading Jupyter notebooks. The approach allows for a larger number of tests to be executed and graded by using just two cells. One cell executes all the tests and calculates the achieved points, while the other cell is used to add the points to the total notebook score.

```
# import the solutions and the packages
import unittest, sys
%run solutions.py

# inputs to test the function
testlist = [10, 9, 13, 40, 1002, 0, 10]
testnum = -1

# creating the test class //
class Test_Function(unittest.TestCase):
    score = 0  # initialize class-level score

    def setUp(self):
        # used to initialize points for each test case
        self.points = 0

    def tearDown(self):
        # used to add the test's points to the total score
        Test_Function.score += self.points

    def test_negatives(self):
        self.assertCountEqual(
            add_numbers(testlist, testnum),
            add_numbers_sol(testlist, testnum)
        )
        # if the asserts run, the points are assigned
        self.points = 2

    def test_input_control(self):
        self.assertEqual(
            add_numbers(testlist, "word"), "Wrong Input")
        )
        self.assertEqual(add_numbers(2, 2), "Wrong Input",
            msg = "Error when checking  list input"
        )
        self.points = 1  # assigning only if both asserts run

# loading the test suite
test_suite = unittest.TestLoader().loadTestsFromTestCase(
    Test_Function)

# using the text test runner to run the tests
unittest.TextTestRunner(verbosity=2, stream=sys.stdout).run(
    test_suite)

score = Test_Function.score
print(f"Total Points Achieved in this Task: {score}")
```

Listing 1: Testing Code, see cell (6) in Appendix A

4.2.2 Testing Code

The code used for the testing cell (6) is visible in Listing 1. All line references in this text are cross-references to this listing. In the following, the individual code snippets will be explained. After that, an explanation of how the code works is given.

In line 1, the line magic command *%run* is used to execute the .py file containing the solution code for the task (see Appendix C). The .py files have to be located in the same directory as the notebook. The naming of the imported solution is a combination of the name of the function needed for the task together with the suffix "_sol". In our example, we end up with *add_numbers_sol()*. It is recommended to stick to one uniform naming scheme for all solution functions that are included. That way, potential errors can be avoided. In addition to the solution file, the necessary libraries are also imported. In our case, we need the *unittest* and *sys* packages.

For the creation of the tests, the class *Test_Function* is used (cf. lines 10 to 36). The class is a subclass of *unittest.TestCase*. Four methods and one variable are defined within this class, two of which are tests. The variable in line 11 is used to keep track of the points that the student achieved. Important to note is, that only methods that start with the string "test" are recognized as tests. In those tests, assert statements are used. The test method starting in line 21, *test_negatives()*, uses *assertCountEqual()* to compare the output of the student's *add_numbers* function with the output of the reference function *add_number_sol*. This test specifically uses -1 to check if the student's function can handle negative integers. Such tests are necessary to cover all edge cases. In practice, more comprehensive tests would be needed to thoroughly validate the student's solution. The amount of points a test method is worth gets set by the *self.points* variable in line 27. In our example, the value of said variable is 2.

The *test_input_control()* method starting in line 29 uses two standard *assertEqual()* statements. The first one tests if the student handles the wrong input parameters correctly. In detail, instead of a list, a second integer value is passed to the function. According to the task, this should result in the error message "Wrong Input". Thus, the assert tests if the function returns said error message when the input is not valid. Additionally, the optional *msg* parameter will be included in the output if the test fails. In our example, *msg* contains the string "Error when checking list input". The other assertion starting in line 33 is very similar to the first one as it checks the second input parameter. To do so, a list and a string are used as input values. Here, the variable *msg* is not defined, but only to keep the presented code short. In reality, it is advisable to add a descriptive message to each test for better

clarity and debugging. This test method is worth one point, as specified in line 36.

The other two methods, *setUp()* and *tearDown()* in lines 13 to 19, are used to define instructions that are executed before and after each individual test method, respectively. In our case, they are utilized to implement a scoring method, using the *self.points* defined in the individual test cases. Additionally, the *setUp()* method can be used to initialize variables that are used by all tests. This can be seen in example Task 2 (see Appendix A).

After defining the class, a test suite is created using the *TestLoader()* function, which aggregates the test into one suite, so they can be executed as a set. The test suite is then executed using the *TextTestRunner()*. This function is one of the main reasons that the *unittest* package was chosen over other unit test libraries like *pytest*. It provides the option to output the test result using a specific stream. In our case, we use the standard output stream. Otherwise, if the *stream* parameter is not specified, the test results are displayed via the error stream. If the error stream is used, the output of the test runner would look like an error message, which could confuse the students. Another benefit of the library is that it can be customized with ease. For example, alternative packages like *nose2* or *pytest* can be used to run the *unittest* test cases. Also, while not necessary for this approach, a custom test runner could be programmed to further change the output. After the tests have been run, lines 44 and 45 are used to assign and print the calculated score.

Now that the code snippets have been explained, the function of the code during the test execution will be discussed:

1. At first, the *setUp()* method is executed. This happens before each individual test method. The function initializes the variable *self.points* used to set the value of the test method. The value 0 gets assigned to the variable.

2. In the next step, the first test is executed. The order of tests is determined by the built-in ordering of strings. In our example, *test_input_control()* will be executed before *test_negatives()*. Due to alphabetical ordering, the word input comes before the word negatives. As the test method uses two assert methods, it will only be completed if both of the assertions pass. In this case, *self.points* is set to 1. Otherwise, if one or more asserts fails, the whole test will break and the points variable will remain at 0. It is of high importance that the points for the test are always assigned after the assertions. Otherwise, this process does not work as intended. In the case of an AND/OR connection of tests,

the assigning of points works slightly different. This is explained in Chapter 4.2.4.

3. After each test, the *tearDown()* function is executed. In our example, this function is used to add the points from the test to the class-level score variable. The important detail is that if the test fails, the points variable will remain at 0. Only if the test passes will the points variable contain the actual points for the test. This ensures that the score increases only when the previous test is completed without errors.

This process is repeated for all tests in the test suite. After all tests have been executed, the score variable will contain the points of all tests that passed. To add the points to the assignment, a second cell is used, in which the score variable gets called. Nbgrader recognizes this as a score and adds the points to the cell. For best practice, the use of a function like min(), max(), map(), or a combination is recommended. Should the output value of this second cell exceed the specified maximum points of the cell, an error occurs and the auto-grading does not work as intended. To prevent this, *min(score, 3)*, with 3 being the maximum score of this cell, is used in the example assignment (see Appendix C).

In Figure 6 the text output of the test class is shown. It was generated by intentionally using wrong code, to showcase how the output looks if the student made errors. The output always follows the same structure. A double line is used to separate the output into sections. First, the results of all the tests are displayed. In the example, the word "FAIL", which is shown after the name of both tests, indicates that the asserts in both tests have not passed. Otherwise, if the test succeeded, "ok" is displayed next to the name of the test. In the case that the test resulted in an exception, the word "ERROR" is shown. In the second section, we can see that the test for input control failed, because the student did not use a capital "I" in his output. Furthermore, the error message "Error when checking list input" is displayed. This message was given as a parameter in the assert method and can be customized by the instructor. The third section contains information about the test with a negative integer. Here we can see that this test failed, as the first solution contains one 9 while the second solution contains two 9s. Lastly, there is a section with general information. The time needed to run the tests, the number of failed tests, and the achieved points are printed out. This structure allows the instructor to quickly find the output of a test and also the source of a potential error. The information is conveniently displayed so that the teaching staff has everything in one place. If necessary, the output could also be modified to fit the needs of more advanced assignments.

```
test_input_control (__main__.Test_Function) ... FAIL
test_negatives (__main__.Test_Function) ... FAIL

======================================================================
FAIL: test_input_control (__main__.Test_Function)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users\fabeb\AppData\Local\Temp\ipykernel_20768\235534016.py", line 35,
in test_input_control
    self.assertEqual(add_numbers(2, 2), "Wrong Input",
AssertionError: 'Wrong input' != 'Wrong Input'
- Wrong input
?       ^
+ Wrong Input
?       ^
 : Error when checking  list input

======================================================================
FAIL: test_negatives (__main__.Test_Function)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users\fabeb\AppData\Local\Temp\ipykernel_20768\235534016.py", line 24,
in test_negatives
    self.assertCountEqual(
AssertionError: Element counts were not equal:
First has 1, Second has 2:  9

----------------------------------------------------------------------
Ran 2 tests in 0.002s

FAILED (failures=2)
Total Points Achieved in this Task: 0
```

Figure 6: Example Test Output

4.2.3 Assert Types

When using the *unittest* package, various assert methods are available [11]. The standard *assertEqual()* takes two elements as input and tests if the first and second elements are equal. If the type of the two input parameters is the same, then the respective type-specific test is automatically called. Those are for example *assertDictEqual()* or *assertListEqual()*. Similar methods are available for sets, tuples, sequences, and multi-line strings. The *assertCountEqual()* statement found in the test code for the first example task is used to test if two sequences contain the same element, regardless of their order (see Listing 1). Also interesting for assignments is the *assertAlmostEqual()* method, which tests if the numbers used as input parameters are approximately equal. Here, the optional *places* input parameter can be used to specify the number of decimal places to which the two values must be approx-

imately equal. This is particularly useful when dealing with floating-point arithmetic, where small rounding errors can occur. There are also assertions to check a single input parameter, for example *assertTrue()*.

In general, the package provides many assert methods that can be useful for testing programming assignments. Also, most of the assertions have an inverse counterpart, like *assertIsInstance()* and *assertNotIsInstance()*. Furthermore, a custom error message can be displayed if the test fails. By specifying the *msg* parameter, a meaningful error message can be provided, which will be printed when the test does not pass. This aids in quickly identifying and understanding the nature of the failure. More information about all the possible assert statements can be found in the official documentation [11].

In the second task of the example assignment provided in Appendix A, the Python package *pandas* is used. When a task requires the student to return a pandas DataFrame or series object, basic assert methods might not be enough. A function from the *pandas.testing* module can provide the necessary functionality. The *pandas.testing.assert_frame_equal()* function can be used to compare two DataFrames. Additionally, the function offers a lot of parameters that can be used to vary the strictness of the test. For example, specifying *check_column_type = False* would allow the test to pass even if the column types differ between the two DataFrames. It is also possible to compare a pandas series object. This function gets called with *pandas.testing.assert_series_equal()* and has similar parameters compared to the DataFrame test function [24].

4.2.4 AND/OR Connection of Tests

While the individual asserts are certainly useful, instructors might need to combine them. The proposed auto-grading approach allows for multiple ways to connect asserts.

A simple "AND" connection of two or more tests is possible by using multiple assert statements in one test method. This can be seen in the *test_input_control()* function in the first example task. When using two or more asserts, all of them must pass, otherwise the test will fail. This compares to the logic of an "AND" connection. The *self.points* variable should only be specified once, after the last assert in the test method. This way, the points variable is only changed if all asserts pass. This structure can be seen in Appendix A.

To create an "OR" connection, many options are available. Two of which will be explained in detail. For the first one, multiple assertions are used inside of one test method. To connect the test, *try - except* blocks are used.

Should the assert in the *try* block fail, then the assert in the *except* block is executed. The test method will be a success as long as at least one of the assert methods passes. Additionally, more than one *except* clause can be used, and the exception name can also be specified. In this case, the assert in the *except* block gets executed only if the try block results in the specified exception. More information on the abilities of the *try* - *except* blocks can be found in the official Python documentation [10]. When using this "OR" method, it is important to remember to assign the points after each assert. This ensures that the point calculation works as intended.

The second option to create an "OR" connection would be to use two or more test methods. This works well for creating a connection comparable to an exclusive or. For example, imagine a task worth one point that requires the student to return a single object that is either of type list or of type dictionary. In this case, we can create two test methods, one for each data type. Then, we can assign one point to both of these tests. Because only one of the two methods can possibly pass, the maximum score still remains one.

4.2.5 Hiding the Test Cases

In some cases, the teaching staff might want to hide the code used for testing. The special syntax used to create hidden tests has already been discussed in Chapter 3. While this technique can be used to hide the test code in the student's assignment notebook, the test code would still be visible in the feedback file. To completely hide the test code from the notebook, it is necessary to contain the code in a separate *.py* file. Similarly to the *solution.py* file containing the solutions to all tasks, a second file is created. This file (let's call it *testcases.py*) should contain all the test classes needed, as well as the code for the creation of the test suite. Inside the *testcases.py* file, it is important to use unique names for all test classes and test suites. To make the student's solutions available, the package *nbimporter* is used. This allows us to import the .ipynb file and use it as a module [27]. Additionally, the solution file has to be imported. It is recommended to import the files with distinctive names, like the student notebook as "studentnb" and the solution file as "solutions". Should this approach be employed to grade the assignments, it is no longer necessary to name the solutions with the "_sol" suffix, as the functions are now called with *studentnb.functionName()* and *solutions.functionName()*. The test classes in the master copy of the assignment notebook can now be removed. Instead, the line magic command *%run testcases.p*y is used to make all the test suites available. Then, the test suite can be run by using the *.TextTestRunner()* function.

4.3 Comparing the Output

Even if the tests are hidden, the output of those tests is still displayed after running them. When the assert methods fail, the differences between the two compared objects are shown. While this certainly helps to identify errors, it might be advisable to print the complete output of the student's function and the solution, to compare them. The information given through the test output might not always be sufficient for a good comparison.

As part of this thesis, a function was developed with the goal in mind of comparing potentially long outputs like lists of lists and DataFrames (see Appendix B). The function is called *comparing_ outputs()* and takes three mandatory input parameters: a list of input parameters for the functions that get compared, the student function, and the solution function. Additionally, the parameters *column* and *median* can be included to filter and enhance the displayed output. If *column* is set to an integer, then only the column with that index is displayed. For example, if the output is a DataFrame, and *column = 0*, then only the first column (index = 0) would be displayed. Alternatively, if the output is a list of lists, only the first element (at index 0) of each sub-list would be displayed. Additionally, if *column* is set to the string "leftMostNumeric", then the leftmost numeric column gets displayed. If *column* is set to "uniqueNumeric", the column with the most unique numeric values gets displayed. Similarly, if *median* is set to True, the median of the specified column is displayed. All those features are designed to handle lists and DataFrames even when all columns are strings, as it can cast strings that contain numbers to numeric data types. Those features have been included to show how the function can be modified to handle specific tasks. To implement said features, sub-functions have been created that find the corresponding column in the output. At the core, *comparing_ outputs()* tries to capture the output of both the student function and the solution, and then display the the result. The output is generated with the data contained in the input list.

```
try:
    for item in InputList:
        clean_print(f"\nThe student's solution using {item}")
        try:
            student_output = student_function(item)
            print(student_output)
        except Exception as e:
            print("Custom Error Msg")

        clean_print(f"\nOur solution using {item}")
        try:
            solution_output = solution(item)
            print(solution_output)
        except Exception as e:
            print("Custom Error Msg")
except Exception as e:
    print(f"An error occurred during comparison: {str(e)}")
```

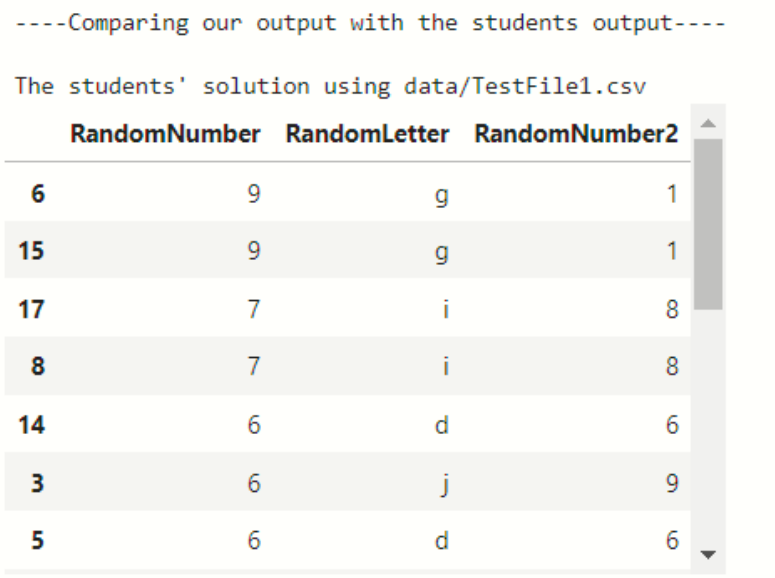
Listing 2: Simplified Code for Comparing Outputs

Listing 2 shows a simplified version of the code snippet responsible for printing the output (see Appendix B for the complete code. Using a for loop nested inside a *try-except* block, the function iterates over each item in the input list. Inside yet another *try-except* block, the current item is then used as an input parameter for the student's function. The result gets assigned to a variable which then gets printed. The same thing happens for the instructor's solution function. Thanks to the elaborate error catching, the displaying of exceptions in the notebooks can be avoided. Instead, a message like "An error occurred during comparison" gets printed.

4.3.1 Formatting the Output

To display the result, the sub-function *clean_print()* is used (see Appendix B, line 100 - 182). The main feature of this function is that it can handle long outputs. To do so, it tests if the length of the result exceeds a predefined value. In that case, the *clean_print()* function uses HTML and CSS to format the output. For DataFrames, the pandas *DataFrame.to_html()* converts the DataFrame into an HTML representation. Then, with the help of the *IPyhton* package, the HTML representation is displayed. By employing CSS code, the DataFrame can be displayed with a scroll bar. This way, no matter how long the DataFrame is, it will always occupy the same amount of space in the notebooks. More information about the *IPython* package can be found in the official documentation [37].

An almost identical approach is used for the list of lists. Using a for loop, the list is iterated and an ordered list HTML representation is created. With



	RandomNumber	RandomLetter	RandomNumber2
6	9	g	1
15	9	g	1
17	7	i	8
8	7	i	8
14	6	d	6
3	6	j	9
5	6	d	6

Figure 7: Scroll-able DataFrame Object

the same package, that list can then also be displayed with a scroll bar.

Figure 7 shows an example output. As you can see, the DataFrame is displayed in its standard layout, but with the addition of a scroll bar. The code of the function should be either in a separate *.py* file or together with the solutions in *the solution.py* file. To use the function, it is recommended to call it after the test class has been run. For example, it could be called after the last line in Listing 1. The function could also be called inside an *if* clause, that tests if the achieved score is smaller than the maximum score. In that case, the output gets printed only if the student has some errors in his or her code.

4.4 Best Practice Checklist

Using the present techniques to enable auto-grading can greatly reduce the time spent on manual grading. Although the developed system eliminates flaws of previous implementations, there are still other points that should be considered. To ensure that the improved auto-grading approach is correctly implemented, the following checklist has been created. The list includes best practices and techniques, which should be employed to ensure that the auto grading system works as intended.

1. **Assigning Points After the Assert Method**: To ensure that the testing approach works as intended, it is important to always assign

the points after the assert statements. This can be confusing when multiple "OR" connected asserts are used inside of one test method. In that case, it is not enough to assign the points once in the test method, as it is necessary to follow slightly different guidelines which are described in detail in Chapter 4.2.4.

2. **Covering All Edge Cases**: If the teaching staff intends to minimize manual grading, it is essential to address all potential edge cases in the test suite. While this will likely result in a large number of tests, it is the only option to properly validate the student's solution. In the literature, the term *full path coverage* is used in this context [13]. This concept includes creating tests that exercise every possible execution path within the code. This way, the instructors can trust that the testing code correctly checks the student's solutions.

3. **Making Use of All Asserts**: It is advised to research all the different assert methods available. Many tests can be efficiently created using specialized asserts. Instructors should try to vary the asserts used, instead of always using *assertEqual()*. For example, the *pandas* package provides specialized asserts to test DataFrames and series objects [24].

4. **Structuring Asserts**: To avoid confusion, it is recommended to always compare the student's function to the instructor's solution. This means, that the first element in the assert is the student's function (or its result), and the second element is the solution. This is important due to the fact that the terms first and second or left and right are used in the test output to address the compared objects. An example can be seen in Figure 6. Here, the results are addressed as *First* and *Second*. If the instructors followed this best practice, they can safely assume that *First* references the student's solution.

5. **Providing both Visible and Hidden Tests**: Each task should provide at least one visible test. This ensures that the students can test their code. Also, it helps them to understand the criteria used to evaluate their solution.

6. **Designing Tests to Work Independently**: As already discussed, the order of execution regarding the test methods is subject to an internal sorting mechanism. Thus, the order might not be predictable. Even if the order is predictable, it is still advised that all test methods should work independently of each other. This way, tests can be freely added and deleted. If one test needs variables or functions created in another test, problems can arise when the order of execution is changed.

7. **Utilizing Feedback Messages**: Almost all assert methods that have been described in 4.2.3 offer the option to print out a custom message if the assert fails. This can be a powerful tool to provide feedback to the student. Statements like "Your solution failed, did you try [...]?" or "Your code does not work as intended, are you sure you implemented [...]?" can help the student to quickly identify the mistake. This can be a great addition to the standard output of the test class. Utilizing this feature is especially useful when the instructors already know what kind of errors are likely to happen. The ability to provide meaningful feedback is frequently named when it comes to an effective auto-grading system [23].

8. **Assigning Points**: The score in cells that are used to assign points to the notebook should be called with a special function like *min(score, max_points)* or *max(score, 0)*. Otherwise, errors can occur when the score accidentally has a value lower than zero or higher than the maximum score.

9. **Synchronized Modification**: The assignment consists of multiple parts. The task description, task solution, and the corresponding testing code are all connected. To ensure that changes do not corrupt the assignment, it is essential that whenever one part of the assignment changes, all other parts are simultaneously changed. This practice helps to maintain consistency throughout the assignment.

10. **Implementing Version Control**: When multiple people are working on the same assignment, it is advisable to implement a version control system. While this can be a little bit challenging in Jupyter, there are multiple extensions and third-party software available that provide the necessary functionality [15].

11. **Smart Naming**: Through the assignment and all supporting files, the names of files and functions should be consistent. For example, all test methods should be named after the case that they are covering. Another example would be the names of the solution functions. It is recommended to use a standardized prefix or suffix for those. In general, the teaching staff should stick to one naming convention.

12. **Comparing and Formatting Outputs**: To keep the feedback files tidy, the instructors should avoid plainly printing out the entire output of a function. Instead, a custom function should be used, that enables quick comparison while also limiting the length of the displayed output. A function that can be used for this is shown in Chapter 4.3.

5 Conclusion

Through the development of a new and improved approach to auto-grading Jupyter notebooks, this thesis aimed to improve the reliability, functionality, and efficiency of the evaluation process. Flaws of the existing implementations of *nbgrader* have been identified by researching dedicated forums and through hands-on experience with the tool. Supported by published literature on the topic, a comprehensive list was created of ideal characteristics that any effective approach in this field should possess.

A new approach was developed, with the list of characteristics in mind. The solution builds upon the *nbgrader* extension and combines it with techniques from unit testing. Individual tests are created as methods of a test class. Each test method is worth a certain number of points. Supporting methods in the test class, which are run before and after each test, are used to calculate the achieved score. While all of this happens in one cell in the notebook, a second cell is used to add the points to the total score of the notebook.

The proposed solution brings advantages in all five identified characteristics. The structure allows for easy maintenance of the testing code, and tests can be added, removed, and modified with ease. Due to the usage of the *unittest* library, the created tests can also be run with a variety of test runners. Additionally, the library comes with a great number of assert methods, allowing the creation of tests for complex tasks. Those tasks can be validated with any number of tests, without the need to use more than two cells. Even if some of the tests fail, all of them are executed and the correct number of points is added to the notebook. The already thorough output of the test cells can be further customized, resulting in meaningful feedback for the students. In addition, a function has been created for displaying and comparing the outputs of the student's functions and the corresponding solution. Complex outputs are presented clearly by utilizing HTML to format the data. The new representation can be displayed in the *.ipynb* documents and in the feedback files.

In conclusion, the goal of improving auto-grading was accomplished. The added features can be used to add value for both the instructors and the students. Teaching staff can benefit from the increased reliability and flexibility of the approach. For the students, decreased time between handing in an assignment and receiving feedback, combined with an enhanced quality of feedback could result in a better learning experience.

5.1 Future Work

In future work, developing a method to grade cells containing text answers would be interesting. During the coding process of this thesis, multiple ideas emerged on how this could be accomplished. A simple approach could include keyword testing. The text in the cell is tested through the comparison with a list of words that the ideal answer would contain.

In a more advanced idea, large language models could be used to analyze the text answer. This might be a more versatile solution that could be used to grade more complex answers. On the other hand, instructors must find a way to justify the approach. If an AI is responsible for the grade, the model and the decision process must be comprehensible. This could turn out to be the biggest challenge with this approach.

References

[1] Abdulmalek Al-Gahmi, Yong Zhang, and Hugo Valle. Jupyter in the Classroom: An Experience Report. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education*, pages 425–431, Providence RI USA, February 2022. ACM.

[2] Inc. Anaconda. Anaconda navigator - anaconda documentation. https://docs.anaconda.com/navigator/, 2024. (Accessed on 06/26/2024).

[3] DamiÃ¡n Avila. Usage - rise 5.7.2.dev2. https://rise.readthedocs.io/en/latest/usage.html, 2019. (Accessed on 08/05/2024).

[4] Lorena A. Barba, Laurie J. Barker, David S. Blank, Jed Brown, Allen Downey, Tyler George, Lindsey J. Heagy, Kyle Mandli, Jason K. Moore, David Lippert, Kyle E. Niemeyer, Ruth Watkins, Rachel S. West, Emily Wickes, Carol Willing, and Michael Zingale. *Teaching and Learning with Jupyter*. jupyter4edu.github.io, 2019. [Cited 27.04.2024].

[5] Jeff Brown. Using jupyterhub in the classroom : Setup and lessons learned. *International Journal of Software Engineering & Applications*, 9:1–8, 2018.

[6] Brenda Cheang, Andy Kurnia, Andrew Lim, and Wee-Chong Oon. On automated grading of programming assignments in an academic institution. *Computers Education*, 41(2):121–131, 2003.

[7] Matt Cone. Markdown guide. https://www.markdownguide.org/, 2024. (Accessed on 08/01/2024).

[8] World Wide Web ConsortiumJ. W3c html. https://www.w3.org/html/, 2017. (Accessed on 08/05/2024).

[9] George E. Forsythe and Niklaus Wirth. Automatic grading programs. *Commun. ACM*, 8(5):275 – 278, may 1965.

[10] Python Software Foundation. 3.12.4 documentation. https://docs.python.org/3/, 2024. (Accessed on 06/24/2024).

[11] Python Software Foundation. unittest - unit testing framework - python 3.12.4 documentation. https://docs.python.org/3/library/unittest.html, 2024. (Accessed on 06/20/2024).

[12] Brian E. Granger and Fernando Pérez. Jupyter: Thinking and storytelling with code and data. *Computing in Science Engineering*, 23(2):7 – 14, 2021.

[13] Georgiana Haldeman, Andrew Tjang, Monica Babeş-Vroman, Stephen Bartos, Jay Shah, Danielle Yucht, and Thu D. Nguyen. Providing meaningful feedback for autograding of programming assignments. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, SIGCSE '18, pages 278 – 283, New York, NY, USA, 2018. Association for Computing Machinery.

[14] ECMA International. Ecma-404 - the json data interchange syntax. https://ecma-international.org/publications-and-standards/standards/ecma-404/. (Accessed on 08/05/2024).

[15] Jeremiah W. Johnson. Benefits and Pitfalls of Jupyter Notebooks in the Classroom. In *Proceedings of the 21st Annual Conference on Information Technology Education*, pages 32–37, Virtual Event USA, October 2020. ACM.

[16] 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.

[17] JupyterHub. The littlest jupyterhub. https://github.com/jupyterhub/the-littlest-jupyterhub, 2024. (Accessed on 08/05/2024).

[18] JupyterHub. Zero to jupyterhub with kubernetes. https://github.com/jupyterhub/zero-to-jupyterhub-k8s, 2024. (Accessed on 08/05/2024).

[19] 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.

[20] Derek Land. AUTOMATIC GRADING IN ENGINEERING CLASSES. *The 10th International Conference on Physics Teaching in Engineering Education PTEE 2019*, 2019.

[21] Hamza Manzoor, Amit Naik, Clifford A. Shaffer, Chris North, and Stephen H. Edwards. Auto-Grading Jupyter Notebooks. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, pages 1139–1144, Portland OR USA, February 2020. ACM.

[22] Giuseppe Mecca, Daniele Santoro, Nicola Sileno, et al. Diogene-ct: tools and methodologies for teaching and learning coding. *International Journal of Educational Technology in Higher Education*, 18(12), 2021.

[23] Fatema Nafa, Lakshmidevi Sreeramareddy, Sriharsha Mallapuram, and Paul Moulema. Improving educational outcomes: Developing and assessing grading system (prograder) for programming courses. In Hirohiko Mori and Yumi Asahi, editors, *Human Interface and the Management of Information*, pages 322–342, Cham, 2023. Springer Nature Switzerland.

[24] The pandas development team. pandas documentation - pandas 2.2.2 documentation. https://pandas.pydata.org/docs/index.html, 2024. (Accessed on 06/24/2024).

[25] Project Jupyter Development Team. nbgrader. https://github.com/jupyter/nbgrader. (Accessed on 2024-06-19).

[26] Jonathan Reades. Teaching on jupyter. *REGION The Journal of ERSA*, 7(1):21–34, March 2020.

[27] Gregor Sturm. grst/nbimporter: Import ipython notebooks as modules. https://github.com/grst/nbimporter, 2021. (Accessed on 06/24/2024).

[28] Jupyter Development Team. The jupyter notebook - jupyter notebook 7.2.0b1 documentation. https://jupyter-notebook.readthedocs.io/en/latest/notebook.html, 2015. (Accessed on 04/27/2024).

[29] Jupyter Development Team. Jupyterlab documentation - jupyterlab 4.2.0 documentation. https://jupyterlab.readthedocs.io/en/latest/, 2024. (Accessed on 05/22/2024).

[30] Jupyter Development Team. The jupyterlab interface - jupyterlab 4.2.0 documentation. https://jupyterlab.readthedocs.io/en/latest/user/interface.html, 2024. (Accessed on 05/22/2024).

[31] Jupyter Development Team. nbgrader - nbgrader 0.9.2 documentation. https://nbgrader.readthedocs.io/en/stable/, 2024. (Accessed on 06/19/2024).

[32] Jupyter Development Team. Notebooks - jupyterlab 4.3.0a0 documentation. https://jupyterlab.readthedocs.io/en/latest/user/notebook.html, 2024. (Accessed on 06/14/2024).

[33] Jupyter Development Team. Project jupyter | about us. https://jupyter.org/about, 2024. (Accessed on 05/22/2024).

[34] Jupyter Development Team. Project jupyter | home. https://jupyter.org/, 2024. (Accessed on 04/27/2024).

[35] Jupyter Development Team. Project jupyter | jupyterhub. https://jupyter.org/hub, 2024. (Accessed on 05/22/2024).

[36] LaTeX Project Team. Latex - a document preparation system. https://www.latex-project.org/, 2024. (Accessed on 08/01/2024).

[37] The IPython Development Team. Ipython documentation - ipython 8.25.0 documentation. https://ipython.readthedocs.io/en/stable/, 2024. (Accessed on 06/24/2024).

6 Appendix

6.1 Appendix A: Example Jupyter Notebook

The following pages contain the example notebook referenced in this thesis. The *.ipynb* file was converted to a LaTeX document.

notebook1_source

August 13, 2024

1 Task 1:

Create a function that takes as input a list X and an int Y. It should return a list, where each element Z[i] results from X[i] + Y. If the input is not a list and an int, then return the msg: "Wrong input".

(This is a "Read-only" cell)

[3]:
```
exampleList = [1,2,3,4,5]
exampleNum = 5

#(This is a "Read-only" cell)
```

[4]:
```
def add_numbers(liste, x):

    ### BEGIN SOLUTION
    if type(liste) != list:
        return "Wrong Input"

    if type(x) != int:
        return "Wrong Input"

    output_list = []

    for i in range(0, len(liste)):
        output_list.append(liste[i] + x)

    return output_list
    ### END SOLUTION

#(This is a "Autograded Answer" cell)
```

[5]:
```
# this is a visble test, it uses the by nbgrader recommended nose.tool package
# if the tests complete, this cell result in no output
from nose.tools import assert_equal

# here we test the output type
assert_equal(type(add_numbers(exampleList, exampleNum)), list)
```

```
# here we test the result using the example inputs
assert_equal(add_numbers(exampleList, exampleNum), [6,7,8,9,10])

#(This is a "Read-only" cell)
```

[6]:
```
### BEGIN HIDDEN TESTS

# import the solutions and the packages
%run solutions.py
import unittest
import sys

# inputs to test the function
testlist = [10, 9, 13, 40, 1002, 0, 10]
testnum = -1

# cretaing the test class
class Test_Function(unittest.TestCase):

    score = 0  # initialize class-level score

    # this method is executed before each individual test, and is used to␣
 ↪initialize points and files
    def setUp(self):
        # used to initialize points for each test case
        self.points = 0

    # this method is executed after each individual test, and is used to assign␣
 ↪the points
    def tearDown(self):
        # here we assign the points to the total score
        # if the previous test resulted in an error, the points variable is␣
 ↪still 0
        Test_Function.score += self.points

    def test_negatives(self):
        self.assertCountEqual(
            add_numbers(testlist, testnum),
            add_numbers_sol(testlist, testnum)
        )
        # if the asserts runs, the points are assigned
        self.points = 2

    def test_input_control(self):
        self.assertEqual(
            add_numbers(testlist, "word"), "Wrong Input")
```

```
        self.assertEqual(add_numbers(2, 2), "Wrong Input",
                         msg = "Error when checking  list input")
        self.points = 1  # assigning only if both asserts run

# here the tests cases are added to a suite
test_suite = unittest.TestLoader().loadTestsFromTestCase(Test_Function)

# using the text test runner to run the tests
unittest.TextTestRunner(verbosity=2, stream=sys.stdout).run(test_suite)

#score gets assigned and printed
score = Test_Function.score
print(f"Total Points Achieved in this Task: {score}")

### END HIDDEN TESTS

#(This is a "Autograded test" cell worth 0 points)
```

```
test_input_control (__main__.Test_Function) ... ok
test_negatives (__main__.Test_Function) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.004s

OK
Total Points Achieved in this Task: 3
```

[7]:
```
# In this code cell, we simply output the score that was computed in the cell␣
 ↪above
# this way, nbgrader will assign the score as the points achieved in this test␣
 ↪cell

### BEGIN HIDDEN TEST
min(score, 3)
### END HIDDEN TEST

#(This is a "Autograded test" cell worth 3 points)
```

[7]: 3

2 Task 2:

Create a function that takes as input a CSV file. Your function should sort the CSV according to the leftmost numeric column. The values should be in descending order. Return a pandas df.

(This is a “Read-only” cell)

[9]:
```
def sort_df(x):

    ### BEGIN SOLUTION
    import pandas as pd

    df = pd.read_csv(x)

    numeric_columns = df.select_dtypes(include = "number").columns
    leftmost_numeric_column = numeric_columns[0]

    sorted_df = df.sort_values(by=leftmost_numeric_column, ascending = False)

    return(sorted_df)
    ### END SOLUTION

#(This is a "Autograded answer" cell)
```

[10]:
```
# this is a visble test, it uses the by nbgrader recommended nose.tool package
# if the tests complete, this cell result in no output
from nose.tools import assert_equal
import pandas as pd

# here we specify a filePath
filePath = "data/TestFile1.csv"

# then we test if the output of the function is a DataFrame Object
assert_equal(type(sort_df(filePath)), pd.DataFrame)

#(This is a "Read-only" cell)
```

[11]:
```
### BEGIN HIDDEN TESTS

# import the solutions and the packages
%run solutions.py

import unittest
import sys
from pandas.testing import assert_frame_equal

# creating the test class
class Test_Function(unittest.TestCase):

    score = 0  # initialize class-level score

    # this method is executed before each test
    def setUp(self):
        self.points = 0   # we initiate the points variable for the test
```

```
        #additionally, we add the test files here
        self.testfile = "data/Testfile1.csv"
        self.hiddenfile = "data/Testfile2.csv"
        self.studentfile = "data/Testfile3.csv"

    # this method is executed after each individual test
    # and is used to assign the points
    def tearDown(self):
        # here we assign the points to the total score
        # if the previous test resulted in an error...
        # ...the points variable is still 0
        Test_Function.score += self.points

    def test_basic_function_test(self): #using the test file

        # computing the expected result with the solution in the .py file
        expected_result = sort_df_sol(self.testfile)
        # computing the students result
        result = sort_df(self.testfile)
        # comparing the results with pandas assert_frame_equal
        assert_frame_equal(result, expected_result)
        # the points are only assigned if the assert completes
        self.points = 2

    def test_basic_function_hidden(self): #using the hidden file

        expected_result = sort_df_sol(self.hiddenfile)
        result = sort_df(self.hiddenfile)
        assert_frame_equal(result, expected_result)
        self.points = 1

    def test_basic_function_student(self): #using the student file

        expected_result = sort_df_sol(self.studentfile)
        result = sort_df(self.studentfile)
        assert_frame_equal(result, expected_result)
        self.points = 2

# here the tests cases are added to a suite
test_suite = unittest.TestLoader().loadTestsFromTestCase(Test_Function)

# using the text runner to run the tests
unittest.TextTestRunner(verbosity=2, stream=sys.stdout).run(test_suite)

# score gets assigned and printed
score = Test_Function.score
```

```
print(f"Total Points Achieved in this Task: {score}")

### END HIDDEN TESTS
#(This is a "Autograded test" cell worth 0 points)
```

```
test_basic_function_hidden (__main__.Test_Function) ... ok
test_basic_function_student (__main__.Test_Function) ... ok
test_basic_function_test (__main__.Test_Function) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.039s

OK
Total Points Achieved in this Task: 5
```

[12]:
```
#In this code cell, we simply return the score that was computed in the cell␣
 ↪above
#this way, nbgrader will assign the score as the points achieved in this test␣
 ↪cell

### BEGIN HIDDEN TEST
min(score, 5)
### END HIDDEN TEST

#(This is a "Autograded test" cell worth 5 points)
```

[12]: 5

6.2 Appendix B: Output Formatting Function

```
import pandas as pd
from IPython.display import display, HTML

MAXROWS = 1000 #t his variable is used to limit the number of rows in the output

# this helper function is used to find the median
def findMedian(output):

    output = findLeftMostNumeric(output)
    if isinstance(output, pd.DataFrame):
        return output.median().values[0]
    elif isinstance(output, list):
        return pd.Series(output).median()

# this helper function is used to find the left most numeric column
def findLeftMostNumeric(output):

    # this varible is used to track if the output is of type list
    wasList = False

    # if list, convert to DataFrame
    if isinstance(output, list):
        wasList = True
        output = pd.DataFrame(output)

    # casting the output to numeric if possible
    # with errors = "ignore", fields that cannot be cast to numeric remain the same
    output = output.apply(pd.to_numeric, errors = "ignore")

    # the follwoing for loop is used to find the column
    column_numeric_index = None
    for index, col in enumerate(output.columns):
        if pd.api.types.is_numeric_dtype(output[col]):
            column_numeric_index = index
            break

    # if a column is found, the output is set to the found column
    if column_numeric_index is not None:
        output = output.iloc[:, [column_numeric_index]]
    else:
        return "No numeric column in df!"

```

```
43      # if the output was a list, then it is turned back into a
        list
44      if wasList:
45          output = output.values.tolist()
46
47      return output
48
49  # this helper function is used to find the column with the
        most unique numeric values
50  def findUniqueNumeric(output):
51
52      # used to track if the output was a list
53      wasList = False
54
55      # if list, convert to DataFrame
56      if isinstance(output, list):
57          wasList = True
58          output = pd.DataFrame(output)
59
60      # trying to cast the output to numeric
61      output = output.apply(pd.to_numeric, errors = "ignore")
62
63      # selecting the numeric columns
64      numeric_columns = output.select_dtypes(include=["number"
        ]).columns
65
66      max_unique_count = 0
67      target_column = None
68
69      # this for loop is used to find the actual column
70      for column in numeric_columns:
71          unique_count = output[column].nunique()
72          if unique_count > max_unique_count:
73              max_unique_count = unique_count
74              target_column = column
75
76      # if the column is not found, a string is returned
77      if target_column is None:
78           return "No numeric column in df!"
79
80      # output is changed to the selected column
81      output = output.loc[:, [target_column]]
82
83      # if it was a list, the output is turned back into a list
84      if wasList:
85          output = output.values.tolist()
86
87      return output
88
```

```
89
90
91  # here the compare function is created
92  # it can be called by inputing a list of input values (
        usually CSVs) and two functions that use the input file
93
94  # Additionally, the parameters column and median can be
        specified
95  # with column, the output can be changed to a certain column,
         either a int or a keyword string like "leftMostNumeric"
96  # with median = True, the median of said column gets
        displayed
97  def comparing_output(InputList, student_function, solution,
        column=None, median=False):
98
99      # this function is used to display the output
100     def clean_print(output, column=None):
101
102         # this part is for df
103         if isinstance(output, pd.DataFrame):
104
105             # here, we test if column is specfifed, and if so
        , we get the correct column
106             if column is not None and isinstance(column, int)
        :
107                 output = output.iloc[:, [column]]
108             elif column == "leftMostNumeric":
109                 output = findLeftMostNumeric(output)
110                 # the helper functions return strings if no
        column is found
111                 # this way, if the returned value is a string
        , we return it again
112                 # so it gets displayed
113                 if isinstance(output, str):
114                     return output
115             elif column == "uniqueNumeric":
116                 output = findUniqueNumeric(output)
117                 if isinstance(output, str):
118                     return output
119
120             # here we handle long outputs
121             if len(output) > 10:
122                 # df is turned into a HTML representation,
        with the maxrows set to the pre defined value
123                 html_df = output.to_html(max_rows = MAXROWS,
        notebook = True, table_id = "table")
124
125                 # CSS is used to make the output scrollable
126                 display(HTML("""<style>
```

```
                                #table {max-height: 250px;
overflow-y: scroll; display: inline-block; }
                                </style>"""))
                display(HTML(html_df))

            # if the df is not long, and no column is
specified, just display the normal df
            else:
                display(output)

        # here, the same thing happens for lists
        elif isinstance(output, list):

            # here the column varible is tested again
            # as the helper function can handle lists and df,
 the implementation
            # is very similar to the the df section
            if column is not None and isinstance(column, int)
:
                # if column is an int, we use list
comprehension to find the column
                output = [item[column] for item in output]
            elif column == "leftMostNumeric":
                output = findLeftMostNumeric(output)
                if isinstance(output, str):
                    return output
            elif column == "uniqueNumeric":
                output = findUniqueNumeric(output)
                if isinstance(output, str):
                    return output

            # here we handle long lists
            if len(output) > 10:

                # the list is iterated, and a string is
created that represents the list as HTLM
                # This is done by adding the tags to the
items in the list
                # Also, if the list is longer than MAXROWS,
the for loop breaks
                rowCount = 0
                html_list = "<ol>"
                for item in output:
                    rowCount += 1
                    html_list += f"<li>{item}</li>"
                    if rowCount == MAXROWS:
                        break
                html_list += "</ol>"

```

```
168                 # now we can display the list
169                 display(HTML(html_list))
170 
171                 # here we use CSS again to make the list
    scrollable
172                 display(HTML("""<style>
173                                 ol { max-height: 200px;
    overflow-y: auto; display: inline-block; }
174                                 </style>"""))
175 
176             # if list is short and no column was specified,
    than we just print the list
177             else:
178                 print(output)
179 
180         # if the output is not a list or df, we print the
    output
181         else:
182             print(output)
183 
184     #here we start the comparison
185     #if a column is specified, we print which column is used
    in the comparison
186     text = ""
187     if column is not None:
188         text = f" on column {column}"
189     print(f"\n----Comparing our output with the students
    output{text}----")
190 
191 
192     # the try-except blocks are used to catch errors with
    comparing or with using the input files on the functions
193     try:
194 
195         # in this for loop, the input list is iterated
196         for item in InputList:
197 
198             # first we dsiplay the student's result
199             print(f"\nThe students' solution using {item}")
200 
201             # if a median is specified, we print it
202             if median:
203                 print(f"The median of {column} = {findMedian(
    student_function(item))}")
204 
205             # here we try to use the student function with
    the current element of inputList, and then display the
    result
206             try:
```

```
                student_output = student_function(item)
                clean_print(student_output, column)
            except Exception as e:
                print(f"An error occurred while using '{student_function.__name__}' with input '{item}': {str(e)}")

            # same thing happens for the solutino function
            print(f"\nOur solution using {item}")

            if median:
                print(f"The median of {column} = {findMedian(solution(item))}")

            try:
                solution_output = solution(item)
                clean_print(solution_output, column)
            except Exception as e:
                print(f"An error occurred while using '{solution.__name__}' with input '{item}' : {str(e)}")

     except Exception as e:
         print(f"An error occurred during comparison: {str(e)}")

# example usage:
# comparing_output(["data/file1.csv", "data/file1.csv"], student_function, solution_function, column="leftMostNumeric", median=True)
```

6.3 Appendix C: solutions.py File

```
def add_numbers_sol(liste, x):

    if type(liste) != list:
        return "Wrong Input"

    if type(x) != int:
        return "Wrong Input"

    output_list = []

    for i in range(0, len(liste)):
        output_list.append(liste[i] + x)

    return output_list

def sort_df_sol(x):
    import pandas as pd

    df = pd.read_csv(x)

    numeric_columns = df.select_dtypes(include='number').
    columns
    leftmost_numeric_column = numeric_columns[0]

    sorted_df = df.sort_values(by=leftmost_numeric_column,
    ascending = False)

    return(sorted_df)
```