JPTest - Grading Data Science Exercises in Jupyter Made Short, Fast and Scalable

Eric Tröbs[1] Stefan Hagedorn[2] Kai-Uwe Sattler[3]

Abstract: Jupyter Notebook is not only a popular tool for publishing data science results, but can also be used for the interactive explanation of teaching content as well as the supervised work on exercises. In order to give students feedback on their solutions, it is necessary to check and evaluate the submitted work. To exploit the possibilities of remote learning as well as to reduce the work needed to evaluate submissions, we present a flexible and efficient framework. It enables automated checking of notebooks for completeness and syntactic correctness as well as fine-grained evaluation of submitted tasks. The framework comes with a high level of parallelization, isolation and a short and efficient API.

Keywords: Jupyter; Teaching; Exercising; Unit-Testing; Automation

1 Motivation

Teaching programming languages, SQL or data science related concepts often involves exercises in which students have to solve tasks by writing programs and queries on their own. Often, these exercises need to be evaluated and graded by some faculty member. However, with hundreds of students, the grading process quickly becomes a burden and often leads to the fact that only a sample is checked or that the number of tasks for the students is reduced. However, especially the latter is to the disadvantage for the students as they miss the potential of exhaustive examples for practicing with valuable feedback.

Thus, in order to exploit the possibilities of online and remote learning as well as to reduce the burden of manually coding related tasks, our goal is to provide an extensive framework to distribute and evaluate programming tasks. Especially for data analytic tasks Jupyter Notebooks[4] have become very popular as they allow to mix formatted text with executable code. Notebooks are also useful for teaching purposes as they allow to show descriptions and tasks within the same file. In our case, the notebooks are used in the context of a data science lecture which contains exercises after every chapter to repeat what has been learned.

In order to give students feedback on their solutions, it is necessary to check and evaluate the submitted work. In our case, this is compounded by the fact that not all assignments

[1] Technische Universität Ilmenau, Germany, Eric.Troebs@tu-ilmenau.de
[2] Technische Universität Ilmenau, Germany, Stefan.Hagedorn@tu-ilmenau.de
[3] Technische Universität Ilmenau, Germany, kus@tu-ilmenau.de
[4] `https://jupyter.org/`

doi:10.18420/BTW2023-37

are submitted at the same time and thus there is no practice effect on those who evaluate solutions. At the same time, multiple attempts might be allowed and we use tasks that are complicated to grade manually, for example, when a lot of `if`-statements need to be used.

In this paper, we present a flexible and efficient framework, called *JPTest*[5], to automatically evaluate and grade code from Jupyter notebooks. The main goal in developing JPTest was therefore to automate the evaluation of coding tasks, while creating a tool that can also check notebooks for completeness and syntax errors. The focus during development was on fast execution through parallelization, isolated execution of student code and an efficient interface. Most of our tasks can be evaluated by classical unit testing of single functions or by comparing manipulated data sets with those of a sample solution. To shorten the process with these task types, annotations exist to express recurring parts of the tests.

2 Related Work

Although the lockdown of schools and universities has drastically increased the need for online learning formats, especially in computer science related lectures, various automated solutions have been created and used for years. However, these are often self-implemented solutions, that are not available to other groups or lack features important for grading. During the peak of the lockdown-induced remote learning phase, the database community presented some of their solutions and experiences with remote learning approaches in the *Datenbank Spektrum* journal.

First among these is SQLValidator by Obionwu et al., where students can easily submit queries to a prepared database and receive detailed feedback and explanations of mistakes they encountered. SQLValidator also includes the possibility to create questionnaires and test students automatically. Even though the authors report positive effects on their courses, the software is limited to SQL [Obi+21].

The second example we would like to mention is a Data Engineering course for 10,000 participants by Alder et al. Based on the openHPI platform of the Hasso Plattner Institute in Potsdam, a so-called *Massive Open Online Course* is offered, which includes lectures supplemented by videos, homework and exams. According to the number of participants, evaluation by hand is nearly impossible. Automated correction was made possible by the use of multiple-choice and multiple-answer questions [Ald+20].

Beyond these teaching related approaches and because Jupyter is widely used not only for prototyping and ad-hoc analytics, there also exist test frameworks for notebooks.

papermill[6] is a project to parameterize notebooks. It works by evaluating tags and allows modifying, storing, inspecting and running notebooks with different sets of parameters.

[5] `https://github.com/erictroebs/jptest`
[6] `https://github.com/nteract/papermill`

Although not directly related to our work, it might be a valuable alternative to create similar reports with different values from a single notebook file.

nbgrader[7] is an integrated solution for grading Jupyter Notebooks. It can be fully operated via a graphical interface and also allows mixing manually and automatically graded tasks. It also allows generating student versions of an assignment. In contrast to nbgrader, JPTest completely detaches tests from notebooks, allows runs to test syntax and completeness, and does not require any plugins in Jupyter. In addition, JPTest allows more freedom in the design of the test code, for example through setup and teardown methods.

3 System Description

JPTest is an unit testing framework for Jupyter Notebooks created with the needs of our data science lecture in mind. It uses *nbclient*[8] as a base for executing code in notebooks. nbclient was originally created for running notebooks to get the output of the cells and process it, for example, for conversion to other formats. As in Jupyter Notebook, a kernel is necessary for the execution of each code cell. It is not strictly necessary to start a separate kernel for each notebook. However, JPTest does just that. From the process that was started to execute the tests, at least a single kernel is started for each test. Each kernel is executed in a separate process so that proper multiprocessing is possible across all tasks, while the test process acts as a coordinator. JPTest always runs on an in-memory copy of the notebook and does not modify files, but tests and code in the notebook still have the possibility to do so.

In summary, there is a process in which the test code runs and from which kernels are started. We refer to this coordinator as the test context. Since our tests contain parts of the solution, it is important that they are managed and stored separately from the notebooks. Each test has exactly one function, which is identified by an annotation. Multiple tests can be collected in one or more files and run together, adding up the scores and collecting the comments. On the other hand, a separate Python process exists for each kernel, which we refer to both individually and as a set of all these processes as the notebook context. Figure 1 visualizes the relationship and communication structure between the created processes.

For communication between contexts the default implementation *jupyter_client* is used, resulting in the use of ZeroMQ. Pickle is used to serialize the objects to be transferred, which, in contrast to JSON for example, can also serialize more complex objects such as NumPy Arrays or Pandas DataFrames. Code written for the test context relies heavily on the asyncio framework to exploit this multiprocessing environment.

The easiest way to execute code in the notebook is via the *cells* property. It returns a list of all cells present in the notebook and allows to filter and execute them one by one. The function

[7] `https://github.com/jupyter/nbgrader`
[8] `https://github.com/jupyter/nbclient`

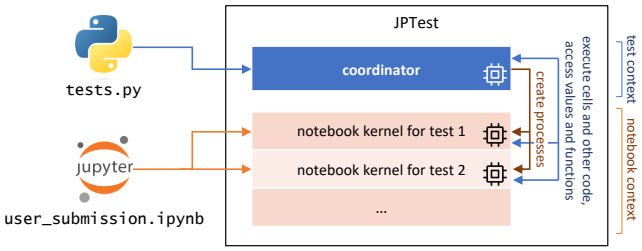

Fig. 1: The coordinator loads the tests and creates at least one kernel in a separate process for each test. The kernel processes communicate only with the coordinator.

execute_cells represents a shortcut to select only code cells by tags prior to executing them in their order of appearance.

Besides executing single cells, it is also possible to interact with objects and code in the notebook context. The most important class in this regard is *NotebookReference*. References returned, for example, by the *ref* and *get* functions, represent objects in the notebook context and may be used for interaction in various ways. For example, they can be serialized and transferred to the test context. However, it is also possible to create sub-references to attributes or keys. References to functions can also be called, where the parameters can be either other references or local variables. In the latter case, these are serialized and sent to the notebook kernel before being called.

The result of almost all operations is delayed. This prevents the need for nesting of *await* statements, what really enhances the readability, and improves the performance by less inter-process communication. Only with a call to *receive* or *execute* the final statement is built and executed in the notebook context, which can cause errors to occur later than their actual call.

Furthermore, functions in the notebook context may be replaced with others, for example to skip network requests and return a fixed response instead to speed them up. Even more interesting is the monitoring of functions, where calls with their parameters and return values are tracked. This makes it possible, for example, to determine whether a user's implementation is using recursion. The available context managers provide the option of replacing functions only for specific statements.

It is also possible to inject own code from the test context into the notebook context. The most simple way is to use a string that is inserted as a new cell at the end of the notebook and executed in the notebook context. However, there are also two methods to inject functions: The first transfers a function to the notebook context and returns a reference. This can be called as described before or passed as a parameter to another function. The second copies the body of a function into a new code cell and executes it. The header is used exclusively in the test context. We use this functionality to write syntactically correct code with all the

user_submission.ipynb

```
[ ]: # task-1
     def fib(n):
         if n <= 1:
             return n
         else:
             return fib(n-1) + fib(n-2)
```

tests.py

```
1  from jptest2 import *
2
3  @JPTest('Task 1', max_score=1)
4  async def test_task1(nb: Notebook):
5      await nb.execute_cells('task-1')
6
7      fib_fun = nb.ref('fib')
8      yield (
9          await fib_fun(5).receive() == 5,
10         1,
11         'better luck next time',
12         'very good'
13     )
```

Fig. 2: Left: A student's submitted fibonacci function. Right: A unit test that executes all cells with the tag `task-1`, creates a reference to the `fib` function and awards one point if a call to this function with the parameter `5` returns `5`.

benefits of analysis within an IDE, although it is later only executed in the notebook context, which is a massive advantage over writing code as a string. The parameters in the header define the necessary variables that are present in the notebook through the execution of previous cells.

To register tests different annotations are used. They are available to either prepare one or two notebooks or to run specific cells and copy single variables into the test context. A maximum number of points can be set as well as an execution timeout. This reduces the writing of tests to as less code as possible. The assignment of points is done using the `yield` keyword. The test function then works as a generator, where each returned value is understood as a part of the score. The value consists of a tuple containing a condition, a score to be awarded when it is met and optional comments on success or failure. Figure 2 shows a basic unit test which uses this concept.

Last but not least, it is possible to connect other kernels, but Python-specific features are lost in this process. Currently SQLite and DuckDB are partly supported.

4 Best Practices

Regarding references, there is also a way to exchange values between notebooks. The function *store* can be used to store values, but also references into notebooks. References can also be used as parameters to call functions within the notebook. If the reference is from the notebook where it should be used, this operation is trivial. If it is from another notebook, it automatically is copied to the former. However, copying objects across notebooks should be avoided where possible.

In general, the test context can become a bottleneck because it uses only one thread. To use the performance of multiple cores, the notebooks should work as independently as possible and the test context should only be used for coordination and evaluation. For example, when we evaluate manipulated DataFrames, one notebook runs the student's solution and one runs

the sample solution. Both resulting DataFrames are copied to the test context and checked there only for equality, so that the computationally intensive operations are outsourced.

In the case of data loading, we use setup functions to modify the data set before starting the tests. Depending on the task, only a portion of all data is selected or the data set is modified. This ensures that the student's code actually solves the problem in general, rather than exclusively providing the answer for the given data. A reduced data set can also speed up the execution.

The simplest test possible is the one where no test file is provided. In this case JPTest loads a default implementation that executes all cells once in the correct order, does not score and passes exceptions. This can be used to check notebooks for syntax errors, determine if libraries are missing within an image or if data sets have not been shipped.

Usually we use Docker to create a reproducible environment for our tests. All necessary dependencies are installed in the image, while required data sets are mounted read-only. By testing a notebook within the container after it has been modified, we can determine whether the image does actually include all the required dependencies. In addition, the containers operate without an internet connection, which creates an isolated environment for each user's notebook.

5 Demo Contents

We provide several Jupyter Notebooks and data sets to create and change unit tests using a set of tasks from our data science lecture, which mainly relies on Python. This includes submissions where functions are tested by simple unit tests as well as comparisons of objects with the results from sample solutions. In addition, we show how data sets can be modified before testing to reject hard-coded solutions and how function replacements can speed up execution times. We also handle cases where values found experimentally by our students have to be taken into account.

At the same time, using asyncio, the communication between tests and the concurrently running notebooks will be explained further. Furthermore, we include test that make use of the API to grade common tasks in just a few lines of code. As there is no way to avoid explaining and using database systems in our lectures, we take a look at the use of external services and how we try to replace them using embedded software.

Last but not least we show the use of the test framework with other kernels, so that SQL statements, for example, can be graded directly.

Acknowledgements. This work was partially funded by the German Federal Ministry of Education and Research under grant no. 16DHBKI085.

References

[Ald+20] Nicolas Alder et al. “Ein Data Engineering Kurs für 10.000 Teilnehmer”. In: *Datenbank-Spektrum* 21.1 (2020), pp. 5–9.

[Obi+21] Victor Obionwu et al. “SQLValidator - An Online Student Playground to Learn SQL”. In: *Datenbank-Spektrum* 21.2 (2021), pp. 73–81.