REPORT ON OpenDreamKit DELIVERABLE D4.8

Facilities for running notebooks as verification tests

MARTIN SANDVE ALNÆS & HANS FANGOHR & VIDAR FAUSKE & THOMAS KLUYVER & BENJAMIN RAGAN-KELLEY & MORE?

Due on	02/28/2017 (Month 18)
Delivered on	02/27/2017
Lead	Simula Research Laboratory (Simula)
Progress on and finalization of this deliverable has been tracked publicly at: `https://github.com/OpenDreamKit/OpenDreamKit/issues/98`	

DELIVERABLE DESCRIPTION, AS TAKEN FROM GITHUB ISSUE #98 ON 2017-02-27

- **WP4:** User Interfaces
- **Lead Institution:** Simula Research Laboratory
- **Due:** 2017-02-28 (month 18)
- **Nature:** Other
- **Task:** T4.3 (#71): Reproducible notebooks
- **Proposal:** p. 48
- **Final report**

The Jupyter Notebook is a web application that enables the creation and sharing of executable documents that contains live code, equations, visualizations and explanatory text. Thanks to a modular design, Jupyter can be used with any computational system that provides a so-called *Jupyter kernel* implementing the *Jupyter messaging protocol* to communicate with the notebook. OpenDreamKit therefore promotes the Jupyter notebook as user interface of choice, in particular since it is particularly suitable for building modular web based Virtual Research Environments.

This deliverable aims at enabling testing of Jupyter notebooks, with a good balance of convenience and configurability to address the range of possible ways to validate noteboooks. Testing is integral to ODK's goals of enabling reproducible practices in computational math and science, and this work enables validating notebooks as documentation and communication products, extending the scope of testing beyond traditional software.

Accomplishments:

- ✓ Develop nbval package for testing notebooks;
- ✓ Allow multiple testing modes, ranging from lax error-checking to strict output comparison;
- ✓ Enable normalizing output for comparison of transient values such as memory addresses and dates;
- ✓ Integrate D4.6 (#95: nbdime) for displaying changes between notebooks when they differ.

CONTENTS

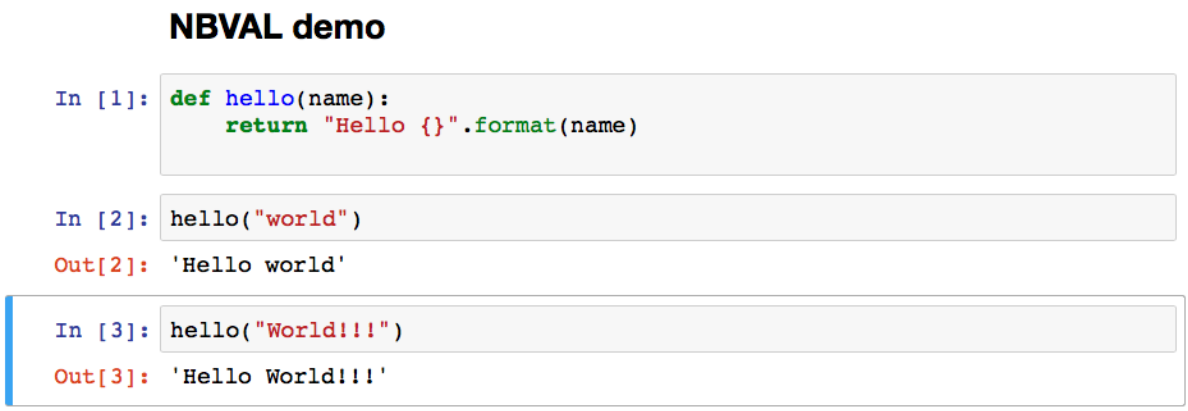


FIGURE 1. Self-contained JUPYTER Notebook demonstrating the concepts of cells that contain different types of material and can be executed (or updated) in arbitrary or sequential order.

1. BACKGROUND

1.1. Jupyter notebook

The Jupyter Notebook is a web application that enables the creation and sharing of executable documents that contain live code, equations, visualizations and explanatory text.

Thanks to a modular design, Jupyter notebooks can be used with any computational system that provides a so-called Jupyter kernel implementing the Jupyter messaging protocol to communicate with the notebook. OpenDreamKit therefore promotes the Jupyter notebook as user interface of choice, in particular since it is particularly suitable for building modular web based Virtual Research Environments.

An example Jupyter notebook (using a Python kernel) is shown in Figure 1. Once the calculation, exploration and annotation is completed, the document can be saved to disk as a (json) file. This notebook can be opened again later at the same or different computers, and will display input and output data as visible in the figure. Furthermore, if a Python kernel is installed, the notebook can be executed again - thus providing an executable document, that can integrate text, equations, code, output from the code, and graphs and other multimedia objects.

The very nature of the notebook enables and encourages reproducible science, contributing to our overall ambition of enabling better science through better electronic infrastructure: researchers can carry out a computation within the notebook, and all steps are automatically recorded and can be saved at the press of a button. While some editing (and maybe removing avenues of computational exploration that were unsuccessful from the notebook) is necessary before the notebook can be shared with others effectively, this is still a significant advancement over managing snippets of code, post processing scripts, figures and (latex) manuscripts in different files and with no explicitly documented dependency.

We see Jupyter notebooks being used in a number of use cases, including

- computational exploration
- documentation of software packages
- tutorials for software
- teaching materials in higher education

In this deliverable, we enrich the ecosystem of Jupyter notebook tools with the NoteBook VALidation tool (`nbval`), that allows to check the types of documents listed above *automatically* for they validity.

1.2. **Testing**

It is good practice to test computer code through so-called unit tests: for every functionality a code provides (through a class, a method of a class, a function, etc), the developers write additional code that tests each unit of functionality. It is key that these unit tests can be executed *automatically*. Once this set up is provided, it is easy for the developer to quickly execute all tests to check the state of the code base: if the tests all "pass" (this is the best possible outcome), then there is confidence that the code is working correctly (at least for the functionality that is covered by the unit tests). On the other hand, if one or more tests "fail", then something is not right with the code.

Having a suite of unit tests available that can be executed automatically greatly assists the development process: developers can change code and be reasonably confident not to have broken other parts of the functionality if the test suite passes without fails after the modification.

The typical structure of a unit test is that the code to be tested is called with some given input data, and that the code to be tested returns some output data for that input data. The test code then compares input and output data: if they are the same (or for numerical tests within acceptable deviation), this unit test is reported at a pass.

The Software Engineering domain uses terms unit test, integration test and system tests (and others) to classify tests. Integration and system tests test more than just one unit of code, but are concerned with multiple units working together when being integrated, or combining multiple integrated units to form the entire production system.

A test suite also increases confidence in (research) software: many (good) software packages will allow users to run the test suite after installing the code on their own system. This is a convenient way to check that the code works correctly, and that - for example - no unexpected incompatibilities exist with installed third party libraries, which may change from installation to installation.

Reasons for test failures of a software tool include (i) developers modifying the actual source code and introducing errors inadvertently, (ii) changes or incompatibilities with other libraries that the tool needs, (iii) changes in data that is used as input for the software.

1.3. **Testing and validation of Jupyter Notebooks**

Jupyter notebook documents have become an important part of the development and communication of computational ideas. When research results of researcher A are communicated by a notebook and researcher B wants to extend them, the first thing that B will need to do is no re-execute the notebook on their computer to check that all the previous results can be reproduced. This can be done by manually executing all cells in the notebook, or re-running the whole notebook, and then checking if any errors have occurred. If the re-execution fails, one needs to test if the notebook was working on researcher A's setup, and what is different in terms of libraries being used etc.

These are known problems for conventional computer code, which are routinely address by having test suites, and executing the tests suites automatically after every code change (known as "continuous integration").

However, the existing testing frameworks do not support notebooks. Further, Jupyter notebooks aim to be a tool for enabling reproducible computation and communication, and the ability to validate and verify the contents of notebooks is critical to that goal. For these reasons, it is important that notebooks can be tested efficiently, so that authors and readers alike can automatically and effectively verify that the notebooks contents remain valid.

2. VALIDATING NOTEBOOKS WITH NBVAL

2.1. Introduction

Jupyter Notebooks are executable documents that transfer sets of input code into output, possibly using data files in the process. We can thus use the software engineering experience of automatic testing (Section 1.2) to establish if output data in the jupyter notebook is compatible with the input cells that have triggered the computation of the data.

The nature of notebooks presents different opportunities and challenges from conventional code testing. Notebooks are narrative documents, and their code is often not arranged in functions and classes, which normally form the units of code to be tested. However, because outputs are stored in the notebook, output from running the code can be compared against previously saved output.

In contrast to unit-test based testing, where the test code has to be written in addition to the production code, we can use every pair of input output cell as a test case: the input cell contains the code to execute as part of the test. The computed output is then compared to the output stored on disk for that cell. If the outputs agree, the test passes, otherwise it fails.

We have developed the `nbval` tool that VALidates a given NoteBook. *`nbval` validates a saved notebook in the sense that stored input cells produce output cells that are identical to the output cell data saved in the notebook.*

2.2. Example 1

We use the trivial notebook in Figure 2 as an example.

NBVAL demo

```
In [1]: def hello(name):
            return "Hello {}".format(name)

In [2]: hello("world")
Out[2]: 'Hello world'

In [3]: hello("World!!!")
Out[3]: 'Hello World!!!'
```

FIGURE 2. Trivial notebook `demo.ipynb`. The code in cell [1] is representative of the computational package we use (typically accessed via the `import` statement).

Once the `nbval` tool is installed (Section 2.4), we can use the `py.test --nbval demo.ipynb` command to validate the notebook with name `demo.ipynb`. Figure 3 shows the output from the testing process.

2.3. Example 2

We show another example where the code computes a different output every time the notebook is run. This could be due to a change in the software that is imported into the notebook, and might be something that we want to report as an error. (There are other cases where we like to ignore changing output - this is discussed in more detail in Section 3.3)

We use the notebook in Figure 4 as an example.

```
minion:~/D/O/W/D/examples$ py.test --nbval -v demo.ipynb
============================= test session starts ==============================
platform linux -- Python 3.5.2, pytest-3.0.3, py-1.4.31, pluggy-0.4.0 -- /usr/bi
n/python3
cachedir: .cache
rootdir: /home/takluyver/Documents/OpenDreamKit/WP4/D4.8/examples, inifile:
plugins: nbval-0.3.6, xonsh-0.5.2
collected 3 items

demo.ipynb::Cell 1 PASSED
demo.ipynb::Cell 2 PASSED
demo.ipynb::Cell 3 PASSED

========================== 3 passed in 2.04 seconds ===========================
```

FIGURE 3. `nbval` Command and output from validating notebook `demo.ipynb` with `nbval` . For each of the 3 cells in the notebook, the input code is re-executed, and any output compared with the output stored in the notebook file (see Figure 2 for the notebook file).

NBVAL demo with failure

```
In [1]: def hello(name):
            return "Hello {}".format(name)

In [2]: hello("world")
Out[2]: 'Hello world'

In [3]: hello("World!!!")
Out[3]: 'Hello World!!!'
```

In `--nbval` mode: normally detect changes in output

```
In [4]: import datetime
```

The command in the next cell will produce different outputs everytime the notebook executes (as the time keeps changing).

```
In [5]: datetime.datetime.now()
Out[5]: datetime.datetime(2017, 2, 13, 10, 16, 15, 240125)
```

FIGURE 4. Trivial notebook `demo-with-fail.ipynb`. The code in cell [5] produces a different output every time the notebook executes as it reports the current date and time.

As before, we trigger validation of the notebook with `py.test --nbval demo-with-fail.ipynb`. Figure 5 shows the output from the testing process.

We can see that cells [1] to [5] pass tests, but cell [7] fails the test. The tool provides a detailed breakdown on what the output for cell [7] is in the stored file, and contrasts it with the output from the re-execution that has just taken place.

```
minion:~/D/O/W/D/examples$ py.test --nbval -v demo-with-fail.ipynb
============================= test session starts ==============================
platform linux -- Python 3.5.2, pytest-3.0.3, py-1.4.31, pluggy-0.4.0 -- /usr/bi
n/python3
cachedir: .cache
rootdir: /home/takluyver/Documents/OpenDreamKit/WP4/D4.8/examples, inifile:
plugins: nbval-0.3.6, xonsh-0.5.2
collected 5 items

demo-with-fail.ipynb::Cell 0 PASSED
demo-with-fail.ipynb::Cell 1 PASSED
demo-with-fail.ipynb::Cell 2 PASSED
demo-with-fail.ipynb::Cell 3 PASSED
demo-with-fail.ipynb::Cell 4 FAILED

=================================== FAILURES ===================================
_____________________________________ cell 4 _____________________________________
Notebook cell execution failed
Cell 4: Cell outputs differ

Input:
datetime.datetime.now()

Traceback: mismatch 'text/plain'
<<<<<<<<<<<< Reference output from ipynb file:
datetime.datetime(2017, 2, 13, 10, 16, 15, 240125)
============ disagrees with newly computed (test) output:
datetime.datetime(2017, 2, 22, 10, 49, 6, 157866)
>>>>>>>>>>>>
====================== 1 failed, 4 passed in 2.03 seconds ======================
```

FIGURE 5. `nbval` Command and output from validating notebook `demo-with-fail.ipynb` with `nbval` . Cells [1], [2], [3], and [5] are code cells and produce output (although it could be `None`), and pass their tests. Cell [7] fails the test. The tool provides a detailed breakdown on what the output for cell [7] is in the stored file, and contrasts it with the output from the re-execution that has just taken place (see Figure 4 for the notebook file).

2.4. Installation

The installation of the package is described as part of the documentation available on Github (https://github.com/computationalmodelling/nbval, in particular https://github.com/computationalmodelling/nbval/blob/master/README.md).

Installation via pip in supported, which allows to install the package in all Python installations.

2.5. Documentation

The detailed behaviour of the tool, as well as basic installation and use instructions, is described in the online documentation (https://github.com/computationalmodelling/nbval/blob/master/documentation.ipynb).

2.6. Testing and continuous integration

We have test suite with about 30 tests, that are executed automatically using the Travis CI continuous integration service (https://travis-ci.org/computationalmodelling/nbval.)

We are testing for successful completion of the test suite for Python versions 2.7, 3.5, 3.5, 3.6 and the nightly build of Python.

2.7. Support

`nbval` support can be requested via GitHub Issues ().

2.8. License

`nbval` is made freely available to all, under the OSI-Approved Berkeley Software Distribution (BSD) License.

3. Design

3.1. Overview

`nbval` (`https://github.com/computationalmodelling/nbval`) is a new tool for testing notebooks, built as a plugin for the pytest testing framework (`http://pytest.org`). By leveraging existing tools, `nbval` fits well into the software development tools such as continuous integration services and testing environments.

When `nbval` encounters a notebook to test, it identifies the language of the notebook from the notebook's metadata and starts a process to run the code found in notebook, called the Kernel. `nbval` communicates with this Kernel via the Jupyter protocol, as in the notebook environment. Each cell in a notebook constitutes a test, and is executed in order, as if the notebook had been executed in the interactive notebook environment.

Unlike traditional source code files, notebooks contain both code to execute and the output. Validating notebooks can take the output into account or not. At its most basic level, called 'lax mode', `nbval` executes a cell, only checking for errors. This verifies that execution of a notebook completes without error, but makes no effort to guarantee that the result is the same across executions.

`nbval` 's default mode is to record the output produced by executing the notebook, and compare it with the output stored in the notebook. At its strictest, any visible change in the output will result in a failed test. Many times, output can contain transient values that are not informative, such as memory addresses or dates. To deal with this, `nbval` provides an extensible mechanism for normalizing output, so that these transient values may be excluded from the comparison (see Section 3.3).

3.2. Implementation

`nbval` uses pytest, which is a popular, extensible testing framework for the Python language. We use pytest because it is a standard tool for running tests in the Python community, and enables extending functionality via a robust plugin system. `nbval` functions as a plugin for pytest, adding the following functionality:

- recognize Jupyter notebooks as files that could contain tests
- construct and run tests based on notebook cells
- report test results based on output, optionally with nbdime

One `nbval` is installed, notebooks can be included in any pytest-based test suite with a single `--nbval` argument. Similarly, collections of notebooks can be tested without any configuration by running `pytest --nbval` in any directory containing notebooks.

3.3. Changing outputs

One of the test features of `nbval` that differs from traditional tests is that notebooks contain the output produced by their previous execution. One of the ways `nbval` tests notebooks is by comparing outputs produced during the tests with those recorded in the notebook. Not all outputs are necessarily useful to test, so `nbval` provides some configuration for deciding which outputs should be tested, and which should get some normalization before comparison.

The general mechanisms for less-than-exact output comparison are:

- mark cell to be ignored
- use regular expressions
- use nbval-lax

The output checking in `nbval` may be controlled on a cell-by-cell basis: notebook authors may either start with no output checking ('lax mode') and mark specific cells whose output should be checked, or start with full output checking and mark cells whose output should be ignored. The author indicate that nbval should not execute a cell at all, or that it an error is the expected result of a cell.

3.4. Exceptional behaviour

There are three ways that a cell run as a test by nbval can result in a test failure:

- The kernel reports that executing the cell contains an error, and the cell is not marked as expected to produce an error.
- The output from running a cell differs from that saved in a notebook, nbval is not instructed to ignore that output, and the changes are not covered by the regular expression patterns used to normalise output.
- Execution of the cell does not complete within a configurable timeout, which by default is 2000 seconds.

In all three cases, the test corresponding to that cell is reported as a failure through pytest, which will result in a description of the error being displayed to the user (see Figure 6).

When the failure is due to a difference in output, `nbval` can also optionally use nbdime to display how the notebook has changed (see Section 5).

When the failure is due to a timeout executing code, execution is stopped. Later cells in the same notebook are marked as 'expected failures', as later code often relies on the completion of earlier code, and reporting several separate failures would obscure the underlying cause.

3.5. Choice of kernel

In the Jupyter protocol, a 'kernel' is the process where code executes, and is a separate process from the one reading the notebook and determining what code should be execute. When kernels are installed on a system, they register a 'kernel spec' that describes how to start the kernel. There can be many such kernels installed on a system, some corresponding to different languages, such as GAP, PARI (D4.7), Sage, or Python, while others may correspond to particular environments. Notebooks record information about the kernel used to run them in metadata. `nbval` can look at this metadata to select the appropriate kernel to run the notebook for tests.

When the notebook's code is Python, Default kernel selection can be bypassed in `nbval` to run the code with the same interpreter with the `--current-env` flag. This makes nbval fit

```
Plotting in the Notebook.ipynb ..FF.F

================================== FAILURES ===================================
_____________________________________ cell 9 _____________________________________
Notebook cell execution failed
Cell 9: Cell outputs differ

Input:
x = np.linspace(0, 3*np.pi, 500)
plt.plot(x, np.sin(x**2))
plt.title('A simple chirp');

Traceback: mismatch 'text/plain'
<<<<<<<<<<<< Reference output from ipynb file:
<matplotlib.figure.Figure at 0x106622080>
============ disagrees with newly computed (test) output:
<matplotlib.figure.Figure at 0x10c8e01d0>
>>>>>>>>>>>>
                                        cell 14
```

FIGURE 6. `nbval` output, showing that output changed.

best in testing frameworks, which often create temporary environments in which to run tests. The test code itself will execute in this environment, but default kernel selection could result in the tests running in a different environment. `--current-env` ensures that not only the test runner, but the test code in the notebooks themselves, is run in the test environment.

3.6. **Numbering of cells**

The Jupyter Notebook numbers code cells, in the order of execution. These can be increasing integers starting from top of the notebook to the bottom, for example if the cells are (written and) executed in this order, or at a later point the user chooses to restart the kernel, and run all cells in order. However, the notebook also allows to execute cells in arbitrary order.

In contrast, With `nbval` cells are always executed in order from start to finish, and labelled with increasing integer numbers starting from 1. This sequential execution enables `nbval` to catch errors that may not have been noticed due to out-of-order execution.

4. NBVAL AND REPRODUCIBILITY

`nbval` facilitates integrating notebooks into a reproducible scientific workflow. Tests are integral to maintaining and verifying software, which is critical for validating and reproducing scientific computation. A publication can include a notebook that produces its results or figures. By using `nbval` , the validation of this notebook and output can be automated, to make it more convenient, and thus more likely, for scientific publications to follow reproducible practices.

5. NBVAL AND NBDIME

Testing notebooks with `nbval` involves comparing the notebook as saved, and the notebook recently run. This is comparing two notebooks, which can build on earlier OpenDreamKit work. `nbval` can use nbdime, produced in D4.6, to display the difference between the before and after notebooks, for more effective comparison and identification of changes.

A graphical display allows humans to quickly take in a lot of information, and intuitively spot visual differences between the two documents, in particular when before and after images of plots, graphs, and figures are shown.

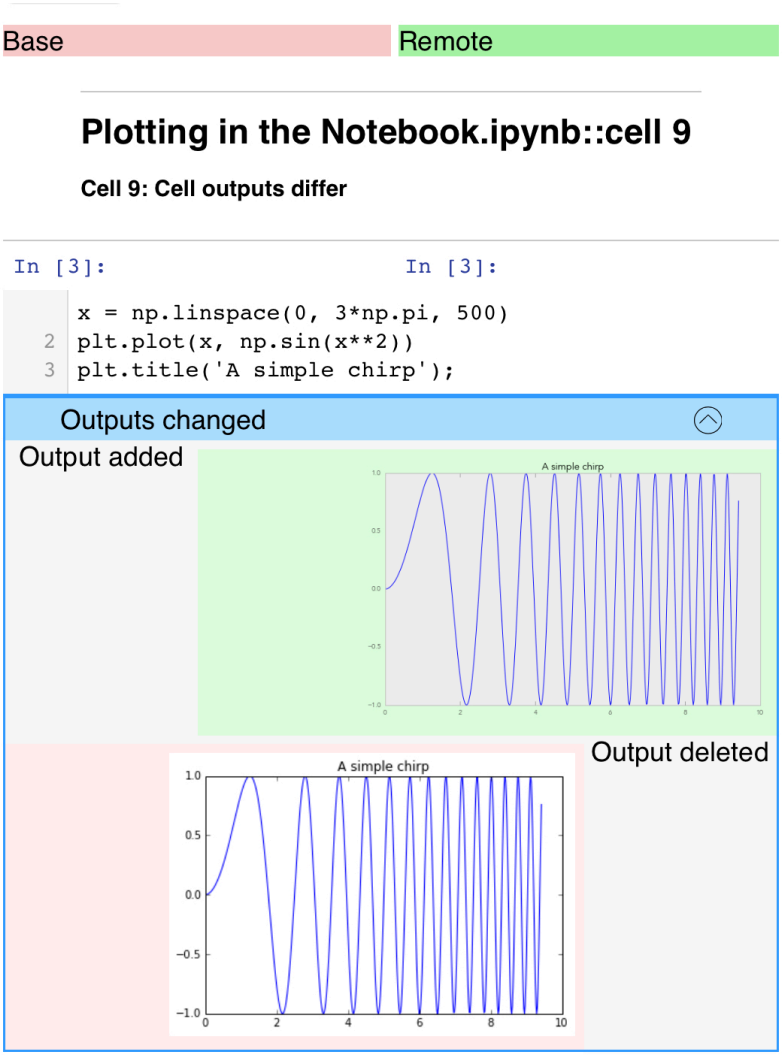


FIGURE 7. An `nbval` output rendered with `nbdime`

6. USAGE REPORTS

As notebooks can be integrated directly into sphinx (and then be hosted on `readthedocs`, for example), they are a convenient way to create documentation for a software package.

Any examples that demonstrate interaction with the code are naturally and efficiently included: the documentation developer just types the command, the note book automatically displays the behaviour.

With `nbval` , we can now include regression testing of the documentation into the continuous integration, and will automatically be alerted if the documented behaviour has changed (and thus the documentation is outdated).

An example can be seen in this repository: https://github.com/joommf/oommfc, which provides the documentation, for example at http://oommfc.readthedocs.io/en/latest/ipynb/new_discretisedfield_interface.html.

Being able to test the documentation *automatically* addresses the common scenario in which documentation is developed for the first version of a tool, but not updated later, and increasingly getting out of date. The package developers are unlikely to notice as they would never read the documentation, in particular not the beginning aimed at new users of the package. With `nbval`

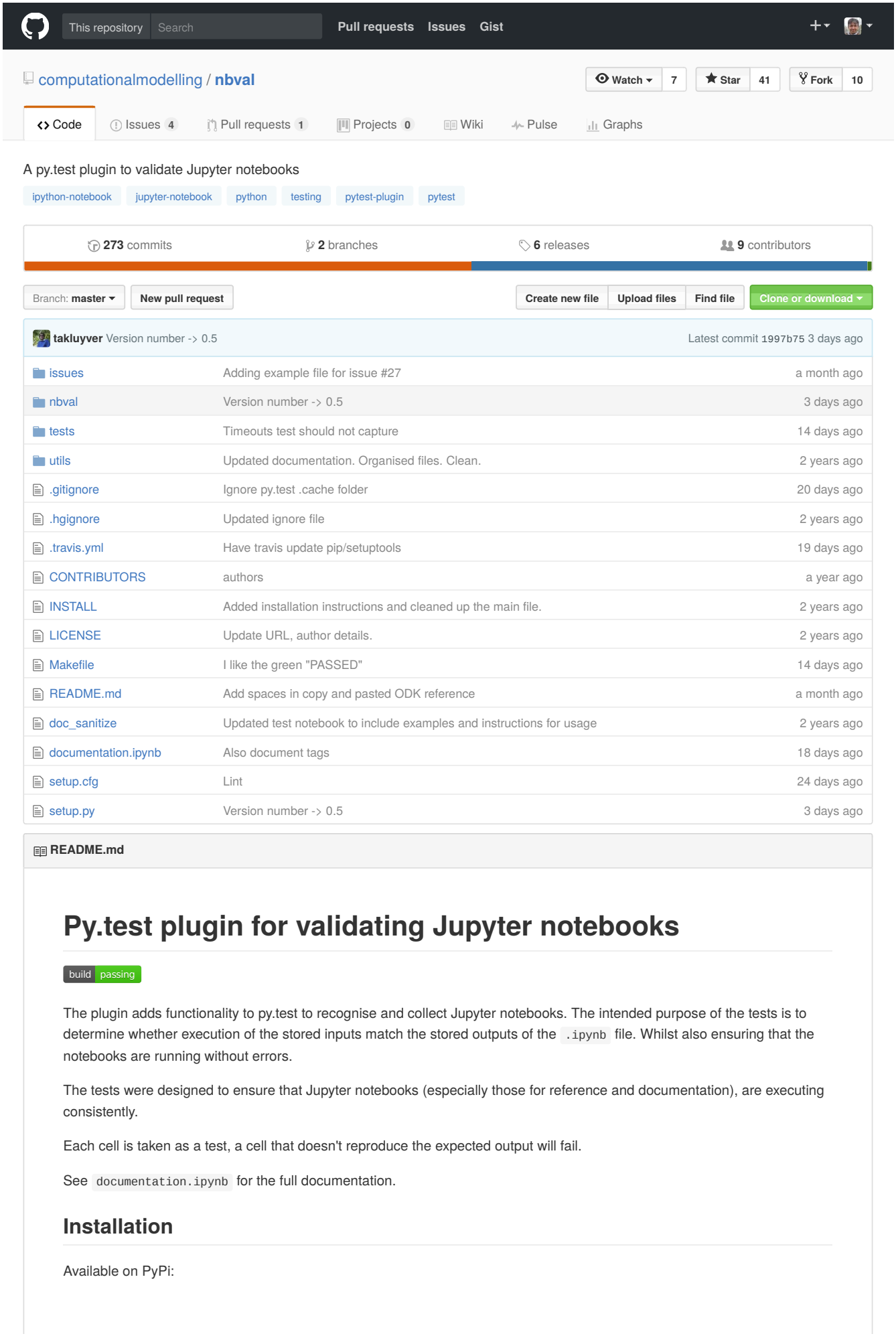
one can integrate the regression testing of the documentation into the continuous integration cycle.

7. Future work

`nbval` has been integrated successfully into some repositories of notebook-based educational materials and software documentation by OpenDreamKit participants, and is being integrated into the existing notebook-based documentation of projects in the wider Jupyter community. We will work to encourage adoption of `nbval` for verifying documentation, and would like to see nbval used to enable verification of scientific publications now that it has proved its effectiveness in educational materials.

APPENDIX A. NBVAL'S HOME PAGE

https://github.com/computationalmodelling/nbval



```
pip install nbval
```

or install the latest version from cloning the repository and running:

```
pip install .
```

from the main directory. To uninstall:

```
pip uninstall nbval
```

How it works

The extension looks through every cell that contains code in an IPython notebook and then the `py.test` system compares the outputs stored in the notebook with the outputs of the cells when they are executed. Thus, the notebook itself is used as a testing function. The output lines when executing the notebook can be sanitized passing an extra option and file, when calling the `py.test` command. This file is a usual configuration file for the `ConfigParser` library.

Regarding the execution, roughly, the script initiates an IPython Kernel with a `shell` and an `iopub` sockets. The `shell` is needed to execute the cells in the notebook (it sends requests to the Kernel) and the `iopub` provides an interface to get the messages from the outputs. The contents of the messages obtained from the Kernel are organised in dictionaries with different information, such as time stamps of executions, cell data types, cell types, the status of the Kernel, username, etc.

In general, the functionality of the IPython notebook system is quite complex, but a detailed explanation of the messages and how the system works, can be found here

http://ipython.org/ipython-doc/stable/development/messaging.html

Execution

To execute this plugin, you need to execute `py.test` with the `nbval` flag to differentiate the testing from the usual python files:

```
py.test --nbval
```

You can also specify `--nbval-lax`, which runs notebooks and checks for errors, but only compares the outputs of cells with a `#NBVAL_CHECK_OUTPUT` marker comment.

```
py.test --nbval-lax
```

The commands above will execute all the `.ipynb` files in the current folder. Alternatively, you can execute a specific notebook:

```
py.test --nbval my_notebook.ipynb
```

If the output lines are going to be sanitized, an extra flag, `--sanitize-with` together with the path to a confguration file with regex expressions, must be passed, i.e.

```
py.test --nbval my_notebook.ipynb --sanitize-with path/to/my_sanitize_file
```

where `my_sanitize_file` has the following structure.

```
[Section1]
regex: [a-z]*
replace: abcd

regex: [1-9]*
replace: 0000

[Section2]
regex: foo
```

```
replace: bar
```

The `regex` option contains the expression that is going to be matched in the outputs, and `replace` is the string that will replace the `regex` match. Currently, the section names do not have any meaning or influence in the testing system, it will take all the sections and replace the corresponding options.

Help

The `py.test` system help can be obtained with `py.test -h`, which will show all the flags that can be passed to the command, such as the verbose `-v` option. The IPython notebook plugin can be found under the `general` section.

Acknowledgements

This plugin was inspired by Andrea Zonca's py.test plugin for collecting unit tests in the IPython notebooks (https://github.com/zonca/pytest-ipynb).

The original prototype was based on the template in https://gist.github.com/timo/2621679 and the code of a testing system for notebooks https://gist.github.com/minrk/2620735 which we integrated and mixed with the `py.test` system.

We acknowledge financial support from

- OpenDreamKit Horizon 2020 European Research Infrastructures project (#676541), http://opendreamkit.org
- EPSRC's Centre for Doctoral Training in Next Generation Computational Modelling, http://ngcm.soton.ac.uk (#EP/L015382/1) and EPSRC's Doctoral Training Centre in Complex System Simulation ((EP/G03690X/1),
- The Gordon and Betty Moore Foundation through Grant GBMF #4856,by the Alfred P. Sloan Foundation and by the Helmsley Trust.

Authors

2014 - 2017 David Cortes-Ortuno, Oliver Laslett, T. Kluyver, Vidar Fauske, Maximilian Albert, MinRK, Ondrej Hovorka, Hans Fangohr

APPENDIX B. NBVAL'S DOCUMENTATION

documentation

February 27, 2017

1 IPython Notebook Validation for py.test - Documentation

One of the powerful uses of the IPython notebook is for documentation purposes, here we use a notebook to demonstrate the behaviour and usage of the IPython Notebook Validation plugin for py.test. The IPython notebook format `.ipynb` stores outputs as well as inputs. Validating the notebook means to rerun the notebook and make sure that it is generating the same output as has been stored.

Therefore, the **user MUST make the following the distinction**:

1. Running a notebook manually will likely change the output stored in the associated .ipynb file. These outputs will be used as references for the tests (i.e. the outputs from the last time you ran the notebook)
2. Validating with py.test plugin - these tests run your notebook code seperately without storing the information, the outputs generated will be compared against those in the .ipynb file

The purpose of the testing module is to ensure that the notebook is behaving as expected and that changes to underlying source code, haven't affected the results of an IPython notebook. For example, for documentation purposes - such as this.

1.0.1 Command line usage

The py.test program doesn't usually collect notebooks for testing; by passing the `--nbval` flag at the command line, the IPython Notebook Validation plugin will collect and test notebook cells, comparing their outputs with those saved in the file.

```
$ py.test --nbval my_notebook.ipynb
```

There is also an option `--nbval-lax`, which collects notebooks and runs them, failing if there is an error. This mode does not check the output of cells unless they are marked with a special `#NBVAL_CHECK_OUTPUT` comment.

```
$ py.test --nbval-lax my_notebook.ipynb
```

1.0.2 REGEX Output sanitizing

Since all output is captured by the IPython notebook, some pesky messages and prompts (with time-stamped messages, for example) may fail tests always, which might be expected. The plugin allows the user to specify a sanitizing file at the command prompt using the following flag:

```
$ py.test --nbval my_notebook.ipynb --sanitize-with my_sanitize_file
```

This sanitize file contains a number of REGEX replacements. It is recommended, when removing output for the tests, that you replace the removed output with some sort of marker, this helps with debugging. The following file is written to the folder of this notebook and can be used to santize its outputs:

```
In [1]: %%writefile doc_sanitize.cfg
        [regex1]
        regex: \d{1,2}/\d{1,2}/\d{2,4}
        replace: DATE-STAMP

        [regex2]
        regex: \d{2}:\d{2}:\d{2}
        replace: TIME-STAMP

Writing doc_sanitize.cfg
```

The first replacement finds dates in the given format replaces them with the label 'DATE-STAMP', likewise for strings that look like time. These will prevent the tests from failing due to time differences.

1.0.3 Validate this notebook

You can validate this notebook yourself, as shown below; the outputs that you see here are stored in the ipynb file. If your system produces different outputs, the testing process will fail. Just use the following commands:

```
$ cd /path/to/this/notebook
$ py.test --nbval documentation.ipynb --sanitize-with doc_sanitize.cfg
```

1.0.4 Examples of plugin behaviour

The following examples demonstrate how the plugin behaves during testing. Test this notebook yourself to see the validation in action!

These two imports produce no output as standard, if any **warnings** are printed out the cell will fail. Under normal operating conditions they will pass.

```
In [2]: import numpy as np
        import time
```

If python doesn't consistently print 7, then something has gone terribly wrong. **Deterministic cells** are expected to pass everytime

```
In [3]: print(5+2)

7
```

Random outputs will always fail.

```
In [4]: print([np.random.rand() for i in range(4)])
        print([np.random.rand() for i in range(4)])
```

```
[0.36133679016382714, 0.5043774697891126, 0.23281910875007927, 0.2713065513128683]
[0.5512421277985322, 0.02592706358897756, 0.05036036771084684, 0.7515926759190724]
```

Inconsistent number of lines of output will cause an error to be thrown.

```
In [5]: for i in range(np.random.randint(1, 8)):
            print(1)
```

```
1
1
1
```

Because the **time and date** will change with each run, we would expect this cell to fail everytime. Using the sanitize file `doc_sanitize.cfg` (created above) you can clean up these outputs.

```
In [6]: print('The time is: ' + time.strftime('%H:%M:%S'))
        print("Today's date is: " + time.strftime('%d/%m/%y'))
```

```
The time is: 15:28:30
Today's date is: 21/12/16
```

1.0.5 Avoid output comparison for specific cells

In case we want to avoid the testing process in specific input cells, we can write the comment ** #NBVAL_IGNORE_OUTPUT ** at the beginning of the them:

```
In [7]: # NBVAL_IGNORE_OUTPUT
        print('This is not going to be tested')
        print(np.random.randint(1, 20000))
```

```
This is not going to be tested
12544
```

There's also a counterpart, to ensure the output is tested even when using `--nbval-lax`:

```
In [8]: # NBVAL_CHECK_OUTPUT
        print("This will be tested")
        print(6 * 7)
```

```
This will be tested
42
```

1.0.6 Skipping specific cells

If, for some reason, a cell should not be executed during testing, the comment **# NBVAL_SKIP** can be used:

```
# NBVAL_SKIP
print("Entering infinite loop...")
while True:
    pass
```

1.0.7 Checking exceptions

Sometimes, we might want to allow a notebook cell to raise an exception, and check that the traceback is as we expect. By annotating the cell with the comment ** # NBVAL_RAISES_EXCEPTION ** you can indicate that the cell is expected to raise an exception. The full traceback is not compared, but rather just that the raised exception is the same as the stored exception.

```
In [3]: # NBVAL_RAISES_EXCEPTION
        print("This exception will be tested")
        raise RuntimeError("Foo")

This exception will be tested


        ---------------------------------------------------------------------------

        RuntimeError                              Traceback (most recent call last)

        <ipython-input-1-b97c0d501d6a> in <module>()
          1 print("This exception will be tested")
    ----> 2 raise RuntimeError("Foo")


        RuntimeError: Foo
```

This composes with the per-cell checking comments, so if you would like to avoid exceptions creating a test failure, but do not want to check the traceback, use `# NBVAL_IGNORE_OUTPUT`

```
In [3]: # NBVAL_RAISES_EXCEPTION
        print("If the raised exception doesn't match the stored exception, we get a
        raise SyntaxError("Foo")

If the raised exception doesn't match the stored exception, we get a failure


        ---------------------------------------------------------------------------
```

```
        RuntimeError                              Traceback (most recent call last)

        <ipython-input-3-32dcc1c70a4e> in <module>()
          1 # NBVAL_RAISES_EXCEPTION
          2 print("If the raised exception doesn't match the stored exception, we g
    ----> 3 raise RuntimeError("Foo")


        RuntimeError: Foo
```

```
In [2]: # NBVAL_IGNORE_OUTPUT
        # NBVAL_RAISES_EXCEPTION
        print("This exception will not be checked, but will not cause a failure.")
        raise RuntimeError("Bar")
```

```
This exception will not be checked, but will not cause a failure.
```

```
        ---------------------------------------------------------------------------

        RuntimeError                              Traceback (most recent call last)

        <ipython-input-2-bbee3f9e7de1> in <module>()
          2 # NBVAL_RAISES_EXCEPTION
          3 print("This exception will not be checked, but will not cause a failure
    ----> 4 raise RuntimeError("Bar")


        RuntimeError: Bar
```

1.0.8 Using tags instead of comments

If you do not want to put nbval comment annotations in your notebook, or your source language is not compatible with such annotations, you can use cell tags instead. Cell tags are strings that are added to the cell metadata under the label "tags", and can be added and remove using the "Tags" toolbar from Notebook version 5. The tags that Nbval recognizes are the same as the comment names, except lowercase, and with dashes ('-') instead of underscores ('_').

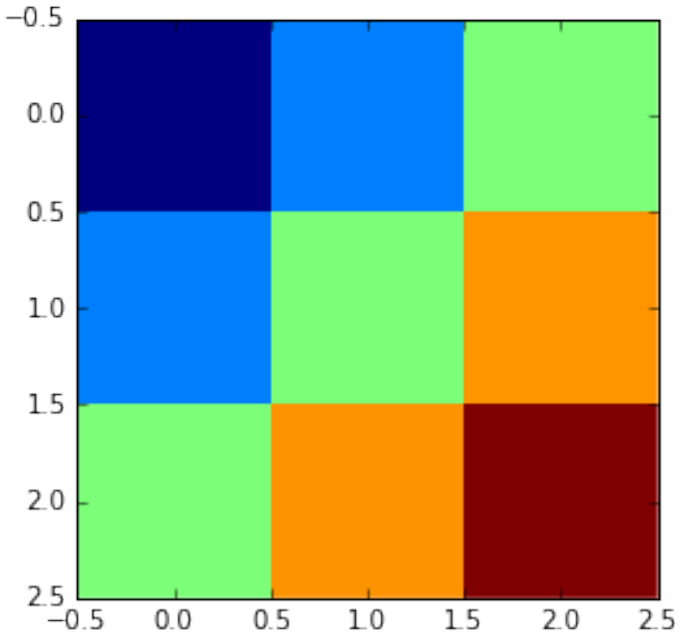
1.0.9 Figures

```
In [9]: import matplotlib.pyplot as plt
        %matplotlib inline
```

Currently, only the matplotlib text output of the Figure is compared, but it is possible to modify the plugin to allow comparison of the image whole string.

```
In [10]: plt.imshow(np.array([[i + j for i in range(3)]
                               for j in range(3)]),
                    interpolation='None'
                    )
```

```
Out[10]: <matplotlib.image.AxesImage at 0x7f2cb3374198>
```




```
In [ ]:
```

Disclaimer: this report, together with its annexes and the reports for the earlier deliverables, is self contained for auditing and reviewing purposes. Hyperlinks to external resources are meant as a convenience for casual readers wishing to follow our progress; such links have been checked for correctness at the time of submission of the deliverable, but there is no guarantee implied that they will remain valid.