nbchkr

Release 0.0.7

Jan 06, 2021

Contents:

A lightweight solution to mark/grade/check notebook assignments.

This documentation is in four parts:

- A tutorial: a step by step walk through of writing an assignment, using `nbchkr` to create a version to be released and checking submissions.
- A how to guide: a collection of short and to the point instructions for carrying out specific tasks.
- A series of explanations: some background information on `nbchkr`.
- A reference section: includes bibliography, a changelog and viewable source code.

CHAPTER 1

Tutorial

This tutorial will take you three the main steps of using `nbchkr`:

- Write an assignment with solutions and checks.
- Create a release notebook with the solutions and checks removed.
- For a collection of submissions: check the work and create individual feedback.

1.1 Installing `nbchkr`

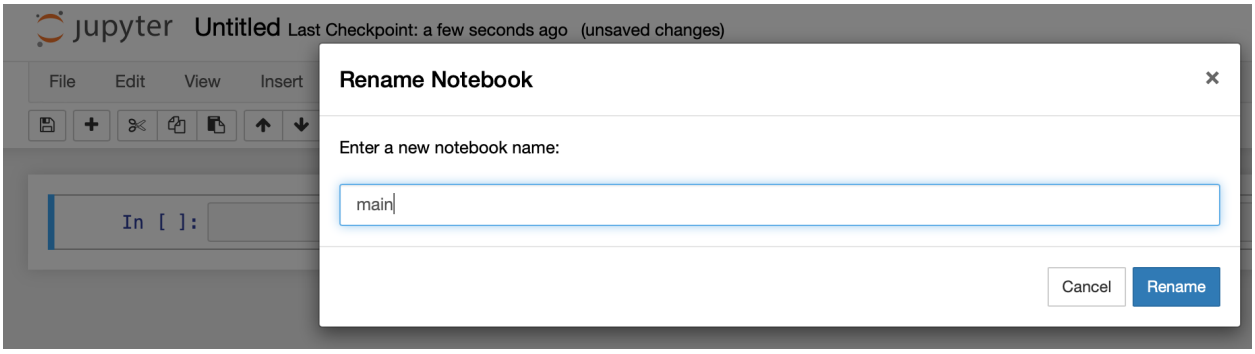
To install the latest release of `nbchkr`, at a command line interface run the following command:

```
$ python -m pip install nbchkr
```

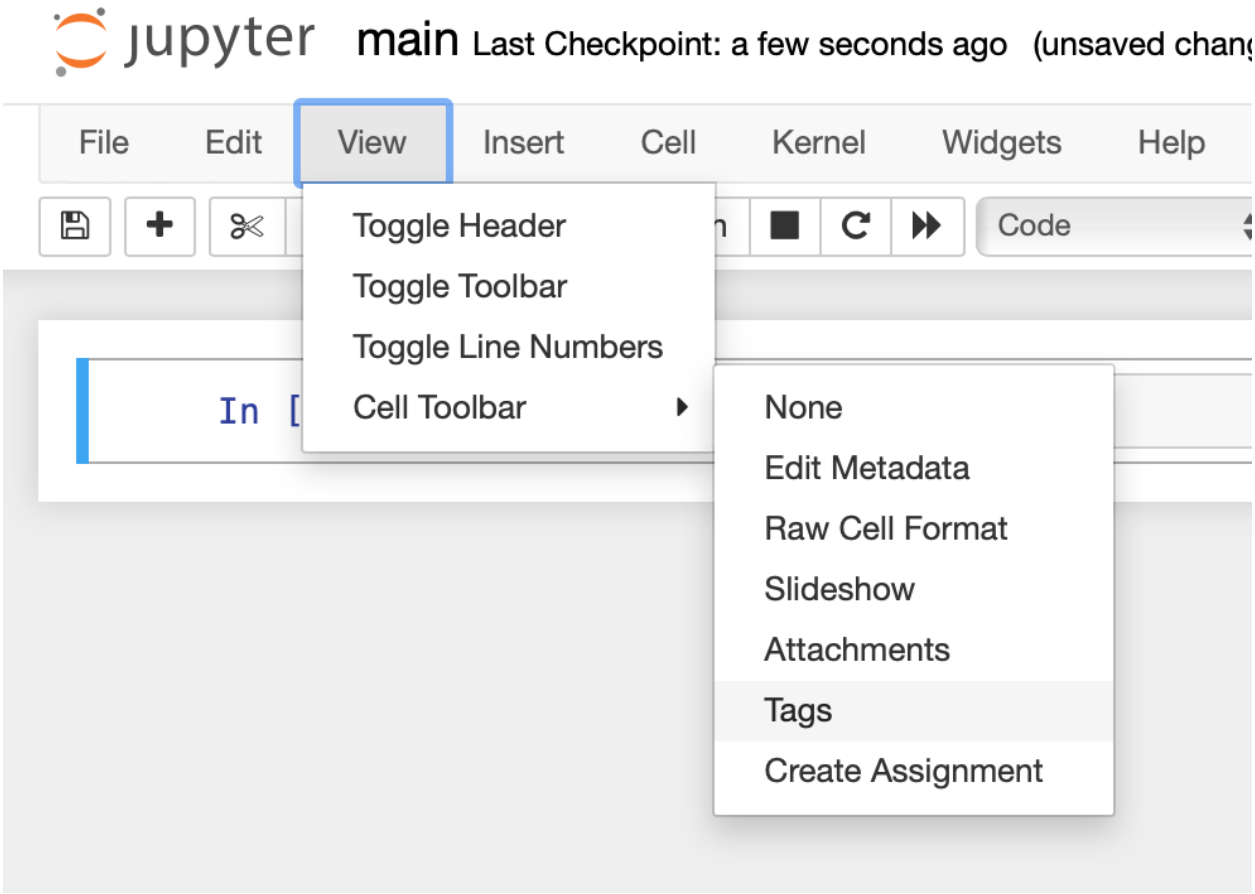
1.2 Writing an assignment

1.2.1 Initial setup

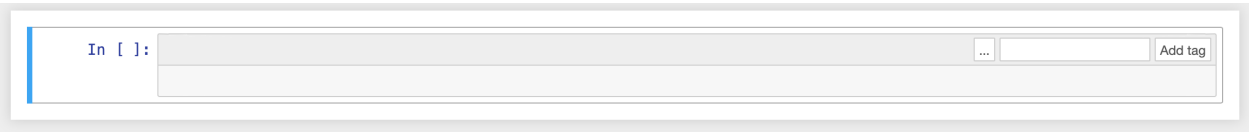
Open a Jupyter notebook, we will choose to name it `main.ipynb` (but the name is not important).



On the Jupyter toolbar, click on `View` and then `Cell Toolbar` and then `Tags`.



This should make the native tag menu available to you on every cell in your Jupyter notebook.



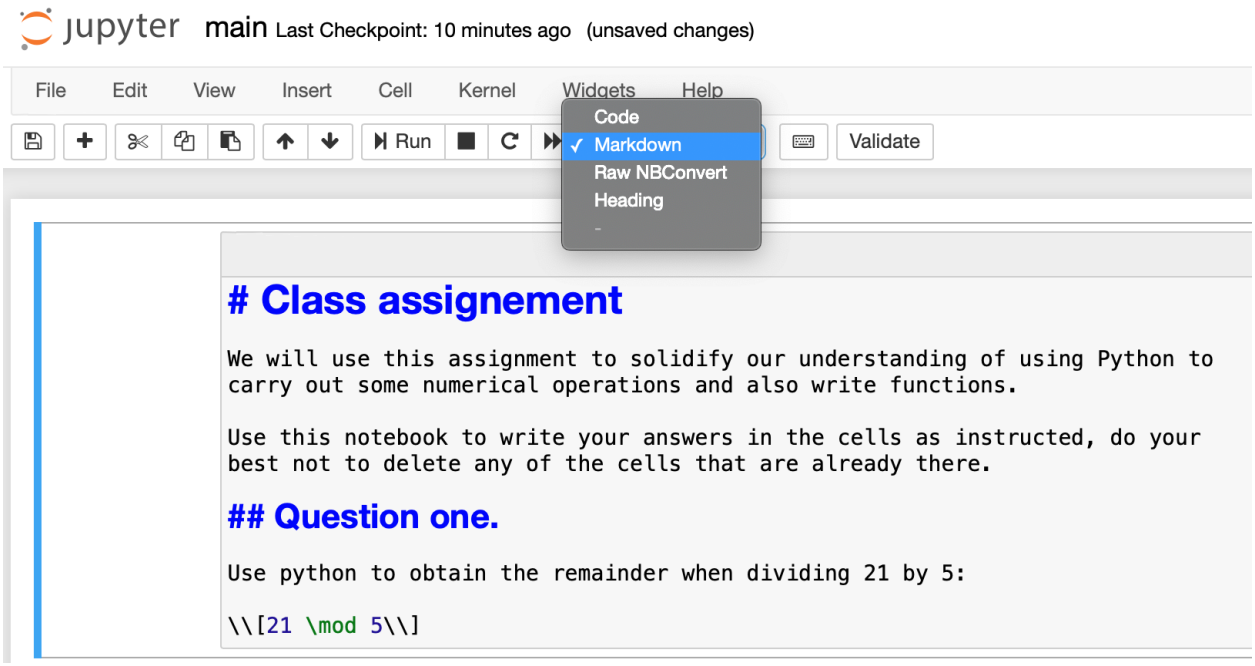
We can now start writing our assignment.

1.2.2 Writing text for a question

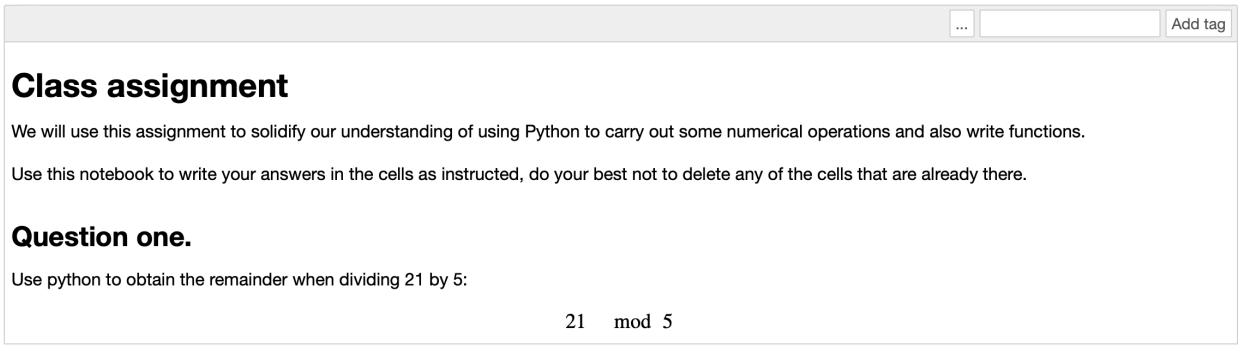
Let us write a markdown cell with some instructions and a question that we want to ask our students:

```
# Class assignment

We will use this assignment to solidify our understanding of using Python to
carry out some numerical operations and also write functions.

Use this notebook to write your answers in the cells as instructed, do your
best not to delete any of the cells that are already there.

## Question one.

Use python to obtain the remainder when dividing 21 by 5.

\\[21 \mod 5\\]
```

Be sure to indicate that that cell is a `markdown` cell and not the usual `code` cell.



Once you run that cell it should like like the following:



1.2.3 Writing the answer to a question

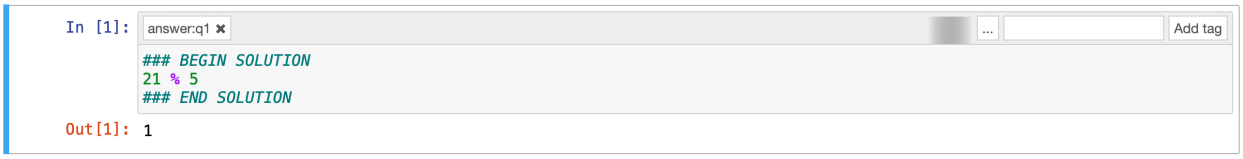
In the next cell we will write down the expected answer but also include a delimiters for what should not be shown to students:

```
### BEGIN SOLUTION
21 % 5
### END SOLUTION
```

We can run that cell if we want to keep an eye on the answer.

An important step at this stage is to let `nbchkr` know that this is an answer cell, we do this by adding `answer:q1` to tags.

Everything should now look like the following:



1.2.4 Writing checks for the answer

We will now write a check for the answer, that `nbchkr` uses to be able to give feedback to a student. We do this using python `assert` statements:

```
q1_answer = _
feedback_text = "Your opteration did not return an integer which is expected"
assert type(q1_answer) is int
```

We will also add a tag: `score:1` to this cell.

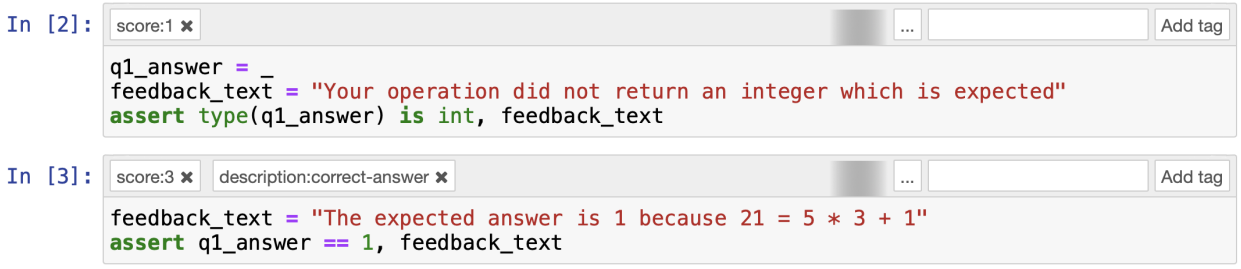
As well as checking that the answer is an integer let us check the actual answer by creating a new cell and writing:

```
feedback_text = "The expected answer is 1 because 21 = 5 * 3 + 1"
assert q1_answer == 1, feedback_text
```

This will be worth 3 points so let us add the tag: `score:3`.

We can choose to add a description to our check which will then appear in the feedback. We do this by adding the tag: `description:correct-answer`.

Everything should now look like the following:



1.2.5 Writing another question

Let us write a second question that asks students to write a function:

```
## Question two.

Write a python function `get_remainder(m, n)` that returns the remainder
the remainder when dividing \\(m\\) by \\(n\\).

\\[m \mod n\\]
```

1.2.6 Writing the answer

As before we write an answer in a cell below:

```
def get_remainder(m, n):
    ### BEGIN SOLUTION
    """
    This function returns the remainder of m when dividing by n
    """
    return m % n
    ### END SOLUTION
```

1.2.7 Including checks

We will now add some cells to check the answer.

First let us make sure there is a docstring:

```
feedback_text = """You did not include a docstring. This is important to help␣
↪document your code.


It is done  using triple quotation marks. For example:

def get_remainder(m, n):
    \"\"\"
    This function returns the remainder of m when dividing by n
    \"\"\"
    ...

Using that it's possible to access the docstring,
one way to do this is to type: `get_remainder?`
(which only works in Jupyter) or help(get_remainder).

We can also comment code using `#` but this is completely
ignored by Python so cannot be accessed in the same way.

"""
assert  get_remainder.__doc__ is not None, feedback_text
```

Whilst we've decided to write quite a lot of feedback with details about writing docstrings we are only going to score this part of the answer 1 point so we use the tag: *score:1*.

We will add the description tag: `description:presence-of-docstring`.

We will also include specific checks for the actual answer:

```
assert get_remainder(5, 3) == 2, "Incorrect answer for m=5, n=3: 5 mod 2 = 1 because␣
↪5 = 3 * 1 + 2"
assert get_remainder(34, 21) == 13, "Incorrect answer for m=34, n=21: 34 mod 21 = 13␣
↪because 34 = 21 * 1 + 13"
assert get_remainder(1000, 10) == 0, "Incorrect answer for m=1000, n=10: 1000 mod 10␣
↪= 0 because 1000 = 10 * 100 + 0"
```

For this we will use the description tag: `description:correct-answer`.

IF you would like to see a final version of this notebook you can find it `here`.

1.3 Releasing an assignment

Now we can take that source notebook and create an assignment that can be given to students. To do this, we use the command line tool that comes with `nbchkr`:

```
$ nbchkr release --source main.ipynb --output assignment.ipynb
```

This creates `assignment.ipynb` with the answers and checks removed.

1.4 Releasing solutions

If we want to create a model solution we can. To do this, we use the command line tool that comes with `nbchkr`:

```
$ nbchkr solve --source main.ipynb --output solution.ipynb
```

This creates `solution.ipynb` with the checks removed.

1.5 Checking student assignments and generating feedback

Assuming we have a class of 3 students who each submitted a notebook with the following naming convention:

```
assignment_<student_number>.ipynb
```

These notebooks are all put in a `submissions/` directory:

- `assignment_01.ipynb`
- `assignment_02.ipynb`
- `assignment_03.ipynb`

To check them and generate the feedback we again use the `nbchkr` command line tool:

```
$ nbchkr check --source main.ipynb --submitted "submissions/*.ipynb" --feedback-
↪suffix -feedback.md --output data.csv
```

This has gone through and checked each notebook, you can see the output here:

Table 1: The summary results

Submission filepath	Score	Maximum score	Tags match
submissions/assignment_01.ipynb	2	11	True
submissions/assignment_02.ipynb	10	11	True
submissions/assignment_03.ipynb	4	11	False

We see that *assignment_03.ipynb* has a `False` flag under the `Tags Match` heading: this is because the student must have deleted one of the cells with a required tag. `nbchkr` does its best to check them anyway but this is a notebook that we should check manually.

In the submissions directory, 3 markdown files have been written with feedback to the students:

`assignment_01.ipynb-feedback.md`:

```
---

## answer:q1

1 / 1

### Correct answer

The expected answer is 1 because 21 = 5 * 3 + 1

0 / 3

---

## answer:q2

### Presence of docstring

1 / 1

### Correct answer

Incorrect answer for m=5, n=3: 5 mod 2 = 1 because 5 = 3 * 1 + 2

0 / 6
```

`assignment_02.ipynb-feedback.md`:

```
---

## answer:q1

1 / 1

### Correct answer

3 / 3

---

## answer:q2
```

(continues on next page)

(continued from previous page)

```
### Presence of docstring

You did not include a docstring. This is important to help document your code.


It is done  using triple quotation marks. For example:

def get_remainder(m, n):
    """
    This function returns the remainder of m when dividing by n
    """
    ...

Using that it's possible to access the docstring,
one way to do this is to type: `get_remainder?`
(which only works in Jupyter) or help(get_remainder).

We can also comment code using `#` but this is completely
ignored by Python so cannot be accessed in the same way.



0 / 1

### Correct answer

6 / 6
```

assignment_03.ipynb-feedback.md:

```
---

## answer:q1

1 / 1

### Correct answer

3 / 3

---

## answer:q2

### Presence of docstring

name 'get_remainder' is not defined

0 / 1

### Correct answer

name 'get_remainder' is not defined

0 / 6
```

CHAPTER 2

How to:

This section of the documentation is aimed at those who want to know how to carry out a specific task with `nbchkr`.

How to:

2.1 install

To install the latest release of `nbchkr`:

```
$ python -m pip install nbchkr
```

2.2 write a source assignment

Writing an assignment is done by writing a Jupyter notebook and using tags:

2.2.1 Write a question

Use `markdown` cells in Jupyter to write your question.

2.2.2 Write an answer

In a `code` cell write the code snippet that is the answer to the question:

```
### BEGIN SOLUTION
<code>
### END SOLUTION
```

The `### BEGIN SOLUTION` and `### END SOLUTION` delimiters are necessary. It is possible to pass your own set of delimiters to `nbchkr` (see further documentation for that).

Add the `answer:<uique_label>` tag to the cell.

2.2.3 Write a check

In a `code` cell write `assert` statements to check specific elements of the answer:

```
assert <condition>, <error message>
```

If the `<condition>` is not meet the `<error message>` will be written to the feedback on a submission.

Note that it is possible to refer to the output of a previous cell using _.

Add the `score:<integer>` tag to the cell. The `<integer>` is the value associated with this specific check. If the `<condition>` is met then the `<integer>` value will be added to the total score of a student.

Optionally, you can also add the `description:<string>` tag to the cell. This will add the `<string>` to the feedback for that specific check. Note that spaces should be replaced with - which will automatically be replaced in the feedback. For example: `description:correct-answer` will appear as `### Correct answer` in the feedback.

Note that it is possible to write multiple checks for a given answer. This can be done so as to programmatically offer varying levels of feedback for specific parts of the task.

2.3 release an assignment

You can release an assignment in 1 or 2 ways:

1. Using the command line tool.
2. Using `nbchkr` as a library.

2.3.1 Using the command line tool

Given a source assignment `main.ipynb`:

```
$ nbchkr release --source main.ipynb --output assignment.ipynb
```

This creates `assignment.ipynb` with relevant cells removed which can then be distributed to students.

2.3.2 Using `nbchkr` as a library

All of `nbchkr`'s functionality is exposed to the user as a library.

Importing the relevant libraries:

```
>>> import pathlib
>>> import nbchkr.utils
```

Reading in the source notebook `main.ipynb` and removing relevant cells:

```
>>> nb_path = pathlib.Path("main.ipynb")
>>> nb_node = nbchkr.utils.read(nb_path=nb_path)
>>> student_nb = nbchkr.utils.remove_cells(nb_node=nb_node)
```

Writing the assignment notebooks `assignment.ipynb`:

```
>>> output_path = pathlib.Path("assignment.ipynb")
>>> nbchkr.utils.write(output_path=output_path, nb_node=nb_node)
```

Note that the `nbchkr.utils.remove_cells` function can take as arguments different regex patterns and replacement strings which allows flexibility for how to write your notebooks.

Writing a slightly different regex for solution delimiters:

```
>>> import re
>>> solution_regex = re.compile(r"### SOLUTION START[\s\S](.*?)[\s\S]### SOLUTION END
↪", re.DOTALL)
```

Writing a different replacement text, this is what the student will see instead of the solution:

```
>>> solution_repl = "# Write your solution here"
```

Removing the cells:

```
>>> student_nb = nbchkr.utils.remove_cells(nb_node=nb_node, solution_regex=solution_
↪regex, solution_repl=solution_repl)
```

2.4 solve an assignment

You can solve an assignment in 1 or 2 ways:

1. Using the command line tool.
2. Using `nbchkr` as a library.

2.4.1 Using the command line tool

Given a source assignment `main.ipynb`:

```
$ nbchkr solve --source main.ipynb --output solution.ipynb
```

This creates `solution.ipynb` with relevant cells removed which can then be distributed to students.

2.4.2 Using `nbchkr` as a library

All of `nbchkr`'s functionality is exposed to the user as a library.

Importing the relevant libraries:

```
>>> import pathlib
>>> import re
>>> import nbchkr.utils
```

Reading in the source notebook `main.ipynb` and removing relevant cells. We here use a regex that matches nothing for the solutions (as we want them to stay in place):

```
>>> nb_path = pathlib.Path("main.ipynb")
>>> solution_regex = re.compile('$^')
>>> nb_node = nbchkr.utils.read(nb_path=nb_path)
>>> student_nb = nbchkr.utils.remove_cells(nb_node=nb_node, solution_regex=solution_
↪regex)
```

Writing the assignment notebooks `assignment.ipynb`:

```
>>> output_path = pathlib.Path("solution.ipynb")
>>> nbchkr.utils.write(output_path=output_path, nb_node=nb_node)
```

2.5 check a submission

You can check a submission in 2 ways:

1. Using the command line tool.
2. Using `nbchkr` as a library.

2.5.1 Using the command line tool

Given a source assignment `main.ipynb` and a submission `submitted.ipynb` you can check the submission using:

```
$ nbchkr check --source main.ipynb --submitted submitted.ipynb --feedback-suffix -
↪feedback.md --output data.csv
```

This creates `submitted.ipynb-feedback.md` with feedback and outputs summary scores to `data.csv`.

Note that given a pattern matching a number of notebooks, for example all notebooks in `submissions/` you can check them all at once using:

```
$ nbchkr check --source main.ipynb --submitted "submissions/*.ipynb" --feedback-
↪suffix -feedback.md --output data.csv
```

2.5.2 Using `nbchkr` as a library

All of `nbchkr`'s functionality is exposed to the user as a library.

Importing the relevant libraries:

```
>>> import pathlib
>>> import nbchkr.utils
```

Reading in the source notebook `main.ipynb` and removing relevant cells:

```
>>> source_nb_path = pathlib.Path("main.ipynb")
>>> source_nb_node = nbchkr.utils.read(nb_path=source_nb_path)
```

Reading in the submitted notebook `submitted.ipyb` and check that the tags match (if they do not match the checker will still work but the results should be confirmed manually):

```
>>> submitted_nb_path = pathlib.Path("submitted.ipynb")
>>> nb_node = nbchkr.utils.read(submitted_nb_path)
>>> tags_match = nbchkr.utils.check_tags_match(source_nb_node=source_nb_node, nb_
↪node=nb_node)
>>> tags_match
True
```

Now we will add the checks to the submission from `main.ipynb` and run them:

```
>>> nb_node = nbchkr.utils.add_checks(nb_node=nb_node, source_nb_node=source_nb_node)
>>> score, maximum_score, feedback_md = nbchkr.utils.check(nb_node=nb_node)
>>> score
10
>>> maximum_score
11
>>> feedback_md
'\n---\n\n## answer:q1\n\n1 / 1\n\n3 / 3\n\n---\n\n## answer:q2\n\nYou did not␣
↪include a docstring. This is important to help document your code. \n\n\nIt is done␣
↪ using triple quotation marks. For example:\n\ndef get_remainder(m, n):\n    """\n␣
↪  This function returns the remainder of m when dividing by n\n    """\n    ...\n␣
↪ \nUsing that it\'s possible to access the docstring, \none way to do this is to␣
↪type: `get_remainder?` \n(which only works in Jupyter) or help(get_remainder).
↪\n\nWe can also comment code using `#` but this is completely \nignored by Python␣
↪so cannot be accessed in the same way.\n\n\n\n0 / 1\n\n6 / 6\n'
```

Note that the `nbrchkr.utils.check_tags_match`, `nbchkr.utils.add_checks` and `nbchkr.utils.check` functions can take further arguments that allow for customisation of behaviour.

2.6 handle a submission in the wrong format

If a file is checked that is not an *ipynb* file then the checker will write the following to the feedback file:

```
"Your notebook file was not in the correct format and could not be read"
```

Note than when batch checking, this will not stop the checker from checking the other files.

2.7 contribute

2.7.1 Installing a development version

To install a development version of the library:

```
$ python setup.py develop
```

2.7.2 Run tests

To run the basic unit tests:

```
$ python -m pytest
```

To run the full set of tests with syntax highlighting, doctests and coverage:

```
$ python -m pytest -v --cov=nbchkr --cov-fail-under=100 --flake8 --doctest-glob='*.rst
```

To run static type checking:

```
$ python -m mypy src/
```

To run the doctest coverage checker:

```
$ python -m interrogate -e setup.py -e tets/ -M -i -v -f 100
```

2.7.3 Style formatting

To the automatic style formatter `black`:

```
$ python -m black .
```

To run the import sorting formatter `isort`:

```
$ python -m isort src/nbchkr/.
```

2.7.4 Build the documentation

To build the documentation:

```
$ cd docs
$ make html
```

2.7.5 Git branching

The most up to date branch that all new features should be branched from is `dev`.

New releases are tagged.

2.8 release

The release process:

1. Adjust the version number in `version.py`.
2. Tag a new release:

   ```
   git tag <release number>
   ```

3. Push the new tag to github:

   ```
   git push --tags
   ```

4. Create a new release on github.
5. Create a distribution:

```
python setup.py sdist bdist_wheel
```

6. Use `twine` to upload to pypi:

```
python -m twine upload dist/*
```

CHAPTER 3

Explanation

3.1 why use `nbchkr`?

The design principles of `nbchkr` are:

1. Lightweight with few dependencies;
2. Flat file outputs;
3. Customisable;
4. Well documented.

There are alternatives to `nbchkr`, the most mature of which is `nbgrader` https://nbgrader.readthedocs.io which is a full course management system. It includes the ability to send emails to students, distribute assignments and feedback. It is a full featured class management solution that I wholeheartedly recommend checking out in case it is the tool you need.

3.2 why use automated checking?

In the discussion of [Schinke2014] (a review of assessment) the following sentence stands out:

> "However, underlying the less encouraging news about grades are numerous opportunities for faculty members to make assessment and evaluation more productive, better aligned with student learning, and less burdensome for faculty and students."

It is with idea in mind that automated checking of assessment should be implemented.

Indeed in [Wilcox2015] (a reflection on an implementation of automated checking) there are two initial questions:

1. Does the automated check detract from student learning?
2. Do the benefits of implementing automated checking outway the cost?

It is hoped that software similar to and like `nbchkr` answer the second question.

The first question however is positively answered in a number of pieces of work such as [Wilcox2015] itself where students reported a positive experience of using automated checking but also benefited academically which indicates a better overall learning process. This student satisfaction with the process is also reported in [Saikkonen2001]

Some of the downsides of human checking are listen in [Cheang2003]:

1. Difficulty of judging efficiency and correctness;
2. The fact that there can be multiple approaches to a problem that would be missed by a human checker.
3. Emphasis on aesthetics. Note that given the modern emphasis on the importance of code readability I am not convinced by this particular downside.
4. Inconsistency of human checkers
5. Time: the workload of checking works is huge.

This last point is often mentioned in the literature and specifically [Schinke2014] highlight the importance of creating time and space for meaningful feedback through self and peer evaluation.

Self evaluation as a general pedagogic strategy relates well to automated checking as described by [Losada2010] where they prescribe giving a number of tests to students as part of the assessment. As part of the [Losada2010] a discussion as part of Bloom's Taxonomy is given however this will not be discussed here given numerous downsides to the taxonomy (see for example: [Case2013]). [Losada2010] lists numerous advantages to automated checking:

1. Fast feedback: in their case and similar to [Cheang2003], [Saikkonen2001] and others the particular framework being described was an "online" one that students could use to gain immediate feedback.
2. Fairer grading;
3. Permanent access;
4. Efficiency;
5. Fostering a positive attitude towards test driven development (TDD).

This last point relates to the testing practice in software engineering of writing a test before writing the software.

Specific strategies for writing checks are described in [Saikkonen2001] and [Wilcox2016] which give insight and guidance on writing using tests that allow for feedback that helps students identify errors. Contrary to [Cheang2003]'s suggestion that aesthetics having a major role in human checking being a negative, [Wilcox2016] points out that static tools can be used to check the code quality itself (all within the testing framework). One such example of this is in the *Tutorial* where a test is included to make sure that the code written is documented.

There are some negative aspects to automated testing another good quote from [Schinke2014] is:

> "In fact, we have presented evidence that accuracy-based grading may, in fact demotivate students and impede learning."

It was noted also in [Wilcox2015] that some students do feel that it the automated checks "unnecessarily strict". Finally, [Wilcox2016] discusses some aspects of security and that automated testing can be done inside of a virtual machine to avoid running of malicious code.

These are all aspects to be considered when writing the specific checks for the assignments and not losing sight of the end goal which is to create a positive environment for student learning. Automated checking should not be thought of as a solution to a problem of assessment but hopefully a tool that enables better learning through:

1. Timely and actionable feedback;
2. The creation of space for productive learning activities.

CHAPTER 4

Reference

4.1 Bibliography

This is a collection of various bibliographic items referenced in the documentation.

4.2 Changelog

4.2.1 v0.0.7 - Add ability to handle non notebook submissions

2021-01-04

4.2.2 v0.0.6 - Improve handling of blank space

2020-12-16

4.2.3 v0.0.5 - Update changelog

2020-12-11

4.2.4 v0.0.4 - Add ability to have descriptions in feedback and create model solutions

2020-12-11

4.2.5 v0.0.3 - Minor internal changes and improvement to docs

Shifted to using typer for the cli.

2020-09-16

4.2.6 v0.0.2 - Minor change to contribution docs

2020-08-25

4.2.7 v0.0.1 - First release

2020-08-25

4.3 Source code

nbchkr.utils.**add_checks**(*nb_node: dict*, *source_nb_node: dict*, *answer_tag_regex=None*) → dict
Given a *nb_node* and a source *source_nb_node*, add the cells in *nb_node* with tags matching *answer_tag_regex* to *source_nb_node*

This is used to add a student's answers to the source notebook.

nbchkr.utils.**check**(*nb_node: dict*, *timeout: int = 600*, *score_regex_pattern=None*, *answer_tag_pattern=None*) → Tuple[Optional[int], Optional[int], str]
Given a *nb_node*, it executes the notebook and keep track of the score.

This returns 3 things:

- The student score
- The total score obtainable
- Some feedback in markdown format

nbchkr.utils.**check_tags_match**(*source_nb_node: dict*, *nb_node: dict*, *tag_seperator: str = '|'*, *tag_regex=None*) → bool
This checks if the count of tags that match *tag_regex* on each cell matches. Note that it does not necessarily guarantee that the tags are on the same cells.

nbchkr.utils.**get_description**(*cell: dict*, *description_regex_pattern=None*, *tag_seperator: str = '|'*) → str
Given a *cell* of a notebook, return the description as defined by the *description_regex_pattern*.

nbchkr.utils.**get_score**(*cell: dict*, *score_regex_pattern=None*) → int
Given a *cell* of a notebook, return the score as defined by the *score_regex_pattern*.

nbchkr.utils.**get_tags**(*cell: dict*, *tag_seperator: str = '|'*, *tag_regex=None*) → str
Given a *cell* of a notebook, return a string with all tags that match *tag_regex* separated by |.

nbchkr.utils.**read**(*nb_path: Union[pathlib.Path, str]*, *as_version: int = 4*) → dict
Read a jupyter notebook file at *nb_path*.

Returns the python *dict* representation.

nbchkr.utils.**remove_cells**(*nb_node*, *tags_regex_patterns_to_ignore=None*, *solution_regex=None*, *solution_repl=None*)
Given a dictionary representation of a notebook, removes:

- Cells with tags matching patterns in *tags_regex_patterns_to_ignore*

- Text in cells matching the *solution_regex* pattern.

Returns the python *dict* representation.

nbchkr.utils.**write**(*output_path: pathlib.Path*, *nb_node: dict*)
Write the python dict representation of a notebook to *output_path*.

Bibliography

[Case2013] Case, Roland. “The Unfortuate Consequences of Bloom’s Taxonomy.” Social Education 77.4 (2013): 196-200.

[Cheang2003] Cheang, Brenda, et al. “Automated grading of programming assignments.” Proceedings of the 11th International Conference on Computers in Education (ICCE 2003). 2003.

[Losada2010] Losada, Isidoro Hernin, Cristóbal Pareja Flores, and J. Éngel Velizquez Iturbide. “Pedagogical use of automatic graders.” Advances in Learning Processes, InTech (2010).

[Saikkonen2001] Saikkonen, Riku, Lauri Malmi, and Ari Korhonen. “Fully automatic assessment of programming exercises.” Proceedings of the 6th annual conference on Innovation and technology in computer science education. 2001.

[Schinke2014] Schinske, Jeffrey, and Kimberly Tanner. “Teaching more by grading less (or differently).” CBE—Life Sciences Education 13.2 (2014): 159-166.

[Wilcox2015] Wilcox, Chris. “The role of automation in undergraduate computer science education.” Proceedings of the 46th ACM Technical Symposium on Computer Science Education. 2015.

[Wilcox2016] Wilcox, Chris. “Testing strategies for the automated grading of student programs.” Proceedings of the 47th ACM Technical Symposium on Computing Science Education. 2016.

Python Module Index

n

Index