Jupyter Sphinx

Release 0.4.0

Jupyter Development Team

Dec 18, 2022

CONTENTS:

Jupyter-sphinx is a Sphinx extension that executes embedded code in a Jupyter kernel, and embeds outputs of that code in the document. It has support for rich output such as images, Latex math and even javascript widgets, and it allows to enable thebelab for live code execution with minimal effort.

CHAPTER

ONE

INSTALLATION

Get jupyter-sphinx from pip:

```
pip install jupyter-sphinx
```

or conda:

```
conda install jupyter_sphinx -c conda-forge
```

CHAPTER

TWO

ENABLING THE EXTENSION

To enable the extension, add `jupyter_sphinx` to your enabled extensions in `conf.py`:

```
extensions = [
    'jupyter_sphinx',
]
```

CHAPTER

THREE

BASIC USAGE

You can use the `jupyter-execute` directive to embed code into the document:

```
.. jupyter-execute::

  name = 'world'
  print('hello ' + name + '!')
```

The above is rendered as follows:

```
name = 'world'
print('hello ' + name + '!')
```

```
hello world!
```

Note that the code produces *output* (printing the string 'hello world!'), and the output is rendered directly after the code snippet.

Because all code cells in a document are run in the same kernel, cells later in the document can use variables and functions defined in cells earlier in the document:

```
a = 1
print('first cell: a = {}'.format(a))
```

```
first cell: a = 1
```

```
a += 1
print('second cell: a = {}'.format(a))
```

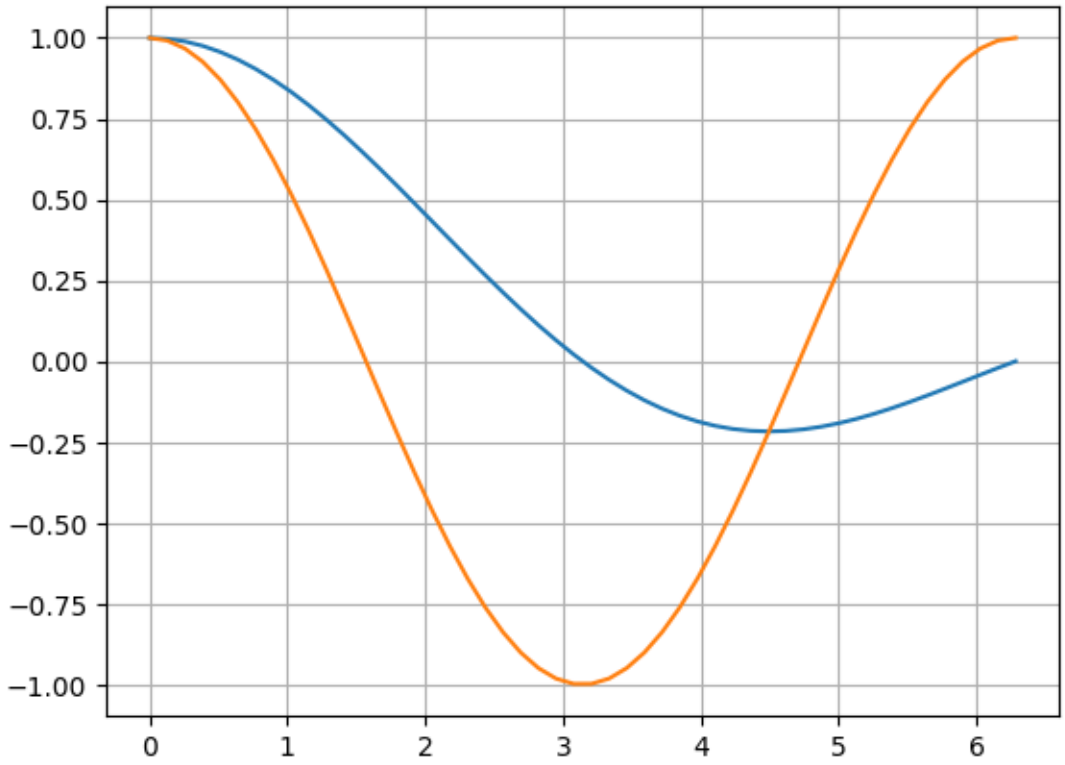
```
second cell: a = 2
```

Because jupyter-sphinx uses the machinery of `nbconvert`, it is capable of rendering any rich output, for example plots:

```
import numpy as np
from matplotlib import pyplot
%matplotlib inline

x = np.linspace(1E-3, 2 * np.pi)

pyplot.plot(x, np.sin(x) / x)
pyplot.plot(x, np.cos(x))
pyplot.grid()
```



LaTeX output:

```
from IPython.display import Latex
Latex('\int_{-\infty}^\infty e^{-x²}dx = \sqrt{\pi}')
```

$$\int_{-\infty}^{\infty} e^{-x^2} dx = \sqrt{\pi}$$

or even full-blown javascript widgets:

```
import ipywidgets as w
from IPython.display import display

a = w.IntSlider()
b = w.IntText()
w.jslink((a, 'value'), (b, 'value'))
display(a, b)
```

```
IntSlider(value=0)
```

```
IntText(value=0)
```

It is also possible to include code from a regular file by passing the filename as argument to `jupyter-execute`:

```
.. jupyter-execute:: some_code.py
```

`jupyter-execute` may also be used in docstrings within your Python code, and will be executed when they are included with Sphinx autodoc.

CHAPTER

FOUR

THEBELAB SUPPORT

To turn on thebelab, specify its configuration directly in `conf.py`:

```
jupyter_sphinx_thebelab_config = {
    'requestKernel': True,
    'binderOptions': {
        'repo': "binder-examples/requirements",
    },
}
```

With this configuration, thebelab is activated with a button click:

By default the button is added at the end of the document, but it may also be inserted anywhere using

```
.. thebe-button:: Optional title
```

CHAPTER

FIVE

DIRECTIVE OPTIONS

You may choose to hide the code of a cell, but keep its output visible using `:hide-code:`:

```
.. jupyter-execute::
    :hide-code:

    print('this code is invisible')
```

produces:

```
this code is invisible
```

this option is particularly useful if you want to embed correctness checks in building your documentation:

```
.. jupyter-execute::
    :hide-code:

    assert everything_works, "There's a bug somewhere"
```

This way even though the code won't make it into the documentation, the build will fail if running the code fails.

Similarly, outputs are hidden with `:hide-output:`:

```
.. jupyter-execute::
    :hide-output:

    print('this output is invisible')
```

produces:

```
print('this output is invisible')
```

You may also display the code *below* the output with `:code-below:`:

```
.. jupyter-execute::
    :code-below:

    print('this code is below the output')
```

produces:

```
this code is below the output
```

```
print('this code is below the output')
```

You may also add *line numbers* to the source code with `:linenos:`:

```
.. jupyter-execute::
   :linenos:

   print('A')
   print('B')
   print('C')
```

produces:

```
1 print('A')
2 print('B')
3 print('C')
```

```
A
B
C
```

To add *line numbers from a specific line* to the source code, use the `lineno-start` directive:

```
.. jupyter-execute::
   :lineno-start: 7

   print('A')
   print('B')
   print('C')
```

produces:

```
7 print('A')
8 print('B')
9 print('C')
```

```
A
B
C
```

You may also emphasize particular lines in the source code with `:emphasize-lines:`:

```
.. jupyter-execute::
    :emphasize-lines: 2,5-6

    d = {
        'a': 1,
        'b': 2,
        'c': 3,
        'd': 4,
        'e': 5,
    }
```

produces:

```
2 d = {
3     'a': 1,
4     'b': 2,
5     'c': 3,
```

(continues on next page)

(continued from previous page)

```
    'd': 4,
    'e': 5,
}
```

CHAPTER

SIX

CONTROLLING EXCEPTIONS

The default behaviour when jupyter-sphinx encounters an error in the embedded code is just to stop execution of the document and display a stack trace. However, there are many cases where it may be illustrative for execution to continue and for a stack trace to be shown as *output of the cell*. This behaviour can be enabled by using the `raises` option:

```
.. jupyter-execute::
    :raises:

    1 / 0
```

produces:

```
1 / 0
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Cell In[13], line 1
----> 1 1 / 0

ZeroDivisionError: division by zero
```

Note that when given no arguments, `raises` will catch all errors. It is also possible to give `raises` a list of error types; if an error is raised that is not in the list then execution stops as usual:

```
.. jupyter-execute::
    :raises: KeyError, ValueError

    a = {'hello': 'world!'}
    a['jello']
```

produces:

```
a = {'hello': 'world!'}
a['jello']
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[14], line 2
      1 a = {'hello': 'world!'}
----> 2 a['jello']

KeyError: 'jello'
```

Additionally, any output sent to the `stderr` stream of a cell will result in jupyter-sphinx producing a warning. This behaviour can be suppressed (and the `stderr` stream printed as regular output) by providing the `stderr` option:

```
.. jupyter-execute::
    :stderr:

    import sys

    print("hello, world!", file=sys.stderr)
```

produces:

```
import sys

print("hello, world!", file=sys.stderr)
```

```
hello, world!
```

CHAPTER

SEVEN

MANUALLY FORMING JUPYTER CELLS

When showing code samples that are computationally expensive, access restricted resources, or have non-deterministic output, it can be preferable to not have them run every time you build. You can simply embed input code without executing it using the `jupyter-input` directive expected output with `jupyter-output`:

```
.. jupyter-input::
    :linenos:

    import time

    def slow_print(str):
        time.sleep(4000)    # Simulate an expensive process
        print(str)

    slow_print("hello, world!")

.. jupyter-output::

    hello, world!
```

produces:

```
import time

def slow_print(str):
    time.sleep(4000)    # Simulate an expensive process
    print(str)

slow_print("hello, world!")
```

```
hello, world!
```

CHAPTER

EIGHT

CONTROLLING THE EXECUTION ENVIRONMENT

The execution environment can be controlled by using the `jupyter-kernel` directive. This directive takes the name of the Jupyter kernel in which all future cells (until the next `jupyter-kernel` directive) should be run:

```
.. jupyter-kernel:: python3
    :id: a_unique_name
```

`jupyter-kernel` can also take a directive option `:id:` that names the Jupyter session; it is used in conjunction with the `jupyter-download` roles described in the next section.

Note that putting a `jupyter-kernel` directive starts a *new* kernel, so any variables and functions declared in cells *before* a `jupyter-kernel` directive will not be available in future cells.

Note that we are also not limited to working with Python: Jupyter Sphinx supports kernels for any programming language, and we even get proper syntax highlighting thanks to the power of Pygments.

CHAPTER

NINE

DOWNLOADING THE CODE AS A SCRIPT

Jupyter Sphinx includes 2 roles that can be used to download the code embedded in a document: `:jupyter-download-script:` (for a raw script file) and `:jupyter-download-notebook:` or `:jupyter-download-nb:` (for a Jupyter notebook).

These roles are equivalent to the standard sphinx download role, **except** the extension of the file should not be given. For example, to download all the code from this document as a script we would use:

```
:jupyter-download-script:`click to download <index>`
```

Which produces a link like this: `click to download`. The target that the role is applied to (`index` in this case) is the name of the document for which you wish to download the code. If a document contains `jupyter-kernel` directives with `:id:` specified, then the name provided to `:id:` can be used to get the code for the cells belonging to the that Jupyter session.

CHAPTER

TEN

STYLING OPTIONS

The CSS (Cascading Style Sheet) class structure of jupyter-sphinx is the following:

```
- jupyter_container, jupyter_cell
  - cell_input
  - cell_output
    - stderr
    - output
```

If a code cell is not displayed, the output is provided without the `jupyter_container`. If you want to adjust the styles, add a new stylesheet, e.g. `custom.css`, and adjust your `conf.py` to load it. How you do so depends on the theme you are using.

Here is a sample `custom.css` file overriding the `stderr` background color:

```
.jupyter_container .stderr {
    background-color: #7FFF00;
}
```

Alternatively, you can also completely overwrite the CSS and JS files that are added by Jupyter Sphinx by providing a full copy of a `jupyter-sphinx.css` (which can be empty) file in your `_static` folder. This is also possible with the thebelab CSS and JS that is added.

CHAPTER

ELEVEN

CONFIGURATION OPTIONS

Typically you will be using Sphinx to build documentation for a software package.

If you are building documentation for a Python package you should add the following lines to your sphinx `conf.py`:

```
import os

package_path = os.path.abspath('../..')
os.environ['PYTHONPATH'] = ':'.join((package_path, os.environ.get('PYTHONPATH', '')))
```

This will ensure that your package is importable by any IPython kernels, as they will inherit the environment variables from the main Sphinx process.

Here is a list of all the configuration options available to the Jupyter Sphinx extension:

jupyter_execute_default_kernel

> The default kernel to launch when executing code in `jupyter-execute` directives. The default is `python3`.

render_priority_html

> The priority of different output mimetypes for displaying in HTML output. Mimetypes earlier in the data priority list are preferred over later ones. This is relevant if a code cell produces an output that has several possible representations (e.g. description text or an image). Please open an issue if you find a mimetype that isn't supported, but should be. The default is `['application/vnd.jupyter.widget-view+json', 'text/html', 'image/svg+xml', 'image/png', 'image/jpeg', 'text/latex', 'text/plain']`.

render_priority_latex

> Same, as `render_priority_html`, but for latex. The default is `['image/svg+xml', 'image/png', 'image/jpeg', 'text/latex', 'text/plain']`.

jupyter_execute_kwargs

> Keyword arguments to pass to `nbconvert.preprocessors.execute.executenb`, which controls how code cells are executed. The default is `dict(timeout=-1, allow_errors=True)`.

jupyter_sphinx_linenos

> Whether to show line numbering in all `jupyter-execute` sources.

jupyter_sphinx_continue_linenos

> Whether to continue line numbering from previous cell in all `jupyter-execute` sources.

CHAPTER

TWELVE

CHANGELOG

12.1 Release 0.4.0

- Allow adding inputs and outputs that are not executed using `jupyter-input` and `jupyter-output` directives.
- Improve script handling by using `nbconvert` directly.
- Remove deprecated enabling of the extension as `jupyter_sphinx.execute`.
- Implement different output priorities in HTML and LaTeX builders. In practice this allows to provide a better fallback in PDF output.
- Introduce new `jupyter-download` syntax compatible with Sphinx≥4, `jupyter-download-nb`, `jupyter-download-notebook`, and `jupyter-download-script`
- Do not overwrite CSS and JS if files already exist in _static/, this allows to customize the CSS and JS file.

12.2 Release 0.3.0

- Switch the extension name to `jupyter-sphinx`, deprecate `jupyter-sphinx.execute`.
- Miscellaneous bugfixes following the restructuring of the codebase.