nbconvert Documentation

Release 5.2.0.dev

Jupyter Development Team

May 08, 2017

User Documentation

Using `nbconvert` enables:

- **presentation** of information in familiar formats, such as PDF.
- **publishing** of research using LaTeX and opens the door for embedding notebooks in papers.
- **collaboration** with others who may not use the notebook in their work.
- **sharing** contents with many people via the web using HTML.

Overall, notebook conversion and the `nbconvert` tool give scientists and researchers the flexibility to deliver information in a timely way across different formats.

Primarily, the `nbconvert` tool allows you to convert a Jupyter `.ipynb` notebook document file into another static format including HTML, LaTeX, PDF, Markdown, reStructuredText, and more. `nbconvert` can also add productivity to your workflow when used to execute notebooks programmatically.

If used as a Python library (`import nbconvert`), `nbconvert` adds notebook conversion within a project. For example, `nbconvert` is used to implement the "Download as" feature within the Jupyter Notebook web application. When used as a command line tool (invoked as `jupyter nbconvert ...`), users can conveniently convert just one or a batch of notebook files to another format.

Contents:

CHAPTER 1

Installation

See also:

Installing Jupyter Nbconvert is part of the Jupyter ecosystem.

Installing nbconvert

Nbconvert is packaged for both pip and conda, so you can install it with:

```
pip install nbconvert
# OR
conda install nbconvert
```

If you're new to Python, we recommend installing Anaconda, a Python distribution which includes nbconvert and the other Jupyter components.

Important: To unlock nbconvert's full capabilities requires Pandoc and TeX (specifically, XeLaTeX). These must be installed separately.

Installing Pandoc

For converting markdown to formats other than HTML, nbconvert uses Pandoc (1.12.1 or later).

To install pandoc on Linux, you can generally use your package manager:

```
sudo apt-get install pandoc
```

On other platforms, you can get pandoc from their website.

Installing TeX

For converting to PDF, nbconvert uses the TeX document preparation ecosystem. It produces an intermediate `.tex` file which is compiled by the XeTeX engine with the LaTeX2e format (via the `xelatex` command) to produce PDF output.

New in version 5.0: We use XeTeX as the rendering engine rather than pdfTeX (as in earlier versions). XeTeX can access fonts through native operating system libraries, it has better support for OpenType formatted fonts and Unicode characters.

To install a complete TeX environment (including XeLaTeX and the necessary supporting packages) by hand can be tricky. Fortunately, there are packages that make this much easier. These packages are specific to different operating systems:

- Linux: TeX Live
- macOS (OS X): MacTeX.
- Windows: MikTex

Because nbconvert depends on packages and fonts included in standard TeX distributions, if you do not have a complete installation, you may not be able to use nbconvert's standard tooling to convert notebooks to PDF.

PDF conversion on a limited TeX environment

If you are only able to install a limited TeX environment, there are two main routes you could take to convert to PDF:

1. **Using TeX by hand**
    (a) You could convert to `.tex` directly; this requires Pandoc.
    (b) edit the file to accord with your local environment
    (c) run `xelatex` directly.
2. **Custom exporter**
    (a) You could write a *custom exporter* that takes your system's limitations into account.

CHAPTER 2

Using as a command line tool

The command-line syntax to run the `nbconvert` script is:

```
$ jupyter nbconvert --to FORMAT notebook.ipynb
```

This will convert the Jupyter notebook file `notebook.ipynb` into the output format given by the `FORMAT` string.

Default output format - HTML

The default output format is HTML, for which the `--to` argument may be omitted:

```
$ jupyter nbconvert notebook.ipynb
```

Supported output formats

The currently supported output formats are:

- *HTML*,
- *LaTeX*,
- *PDF*,
- *Reveal.js HTML slideshow*,
- *Markdown*,
- *reStructuredText*,
- *executable script*,
- *notebook*.

Jupyter also provides a few templates for output formats. These can be specified via an additional `--template` argument and are listed in the sections below.

HTML

- `--to html`
  - `--template full` (default)

    A full static HTML render of the notebook. This looks very similar to the interactive view.
  - `--template basic`

    Simplified HTML, useful for embedding in webpages, blogs, etc. This excludes HTML headers.

LaTeX

- `--to latex`

  Latex export. This generates `NOTEBOOK_NAME.tex` file, ready for export.
  - `--template article` (default)

    Latex article, derived from Sphinx's howto template.
  - `--template report`

    Latex report, providing a table of contents and chapters.
  - `--template basic`

    Very basic latex output - mainly meant as a starting point for custom templates.

Note: nbconvert uses pandoc to convert between various markup languages, so pandoc is a dependency when converting to latex or reStructuredText.

PDF

- `--to pdf`

  Generates a PDF via latex. Supports the same templates as `--to latex`.

Reveal.js HTML slideshow

- `--to slides`

  This generates a Reveal.js HTML slideshow. It must be served by an HTTP server. The easiest way to do this is adding `--post serve` on the command-line. The `serve` post-processor proxies Reveal.js requests to a CDN if no local Reveal.js library is present. To make slides that don't require an internet connection, just place the Reveal.js library in the same directory where your_talk.slides.html is located, or point to another directory using the `--reveal-prefix` alias.

Note: In order to designate a mapping from notebook cells to Reveal.js slides, from within the Jupyter notebook, select menu item View –> Cell Toolbar –> Slideshow. That will reveal a drop-down menu on the upper-right of each cell. From it, one may choose from "Slide," "Sub-Slide", "Fragment", "Skip", and "Notes." On conversion, cells designated as "skip" will not be included, "notes" will be included only in presenter notes, etc.

Markdown

- `--to markdown`

  Simple markdown output. Markdown cells are unaffected, and code cells indented 4 spaces.

reStructuredText

- `--to rst`

  Basic reStructuredText output. Useful as a starting point for embedding notebooks in Sphinx docs.

Note: nbconvert uses pandoc to convert between various markup languages, so pandoc is a dependency when converting to latex or reStructuredText.

Executable script

- `--to script`

  Convert a notebook to an executable script. This is the simplest way to get a Python (or other language, depending on the kernel) script out of a notebook. If there were any magics in an Jupyter notebook, this may only be executable from a Jupyter session.

  For example, to convert a Julia notebook to a Julia executable script:

```
jupyter nbconvert --to script my_julia_notebook.ipynb
```

Notebook and preprocessors

- `--to notebook`

  New in version 3.0.

  This doesn't convert a notebook to a different format *per se*, instead it allows the running of nbconvert preprocessors on a notebook, and/or conversion to other notebook formats. For example:

```
jupyter nbconvert --to notebook --execute mynotebook.ipynb
```

This will open the notebook, execute it, capture new output, and save the result in `mynotebook.nbconvert.ipynb`. By default, `nbconvert` will abort conversion if any exceptions occur during execution of a cell. If you specify `--allow-errors` (in addition to the `--execute` flag) then conversion will continue and the output from any exception will be included in the cell output.

The following command:

```
jupyter nbconvert --to notebook --nbformat 3 mynotebook
```

will create a copy of `mynotebook.ipynb` in `mynotebook.v3.ipynb` in version 3 of the notebook format.

If you want to convert a notebook in-place, you can specify the ouptut file to be the same as the input file:

```
jupyter nbconvert --to notebook mynb --output mynb
```

Be careful with that, since it will replace the input file.

Note: nbconvert uses pandoc to convert between various markup languages, so pandoc is a dependency when converting to latex or reStructuredText.

The output file created by `nbconvert` will have the same base name as the notebook and will be placed in the current working directory. Any supporting files (graphics, etc) will be placed in a new directory with the same base name as the notebook, suffixed with `_files`:

```
$ jupyter nbconvert notebook.ipynb
$ ls
notebook.ipynb   notebook.html    notebook_files/
```

For simple single-file output, such as html, markdown, etc., the output may be sent to standard output with:

```
$ jupyter nbconvert --to markdown notebook.ipynb --stdout
```

Converting multiple notebooks

Multiple notebooks can be specified from the command line:

```
$ jupyter nbconvert notebook*.ipynb
$ jupyter nbconvert notebook1.ipynb notebook2.ipynb
```

or via a list in a configuration file, say `mycfg.py`, containing the text:

```
c = get_config()
c.NbConvertApp.notebooks = ["notebook1.ipynb", "notebook2.ipynb"]
```

and using the command:

```
$ jupyter nbconvert --config mycfg.py
```

CHAPTER 3

Using nbconvert as a library

In this notebook, you will be introduced to the programmatic API of nbconvert and how it can be used in various contexts.

A great blog post by [@jakevdp](https://github.com/jakevdp) will be used to demonstrate. This notebook will not focus on using the command line tool. The attentive reader will point-out that no data is read from or written to disk during the conversion process. This is because nbconvert has been designed to work in memory so that it works well in a database or web-based environment too.

Quick overview

Credit: Jonathan Frederic (@jdfreder on github)

The main principle of nbconvert is to instantiate an `Exporter` that controls the pipeline through which notebooks are converted.

First, download @jakevdp's notebook (if you do not have `requests`, install it by running `pip install requests`, or if you don't have pip installed, you can find it on PYPI):

```
In [1]: from urllib.request import urlopen

        url = 'http://jakevdp.github.com/downloads/notebooks/XKCD_plots.ipynb'
        response = urlopen(url).read().decode()
        response[0:60] + ' ...'
```

```
Out[1]: '{\n "metadata": {\n  "name": "XKCD_plots"\n },\n "nbformat": 3,\n ...'
```

The response is a JSON string which represents a Jupyter notebook.

Next, we will read the response using nbformat. Doing this will guarantee that the notebook structure is valid. Note that the in-memory format and on disk format are slightly different. In particual, on disk, multiline strings might be split into a list of strings.

```
In [2]: import nbformat
        jake_notebook = nbformat.reads(response, as_version=4)
        jake_notebook.cells[0]
```

```
Out[2]: {'cell_type': 'markdown',
         'metadata': {},
         'source': '# XKCD plots in Matplotlib'}
```

The nbformat API returns a special type of dictionary. For this example, you don't need to worry about the details of the structure (if you are interested, please see the nbformat documentation).

The nbconvert API exposes some basic exporters for common formats and defaults. You will start by using one of them. First, you will import one of these exporters (specifically, the HTML exporter), then instantiate it using most of the defaults, and then you will use it to process the notebook we downloaded earlier.

```
In [3]: from traitlets.config import Config

        # 1. Import the exporter
        from nbconvert import HTMLExporter

        # 2. Instantiate the exporter. We use the `basic` template for now; we'll get into more deta
        # later about how to customize the exporter further.
        html_exporter = HTMLExporter()
        html_exporter.template_file = 'basic'

        # 3. Process the notebook we loaded earlier
        (body, resources) = html_exporter.from_notebook_node(jake_notebook)
```

The exporter returns a tuple containing the source of the converted notebook, as well as a resources dict. In this case, the source is just raw HTML:

```
In [4]: print(body[:400] + '...')
```

```
<div class="cell border-box-sizing text_cell rendered">
<div class="prompt input_prompt">
</div>
<div class="inner_cell">
<div class="text_cell_render border-box-sizing rendered_html">
<h1 id="XKCD-plots-in-Matplotlib">XKCD plots in Matplotlib<a class="anchor-link" href="#XKCD-plots-in
</div>
</div>
</div>
<div class="cell border-box-sizing text_cell rendered">
<div cl...
```

If you understand HTML, you'll notice that some common tags are ommited, like the `body` tag. Those tags are included in the default `HtmlExporter`, which is what would have been constructed if we had not modified the `template_file`.

The resource dict contains (among many things) the extracted `.png`, `.jpg`, etc. from the notebook when applicable. The basic HTML exporter leaves the figures as embedded base64, but you can configure it to extract the figures. So for now, the resource dict should be mostly empty, except for a key containing CSS and a few others whose content will be obvious:

```
In [5]: print("Resources:", resources.keys())
        print("Metadata:", resources['metadata'].keys())
        print("Inlining:", resources['inlining'].keys())
        print("Extension:", resources['output_extension'])
```

```
Resources: dict_keys(['metadata', 'output_extension', 'raw_mimetypes', 'inlining'])
Metadata: dict_keys(['name'])
Inlining: dict_keys(['css'])
Extension: .html
```

`Exporters` are stateless, so you won't be able to extract any useful information beyond their configuration. You can

re-use an exporter instance to convert another notebook. In addition to the `from_notebook_node` used above, each exporter exposes `from_file` and `from_filename` methods.

Extracting Figures using the RST Exporter

When exporting, you may want to extract the base64 encoded figures as files. While the HTML exporter does not do this by default, the `RstExporter` does:

```
In [6]: # Import the RST exproter
        from nbconvert import RSTExporter
        # Instantiate it
        rst_exporter = RSTExporter()
        # Convert the notebook to RST format
        (body, resources) = rst_exporter.from_notebook_node(jake_notebook)

        print(body[:970] + '...')
        print('[.....]')
        print(body[800:1200] + '...')
```

```
XKCD plots in Matplotlib
========================

This notebook originally appeared as a blog post at `Pythonic
Perambulations <http://jakevdp.github.com/blog/2012/10/07/xkcd-style-plots-in-matplotlib/>`__
by Jake Vanderplas.

.. raw:: html

   <!-- PELICAN_BEGIN_SUMMARY -->

*Update: the matplotlib pull request has been merged! See* `*This
post* <http://jakevdp.github.io/blog/2013/07/10/XKCD-plots-in-matplotlib/>`__
*for a description of the XKCD functionality now built-in to
matplotlib!*

One of the problems I've had with typical matplotlib figures is that
everything in them is so precise, so perfect. For an example of what I
mean, take a look at this figure:

.. code:: python

    from IPython.display import Image
    Image('http://jakevdp.github.com/figures/xkcd_version.png')




.. image:: output_3_0.png



Sometimes when showing schematic plots, this is the type of figure I
want to display. But drawing it by hand is a pain: I'd rather just use
matp...
[.....]
image:: output_3_0.png
```

```
Sometimes when showing schematic plots, this is the type of figure I
want to display. But drawing it by hand is a pain: I'd rather just use
matplotlib. The problem is, matplotlib is a bit too precise. Attempting
to duplicate this figure in matplotlib leads to something like this:

.. code:: python

    Image('http://jakevdp.github.com/figures/mpl_version.png')



.. imag...
```

Notice that base64 images are not embeded, but instead there are filename-like strings, such as `output_3_0.png`. The strings actually are (configurable) keys that map to the binary data in the resources dict.

Note, if you write an RST Plugin, you are responsible for writing all the files to the disk (or uploading, etc...) in the right location. Of course, the naming scheme is configurable.

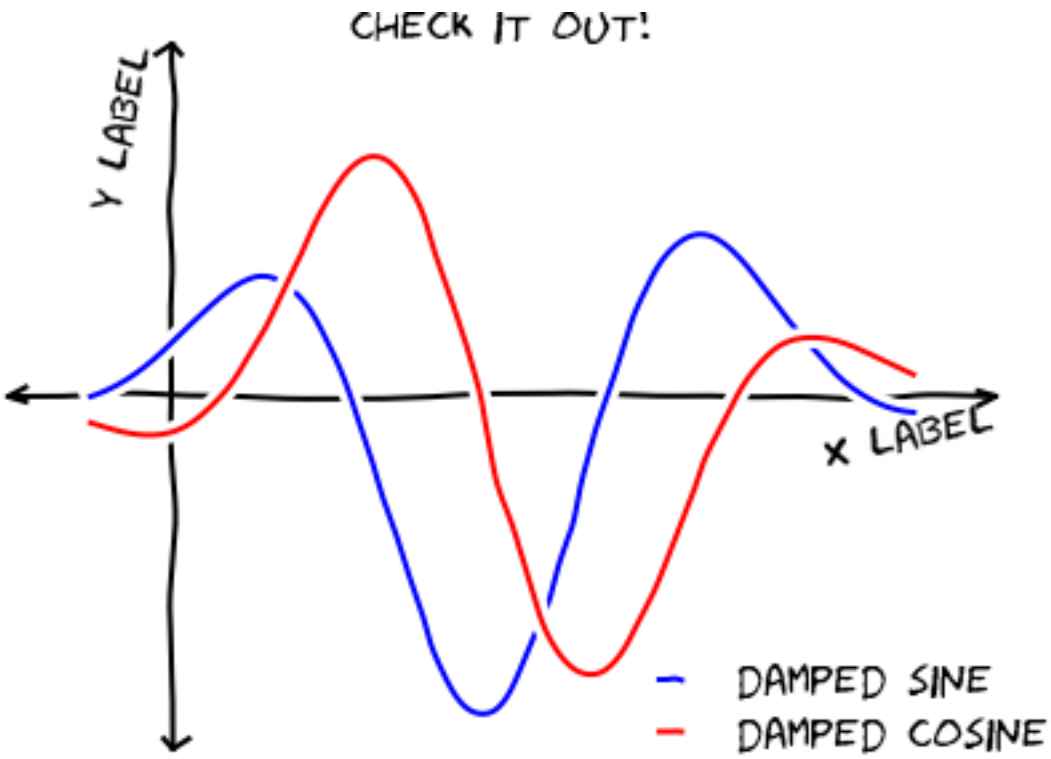
As an exercise, this notebook will show you how to get one of those images. First, take a look at the `'outputs'` of the returned resources dictionary. This is a dictionary that contains a key for each extracted resource, with values corresponding to the actual base64 encoding:

```
In [7]: sorted(resources['outputs'].keys())
```

```
Out[7]: ['output_13_1.png',
         'output_16_0.png',
         'output_18_1.png',
         'output_3_0.png',
         'output_5_0.png']
```

In this case, there are 5 extracted binary figures, all `pngs`. We can use the Image display object to actually display one of the images:

```
In [8]: from IPython.display import Image
        Image(data=resources['outputs']['output_3_0.png'], format='png')
```



Note that this image is being rendered without ever reading or writing to the disk.

Extracting Figures using the HTML Exporter

As mentioned above, by default, the HTML exporter does not extract images – it just leaves them as inline base64 encodings. However, this is not always what you might want. For example, here is a use case from @jakevdp:

> I write an awesome blog using Jupyter notebooks converted to HTML, and I want the images to be cached. Having one html file with all of the images base64 encoded inside it is nice when sharing with a coworker, but for a website, not so much. I need an HTML exporter, and I want it to extract the figures!

Some theory

Before we get into actually extracting the figures, it will be helpful to give a high-level overview of the process of converting a notebook to a another format:

1. Retrieve the notebook and it's accompanying resources (you are responsible for this).
2. Feed the notebook into the `Exporter`, which:
   (a) Sequentially feeds the notebook into an array of `Preprocessors`. Preprocessors only act on the **structure** of the notebook, and have unrestricted access to it.
   (b) Feeds the notebook into the Jinja templating engine, which converts it to a particular format depending on which template is selected.
3. The exporter returns the converted notebook and other relevant resources as a tuple.
4. You write the data to the disk using the built-in `FilesWriter` (which writes the notebook and any extracted files to disk), or elsewhere using a custom `Writer`.

Using different preprocessors

To extract the figures when using the HTML exporter, we will want to change which `Preprocessors` we are using. There are several preprocessors that come with nbconvert, including one called the `ExtractOutputPreprocessor`.

The `ExtractOutputPreprocessor` is responsible for crawling the notebook, finding all of the figures, and putting them into the resources directory, as well as choosing the key (i.e. `filename_xx_y.extension`) that can replace the figure inside the template. To enable the `ExtractOutputPreprocessor`, we must add it to the exporter's list of preprocessors:

```
In [9]: # create a configuration object that changes the preprocessors
        from traitlets.config import Config
        c = Config()
        c.HTMLExporter.preprocessors = ['nbconvert.preprocessors.ExtractOutputPreprocessor']

        # create the new exporter using the custom config
        html_exporter_with_figs = HTMLExporter(config=c)
        html_exporter_with_figs.preprocessors

Out[9]: ['nbconvert.preprocessors.ExtractOutputPreprocessor']
```

We can compare the result of converting the notebook using the original HTML exporter and our new customized one:

```
In [10]: (_, resources)           = html_exporter.from_notebook_node(jake_notebook)
         (_, resources_with_fig) = html_exporter_with_figs.from_notebook_node(jake_notebook)

         print("resources without figures:")
         print(sorted(resources.keys()))

         print("\nresources with extracted figures (notice that there's one more field called 'output
         print(sorted(resources_with_fig.keys()))

         print("\nthe actual figures are:")
         print(sorted(resources_with_fig['outputs'].keys()))
```

```
resources without figures:
['inlining', 'metadata', 'output_extension', 'raw_mimetypes']

resources with extracted figures (notice that there's one more field called 'outputs'):
['inlining', 'metadata', 'output_extension', 'outputs', 'raw_mimetypes']

the actual figures are:
['output_13_1.png', 'output_16_0.png', 'output_18_1.png', 'output_3_0.png', 'output_5_0.png']
```

Custom Preprocessors

There are an endless number of transformations that you may want to apply to a notebook. In particularly complicated cases, you may want to actually create your own `Preprocessor`. Above, when we customized the list of preprocessors accepted by the `HTMLExporter`, we passed in a string – this can be any valid module name. So, if you create your own preprocessor, you can include it in that same list and it will be used by the exporter.

To create your own preprocessor, you will need to subclass from `nbconvert.preprocessors.Preprocessor` and overwrite either the `preprocess` and/or `preprocess_cell` methods.

Example

The following demonstration adds the ability to exclude a cell by index.

Note: injecting cells is similar, and won't be covered here. If you want to inject static content at the beginning/end of a notebook, use a custom template.

```
In [11]: from traitlets import Integer
         from nbconvert.preprocessors import Preprocessor

         class PelicanSubCell(Preprocessor):
             """A Pelican specific preprocessor to remove some of the cells of a notebook"""

             # I could also read the cells from nb.metadata.pelican if someone wrote a JS extension,
             # but for now I'll stay with configurable value.
             start = Integer(0,  help="first cell of notebook to be converted")
             end   = Integer(-1, help="last cell of notebook to be converted")
             start.tag(config='True')
             end.tag(config='True')

             def preprocess(self, nb, resources):
                 self.log.info("I'll keep only cells from %d to %d", self.start, self.end)
                 nb.cells = nb.cells[self.start:self.end]
                 return nb, resources
```

Here a Pelican exporter is created that takes `PelicanSubCell` preprocessors and a `config` object as parameters. This may seem redundant, but with the configuration system you can register an inactive preprocessor on all of the exporters and activate it from config files or the command line.

```
In [12]: # Create a new config object that configures both the new preprocessor, as well as the expo
         c =  Config()
         c.PelicanSubCell.start = 4
         c.PelicanSubCell.end = 6
         c.RSTExporter.preprocessors = [PelicanSubCell]

         # Create our new, customized exporter that uses our custom preprocessor
         pelican = RSTExporter(config=c)

         # Process the notebook
         print(pelican.from_notebook_node(jake_notebook)[0])
```

```
Sometimes when showing schematic plots, this is the type of figure I
want to display. But drawing it by hand is a pain: I'd rather just use
matplotlib. The problem is, matplotlib is a bit too precise. Attempting
to duplicate this figure in matplotlib leads to something like this:

.. code:: python

    Image('http://jakevdp.github.com/figures/mpl_version.png')



.. image:: output_5_0.png
```

Programatically creating templates

```
In [13]: from jinja2 import DictLoader

         dl = DictLoader({'full.tpl':
         """
         {%- extends 'basic.tpl' -%}

         {% block footer %}
         FOOOOOOOOTEEEEER
         {% endblock footer %}
         """})


         exportHTML = HTMLExporter(extra_loaders=[dl])
         (body, resources) = exportHTML.from_notebook_node(jake_notebook)
         for l in body.split('\n')[-4:]:
             print(l)
```

```
</div>
</div>
FOOOOOOOOTEEEEER
```

Real World Uses

@jakevdp uses Pelican and Jupyter Notebook to blog. Pelican will use nbconvert programatically to generate blog post. Have a look a Pythonic Preambulations for Jake's blog post.

@damianavila wrote the Nikola Plugin to write blog post as Notebooks and is developping a js-extension to publish notebooks via one click from the web app.

As @Mbussonn requested... easieeeeer! Deploy your Nikola site with just a click in the IPython notebook! http://t.co/860sJunZvj cc @ralsina

— Damián Avila (@damian_avila) August 21, 2013

CHAPTER 4

LaTeX citations

`nbconvert` now has support for LaTeX citations. With this capability you can:

- Manage citations using BibTeX.
- Cite those citations in Markdown cells using HTML data attributes.
- Have `nbconvert` generate proper LaTeX citations and run BibTeX.

For an example of how this works, please see the citations example in the nbconvert-examples repository.

CHAPTER 5

Executing notebooks

Jupyter notebooks are often saved with output cells that have been cleared. nbconvert provides a convenient way to execute the input cells of an .ipynb notebook file and save the results, both input and output cells, as a .ipynb file.

In this section we show how to execute a `.ipynb` notebook document saving the result in notebook format. If you need to export notebooks to other formats, such as reStructured Text or Markdown (optionally executing them) see section *Using nbconvert as a library*.

Executing notebooks can be very helpful, for example, to run all notebooks in Python library in one step, or as a way to automate the data analysis in projects involving more than one notebook.

Executing notebooks from the command line

The same functionality of executing notebooks is exposed through a *command line interface* or a Python API interface. As an example, a notebook can be executed from the command line with:

```
jupyter nbconvert --to notebook --execute mynotebook.ipynb
```

Executing notebooks using the Python API interface

This section will illustrate the Python API interface.

Example

Let's start with a complete quick example, leaving detailed explanations to the following sections.

Import: First we import nbconvert and the *`ExecutePreprocessor`* class:

```
import nbformat
from nbconvert.preprocessors import ExecutePreprocessor
```

Load: Assuming that `notebook_filename` contains the path of a notebook, we can load it with:

```
with open(notebook_filename) as f:
    nb = nbformat.read(f, as_version=4)
```

Configure: Next, we configure the notebook execution mode:

```
ep = ExecutePreprocessor(timeout=600, kernel_name='python3')
```

We specified two (optional) arguments `timeout` and `kernel_name`, which define respectively the cell execution timeout and the execution kernel.

> The option to specify **kernel_name** is new in nbconvert 4.2. When not specified or when using nbconvert <4.2, the default Python kernel is chosen.

Execute/Run (preprocess): To actually run the notebook we call the method `preprocess`:

```
ep.preprocess(nb, {'metadata': {'path': 'notebooks/'}})
```

Hopefully, we will not get any errors during the notebook execution (see the last section for error handling). Note that `path` specifies in which folder to execute the notebook.

Save: Finally, save the resulting notebook with:

```
with open('executed_notebook.ipynb', 'wt') as f:
    nbformat.write(nb, f)
```

That's all. Your executed notebook will be saved in the current folder in the file `executed_notebook.ipynb`.

Execution arguments (traitlets)

The arguments passed to *`ExecutePreprocessor`* are configuration options called traitlets. There are many cool things about traitlets. For example, they enforce the input type, and they can be accessed/modified as class attributes. Moreover, each traitlet is automatically exposed as command-line options. For example, we can pass the timeout from the command-line like this:

```
jupyter nbconvert --ExecutePreprocessor.timeout=600 --to notebook --execute␣
↪mynotebook.ipynb
```

Let's now discuss in more detail the two traitlets we used.

The `timeout` traitlet defines the maximum time (in seconds) each notebook cell is allowed to run, if the execution takes longer an exception will be raised. The default is 30 s, so in cases of long-running cells you may want to specify an higher value. The `timeout` option can also be set to `None` or `-1` to remove any restriction on execution time.

The second traitlet, `kernel_name`, allows specifying the name of the kernel to be used for the execution. By default, the kernel name is obtained from the notebook metadata. The traitlet `kernel_name` allows specifying a user-defined kernel, overriding the value in the notebook metadata. A common use case is that of a Python 2/3 library which includes documentation/testing notebooks. These notebooks will specify either a python2 or python3 kernel in their metadata (depending on the kernel used the last time the notebook was saved). In reality, these notebooks will work on both Python 2 and Python 3, and, for testing, it is important to be able to execute them programmatically on both versions. Here the traitlet `kernel_name` helps simplify and maintain consistency: we can just run a notebook twice, specifying first "python2" and then "python3" as the kernel name.

Handling errors and exceptions

In the previous sections we saw how to save an executed notebook, assuming there are no execution errors. But, what if there are errors?

Execution until first error

An error during the notebook execution, by default, will stop the execution and raise a `CellExecutionError`. Conveniently, the source cell causing the error and the original error name and message are also printed. After an error, we can still save the notebook as before:

```
with open('executed_notebook.ipynb', mode='wt') as f:
    nbformat.write(nb, f)
```

The saved notebook contains the output up until the failing cell, and includes a full stack-trace and error (which can help debugging).

Handling errors

A useful pattern to execute notebooks while handling errors is the following:

```
try:
    out = ep.preprocess(nb, {'metadata': {'path': run_path}})
except CellExecutionError:
    msg = 'Error executing the notebook "%s".\n\n' % notebook_filename
    msg += 'See notebook "%s" for the traceback.' % notebook_filename_out
    print(msg)
    raise
finally:
    with open(notebook_filename_out, mode='wt') as f:
        nbformat.write(nb, f)
```

This will save the executed notebook regardless of execution errors. In case of errors, however, an additional message is printed and the `CellExecutionError` is raised. The message directs the user to the saved notebook for further inspection.

Execute and save all errors

As a last scenario, it is sometimes useful to execute notebooks which raise exceptions, for example to show an error condition. In this case, instead of stopping the execution on the first error, we can keep executing the notebook using the traitlet `allow_errors` (default is False). With `allow_errors=True`, the notebook is executed until the end, regardless of any error encountered during the execution. The output notebook, will contain the stack-traces and error messages for **all** the cells raising exceptions.

CHAPTER 6

Configuration options

Configuration options may be set in a file, `~/.jupyter/jupyter_nbconvert_config.py`, or at the command line when starting nbconvert, i.e. `jupyter nbconvert --Application.log_level=10`.

Application.log_datefmt [Unicode] Default: `'%Y-%m-%d %H:%M:%S'`

The date format used by logging formatters for %(asctime)s

Application.log_format [Unicode] Default: `'[%(name)s]%(highlevel)s %(message)s'`

The Logging format template

Application.log_level [0|10|20|30|40|50|'DEBUG'|'INFO'|'WARN'|'ERROR'|'CRITICAL'] Default: `30`

Set the log level by value or name.

JupyterApp.answer_yes [Bool] Default: `False`

Answer yes to any prompts.

JupyterApp.config_file [Unicode] Default: `''`

Full path of a config file.

JupyterApp.config_file_name [Unicode] Default: `''`

Specify a config file to load.

JupyterApp.generate_config [Bool] Default: `False`

Generate default config file.

NbConvertApp.export_format [Unicode] Default: `'html'`

The export format to be used, either one of the built-in formats, or a dotted object name that represents the import path for an *Exporter* class

NbConvertApp.from_stdin [Bool] Default: `False`

read a single notebook from stdin.

NbConvertApp.notebooks [List] Default: `[]`

List of notebooks to convert. Wildcards are supported. Filenames passed positionally will be added to the list.

NbConvertApp.output_base [Unicode] Default: `''`

overwrite base name use for output files. can only be used when converting one notebook at a time.

NbConvertApp.output_files_dir [Unicode] Default: `'{notebook_name}_files'`

Directory to copy extra files (figures) to. '{notebook_name}' in the string will be converted to notebook basename

NbConvertApp.postprocessor_class [DottedOrNone] Default: `''`

PostProcessor class used to write the results of the conversion

NbConvertApp.use_output_suffix [Bool] Default: `True`

Whether to apply a suffix prior to the extension (only relevant when converting to notebook format). The suffix is determined by the exporter, and is usually '.nbconvert'.

NbConvertApp.writer_class [DottedObjectName] Default: `'FilesWriter'`

Writer class used to write the results of the conversion

NbConvertBase.default_language [Unicode] Default: `'ipython'`

Deprecated default highlight language as of 5.0, please use language_info metadata instead

NbConvertBase.display_data_priority [List] Default: `['text/html', 'application/pdf', 'text/latex', 'image/svg+xml...`

An ordered list of preferred output type, the first encountered will usually be used when converting discarding the others.

Exporter.default_preprocessors [List] Default: `['nbconvert.preprocessors.ClearOutputPreprocessor', 'nbconver...`

List of preprocessors available by default, by name, namespace, instance, or type.

Exporter.file_extension [FilenameExtension] Default: `'.txt'`

Extension of the file that should be written to disk

Exporter.preprocessors [List] Default: `[]`

List of preprocessors, by name or namespace, to enable.

TemplateExporter.filters [Dict] Default: `{}`

Dictionary of filters, by name and namespace, to add to the Jinja environment.

TemplateExporter.raw_mimetypes [List] Default: `[]`

formats of raw cells to be included in this Exporter's output.

TemplateExporter.template_extension [Unicode] Default: `'.tpl'`

No description

TemplateExporter.template_file [Unicode] Default: `''`

Name of the template file to use

TemplateExporter.template_path [List] Default: `['.']`

No description

LatexExporter.template_extension [Unicode] Default: `'.tplx'`

No description

NotebookExporter.nbformat_version [1|2|3|4] Default: `4`

The nbformat version to write. Use this to downgrade notebooks.

PDFExporter.bib_command [List] Default: `['bibtex', '{filename}']`

Shell command used to run bibtex.

PDFExporter.latex_command [List] Default: `['xelatex', '{filename}']`

Shell command used to compile latex.

PDFExporter.latex_count [Int] Default: `3`

How many times latex will be called.

PDFExporter.temp_file_exts [List] Default: `['.aux', '.bbl', '.blg', '.idx', '.log', '.out']`

File extensions of temp files to remove after running.

PDFExporter.verbose [Bool] Default: `False`

Whether to display the output of latex commands.

SlidesExporter.reveal_url_prefix [Unicode] Default: `''`

The URL prefix for reveal.js. This can be a a relative URL for a local copy of reveal.js, or point to a CDN.

For speaker notes to work, a local reveal.js prefix must be used.

Preprocessor.enabled [Bool] Default: `False`

No description

CSSHTMLHeaderPreprocessor.highlight_class [Unicode] Default: `'.highlight'`

CSS highlight class identifier

ConvertFiguresPreprocessor.from_format [Unicode] Default: `''`

Format the converter accepts

ConvertFiguresPreprocessor.to_format [Unicode] Default: `''`

Format the converter writes

ExecutePreprocessor.allow_errors [Bool] Default: `False`

If *False* (default), when a cell raises an error the execution is stopped and a *CellExecutionError* is raised. If *True*, execution errors are ignored and the execution is continued until the end of the notebook. Output from exceptions is included in the cell output in both cases.

ExecutePreprocessor.interrupt_on_timeout [Bool] Default: `False`

If execution of a cell times out, interrupt the kernel and continue executing other cells rather than throwing an error and stopping.

ExecutePreprocessor.iopub_timeout [Int] Default: `4`

The time to wait (in seconds) for IOPub output. This generally doesn't need to be set, but on some slow networks (such as CI systems) the default timeout might not be long enough to get all messages.

ExecutePreprocessor.kernel_manager_class [Type] Default: `'jupyter_client.manager.KernelManager'`

The kernel manager class to use.

ExecutePreprocessor.kernel_name [Unicode] Default: `''`

Name of kernel to use to execute the cells. If not set, use the kernel_spec embedded in the notebook.

ExecutePreprocessor.raise_on_iopub_timeout [Bool] Default: `False`

If *False* (default), then the kernel will continue waiting for iopub messages until it receives a kernel idle message, or until a timeout occurs, at which point the currently executing cell will be skipped. If *True*, then an error will be raised after the first timeout. This option generally does not need to be used, but may be useful in contexts where there is the possibility of executing notebooks with memory-consuming infinite loops.

ExecutePreprocessor.shutdown_kernel ['graceful'|'immediate'] Default: `'graceful'`

If *graceful* (default), then the kernel is given time to clean up after executing all cells, e.g., to execute its *atexit* hooks. If *immediate*, then the kernel is signaled to immediately terminate.

ExecutePreprocessor.timeout [Int] Default: `30`

The time to wait (in seconds) for output from executions. If a cell execution takes longer, an exception (TimeoutError on python 3+, RuntimeError on python 2) is raised.

None or *-1* will disable the timeout. If *timeout_func* is set, it overrides *timeout*.

ExecutePreprocessor.timeout_func [Any] Default: `None`

A callable which, when given the cell source as input, returns the time to wait (in seconds) for output from cell executions. If a cell execution takes longer, an exception (TimeoutError on python 3+, RuntimeError on python 2) is raised.

Returning *None* or *-1* will disable the timeout for the cell. Not setting *timeout_func* will cause the preprocessor to default to using the *timeout* trait for all cells. The *timeout_func* trait overrides *timeout* if it is not *None*.

ExtractOutputPreprocessor.extract_output_types [Set] Default: `{'application/pdf', 'image/png', 'image/jpeg', 'image/svg+xml'}`

No description

ExtractOutputPreprocessor.output_filename_template [Unicode] Default: `'{unique_key}_{cell_index}_{index}{ext`

No description

HighlightMagicsPreprocessor.languages [Dict] Default: `{}`

Syntax highlighting for magic's extension languages. Each item associates a language magic extension such as %%R, with a pygments lexer such as r.

SVG2PDFPreprocessor.command [Unicode] Default: `''`

The command to use for converting SVG to PDF

This string is a template, which will be formatted with the keys to_filename and from_filename.

The conversion call must read the SVG from {from_flename}, and write a PDF to {to_filename}.

SVG2PDFPreprocessor.inkscape [Unicode] Default: `''`

The path to Inkscape, if necessary

WriterBase.files [List] Default: `[]`

List of the files that the notebook references. Files will be included with written output.

FilesWriter.build_directory [Unicode] Default: `''`

Directory to write output(s) to. Defaults to output to the directory of each notebook. To recover previous default behaviour (outputting to the current working directory) use . as the flag value.

FilesWriter.relpath [Unicode] Default: `''`

When copying files that the notebook depends on, copy them in relation to this path, such that the destination filename will be os.path.relpath(filename, relpath). If FilesWriter is operating on a notebook that already exists elsewhere on disk, then the default will be the directory containing that notebook.

ServePostProcessor.ip [Unicode] Default: `'127.0.0.1'`

The IP address to listen on.

ServePostProcessor.open_in_browser [Bool] Default: `True`

Should the browser be opened automatically?

ServePostProcessor.port [Int] Default: `8000`

port for the server to listen on.

ServePostProcessor.reveal_cdn [Unicode] Default: `'https://cdnjs.cloudflare.com/ajax/libs/reveal.js/3.1.0'`

URL for reveal.js CDN.

ServePostProcessor.reveal_prefix [Unicode] Default: `'reveal.js'`

URL prefix for reveal.js

CHAPTER 7

Customizing nbconvert

Under the hood, nbconvert uses Jinja templates to specify how the notebooks should be formatted. These templates can be fully customized, allowing you to use nbconvert to create notebooks in different formats with different styles as well.

Converting a notebook to an (I)Python script and printing to stdout

Out of the box, nbconvert can be used to convert notebooks to plain Python files. For example, the following command converts the `example.ipynb` notebook to Python and prints out the result:

```
In [1]: !jupyter nbconvert --to python 'example.ipynb' --stdout
```

```
[NbConvertApp] Converting notebook example.ipynb to python

# coding: utf-8

# # Example notebook

# ### Markdown cells
#
# This is an example notebook that can be converted with `nbconvert` to different formats. This is ar

# ### LaTeX Equations
#
# Here is an equation:
#
# $$
```

From the code, you can see that non-code cells are also exported. If you wanted to change that behaviour, you would first look to nbconvert *configuration options page* to see if there is an option available that can give you your desired behaviour.

In this case, if you wanted to remove code cells from the output, you could use the `TemplateExporter.exclude_markdown` traitlet directly, as below.

```
In [2]: !jupyter nbconvert --to python 'example.ipynb' --stdout --TemplateExporter.exclude_markdown=1
```

```
[NbConvertApp] Converting notebook example.ipynb to python

# coding: utf-8

# In[1]:


print("This is a code cell that produces some output")


# In[1]:


import matplotlib.pyplot as plt
import numpy as np
plt.ion()

x = np.linspace(0, 2 * np.pi, 100)
y = np.sin(x)
plt.plot(x, y)
```

Custom Templates

As mentioned above, if you want to change this behavior, you can use a custom template. The custom template inherits from the Python template and overwrites the markdown blocks so that they are empty.

Below is an example of a custom template, which we write to a file called `simplepython.tpl`. This template removes markdown cells from the output, and also changes how the execution count numbers are formatted:

```
In [3]: %%writefile simplepython.tpl

        {% extends 'python.tpl'%}

        ## remove markdown cells
        {% block markdowncell -%}
        {% endblock markdowncell %}

        ## change the appearance of execution count
        {% block in_prompt %}
        # [{{ cell.execution_count if cell.execution_count else ' ' }}]:
        {% endblock in_prompt %}
```

```
Overwriting simplepython.tpl
```

Using this template, we see that the resulting Python code does not contain anything that was previously in a markdown cell, and only displays execution counts (i.e., `[#]:` not `In[#]:`):

```
In [4]: !jupyter nbconvert --to python 'example.ipynb' --stdout --template=simplepython.tpl
```

```
[NbConvertApp] Converting notebook example.ipynb to python


# coding: utf-8

# [1]:

print("This is a code cell that produces some output")
```

```
# [1]:

import matplotlib.pyplot as plt
import numpy as np
plt.ion()

x = np.linspace(0, 2 * np.pi, 100)
y = np.sin(x)
plt.plot(x, y)
```

Template structure

Nbconvert templates consist of a set of nested blocks. When defining a new template, you extend an existing template by overriding some of the blocks.

All the templates shipped in nbconvert have the basic structure described here, though some may define additional blocks.

```
In [5]: from IPython.display import HTML, display
        with open('template_structure.html') as f:
            display(HTML(f.read()))
```

```
<IPython.core.display.HTML object>
```

A few gotchas

Jinja blocks use `{% %}` by default which does not play nicely with LaTeX, so those are replaced by `((* *))` in LaTeX templates.

Templates using cell tags

The notebook file format supports attaching arbitrary JSON metadata to each cell. In addition, every cell has a special `tags` metadata field that accepts a list of strings that indicate the cell's tags. To apply these, go to the `View` → `CellToolbar` → `Tags` option which will create a Tag editor at the top of every cell.

First choose a notebook you want to convert to html, and apply the tags: `"Easy"`, `"Medium"`, or `"Hard"`.

With this in place, the notebook can be converted using a custom template.

Design your template in the cells provided below.

Hint: tags are located at `cell.metadata.tags`, the following Python code collects the value of the tag:

```
cell['metadata'].get('tags', [])
```

Which you can then use inside a Jinja template as in the following:

```
In [6]: %%writefile mytemplate.tpl

        {% extends 'full.tpl'%}
        {% block any_cell %}
        {% if 'Hard' in cell['metadata'].get('tags', []) %}
            <div style="border:thin solid red">
                {{ super() }}
```

```
            </div>
        {% elif 'Medium' in cell['metadata'].get('tags', []) %}
            <div style="border:thin solid orange">
                {{ super() }}
            </div>
        {% elif 'Easy' in cell['metadata'].get('tags', []) %}
            <div style="border:thin solid green">
                {{ super() }}
            </div>
        {% else %}
            {{ super() }}
        {% endif %}
        {% endblock any_cell %}
```

```
Overwriting mytemplate.tpl
```

Now, if we collect the result of using nbconvert with this template, and display the resulting html, we see the following:

```
In [7]: example = !jupyter nbconvert --to html 'example.ipynb' --template=mytemplate.tpl --stdout
        example = example[3:] # have to remove the first three lines which are not proper html
        from IPython.display import HTML, display
        display(HTML('\n'.join(example)))
```

```
<IPython.core.display.HTML object>
```

Templates using custom cell metadata

We demonstrated *above* how to use cell tags in a template to apply custom styling to a notebook. But remember, the notebook file format supports attaching *arbitrary* JSON metadata to each cell, not only cell tags. Here, we describe an exercise for using an `example.difficulty` metadata field (rather than cell tags) to do the same as before (to mark up different cells as being "Easy", "Medium" or "Hard").

How to edit cell metadata

To edit the cell metadata from within the notebook, go to the menu item: `View → Cell Toolbar → Edit Metadata`. This will bring up a toolbar above each cell with a button that says "Edit Metadata". Click this button, and a field will pop up in which you will directly edit the cell metadata JSON.

NB: Because it is JSON, you will need to ensure that what you write is valid JSON.

Template challenges: dealing with missing custom metadata fields

One of the challenges of dealing with custom metadata is to handle the case where the metadata is not present on every cell. This can get somewhat tricky because of JSON objects tendency to be deeply nested coupled with Python's (and therefore Jinja's) approach to calling into dictionaries. Specifically, the following code will error:

```
foo = {}
foo["bar"]
```

Accordingly, it is better to use the `` `{}.get `` method <https://docs.python.org/3.6/library/stdtypes.html#dict.get>`__ which allows you to set a default value to return if no key is found as the second argument.

Hint: if your metadata items are located at `cell.metadata.example.difficulty`, the following Python code would get the value defaulting to an empty string (`''`) if nothing is found:

```
cell['metadata'].get('example', {}).get('difficulty', '')
```

Exercise: Write a template for handling custom metadata

Now, write a template that will look for `Easy`, `Medium` and `Hard` metadata values for the `cell.metadata.example.difficulty` field and wrap them in a div with a green, orange, or red thin solid border (respectively).

NB: This is the same design and logic as used in the previous cell tag example.

We have provided an example file in `example.ipynb` in the nbconvert documentation that has already been marked up with both tags and the above metadata for you to test with. You can get it from this link to the raw file or by cloning the repository from GitHub and navingating to `nbconvert/docs/source/example.ipynb`.

First, make sure that you can reproduce the previous result using the cell tags template that we have provided above.

Easy: If you want to make it easy on yourself, create a new file `my_template.tpl` in the same directory as `example.ipynb` and copy the contents of the cell we use to write `mytemplate.tpl` to the file system.

Then run `jupyter nbconvert --to html 'example.ipynb' --template=mytemplate.tpl` and see if your

Moderate: If you want more of a challenge, try recreating the jinja template by modifying the following jinja template file:

```
{% extends 'full.tpl'%}
{% block any_cell %}
    <div style="border:thin solid red">
        {{ super() }}
    </div>
{% endblock any_cell %}
```

Hard: If you want even more of a challenge, try recreating the jinja template from scratch.

Once you've done at least the **Easy** version of the previous step, try modifying your template to use `cell.metadata.example.difficulty` fields rather than cell tags.

Once you've written your template, try converting `example.ipynb` using the following command (making sure that `your_template.tpl` is in your local directory where you are running the command):

```
jupyter nbconvert --to html 'example.ipynb' --template=your_template.tpl --stdout
```

The resulting display should pick out different cells to be bordered with green, orange, or red.

If you do that successfullly, the resulting html document should look like the following cell's contents:

example

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/mathjax/2.7.0/MathJax.js?
↪config=TeX-AMS_HTML"></script>
<!-- MathJax configuration -->
<script type="text/x-mathjax-config">
MathJax.Hub.Config({
    tex2jax: {
        inlineMath: [ ['$','$'], ["\\(","\\)"] ],
        displayMath: [ ['$$','$$'], ["\\[","\\]"] ],
        processEscapes: true,
        processEnvironments: true
    },
```

```
    // Center justify equations in code and markdown cells. Elsewhere
    // we use CSS to left justify single line equations in code cells.
    displayAlign: 'center',
    "HTML-CSS": {
        styles: {'.MathJax_Display': {"margin": 0}},
        linebreaks: { automatic: true }
    }
});
</script>
<!-- End of mathjax configuration --></head>
```

```
<div class="container" id="notebook-container">


<div style="border:thin solid red">
```

Example notebook¶

```
</div>
```

Markdown cells¶

This is an example notebook that can be converted with nbconvert to different formats. This is an example of a markdown cell.

LaTeX Equations¶

Here is an equation:

```
<div style="border:thin solid green">
```

Code cells¶

```
</div>
```

In [1]:

```
<div class="input_area">
```

Inline figures¶

```
<div style="border:thin solid orange">
```

In [1]:

```
<div class="input_area">
```

Out[1]:

```
</div>
```

In [ ]:

```
<div class="input_area">
```

```
</div>
```

CHAPTER 8

Customizing exporters

New in version 4.2: You can now use the `--to` flag to use custom export formats defined outside nbconvert.

The command-line syntax to run the `nbconvert` script is:

```
jupyter nbconvert --to FORMAT notebook.ipynb
```

This will convert the Jupyter document file `notebook.ipynb` into the output format designated by the `FORMAT` string as explained below.

Extending the built-in format exporters

A few built-in formats are available by default: *html*, *pdf*, *script*, *latex*. Each of these has its own *exporter* with many configuration options that can be extended. Having the option to point to a different *exporter* allows authors to create their own fully customized templates or export formats.

A custom *exporter* must be an importable Python object. We recommend that these be distributed as Python libraries.

Registering a custom exporter as an entry point

Additional exporters may be registered as named entry_points. nbconvert uses the `nbconvert.exporters` entry point to find exporters from any package you may have installed.

If you are writing a Python package that provides custom exporters, you can register the custom exporters in your package's `setup.py`. For example, your package may contain two custom exporters, named "simple" and "detail", and can be registered in your package's `setup.py` as follows:

```
setup(
    ...
    entry_points = {
        'nbconvert.exporters': [
            'simple = mymodule:SimpleExporter',
```

```
            'detail = mymodule:DetailExporter',
        ],
    }
)
```

Now people who have installed your Python package containing the two custom exporters can call the entry point name:

```
jupyter nbconvert --to detail mynotebook.ipynb
```

instead of having to specify the full import name of the custom exporter.

Using a custom exporter without entrypoints

We encourage registering custom exporters as entry points as described in the previous section. Registering a custom exporter with an entry point simplifies using the exporter. If a custom exporter has not been registered with an entry point, the exporter can still be used by providing the fully qualified name of this exporter as the argument of the `--to` flag when running from the command line:

```
$ jupyter nbconvert --to <full.qualified.name of custom exporter> notebook.ipynb
```

For example, assuming a library *tcontrib* has a custom exporter name *TExporter*, you would convert to this custom format using the following:

```
$ jupyter nbconvert --to tcontrib.TExporter notebook.ipynb
```

A library can contain multiple exporters. Creators of custom exporters should make sure that all other flags of the command line behave the same for the custom exporters as for built-in exporters.

CHAPTER 9

Parameters controlled by an external exporter

An external exporter can control almost any parameter of the notebook conversion process, from simple parameters such as the output file extension, to more complex ones such as the execution of the notebook or a custom rendering template.

All external exporters can expose custom options using the `traitlets` configurable API. Refer to the library that provides these exporters for details on how these configuration options works.

You can use the Jupyter configuration files to configure an external exporter. As for any `nbconvert` exporters you can use either the configuration file syntax of `c.MyExporter.config_option=value` or the command line flag form `--MyExporter.config_option=value`.

CHAPTER 10

Writing a custom `Exporter`

Under the hood exporters are python classes that expose a certain interface. Any importable classes that expose this interface can be use as an exporter for nbconvert.

For simplicity we expose basic classes that implement all the relevant methods that you have to subclass and overwrite just the relevant methods to provide a custom exporter. Below we show you the step to create a custom exporter that provides a custom file extension, and a custom template that inserts before and after each markdown cell.

We will lay out files to be ready for Python packaging and distributing on PyPI, although the exact art of Python packaging is beyond the scope of this explanation.

We will use the following layout for our package to expose a custom exporter:

```
mypackage
- LICENSE.md
- setup.py
- mypackage
    - __init__.py
    - templates
        - test_template.tpl
```

If you wished to create this same directory structure you could use the following commands when you are at the directory under which you wish to build your `mypackage` package:

```
mkdir -p mypackage/mypackage/templates
touch mypackage/LICENSE.md
touch mypackage/setup.py
touch mypackage/mypackage/__init__.py
touch mypackage/mypackage/templates/test_template.tpl
```

Important: You should not publish this package without adding content to your `LICENSE.md` file. For example, `nbconvert` follows the Jupyter Project convention of using a Modified BSD License (also known as New or Revised or 3-Clause BSD). For a guide on picking the right license for your use case, please see choose a license. If you do not specify the license, your code may be unusable by many open source projects.

As you can see the layout is relatively simple, in the case where a template is not needed we would actually have only one file with an Exporter implementation. Of course you can change the layout of your package to have a more fine-grained structure of the subpackage. But lets see what a minimum example looks like.

We are going to write an exporter that:

- exports to html, so we will reuse the built-in html exporter
- changes the file extension to *.test_ext*

```
# file __init__.py
import os
import os.path

from traitlets.config import Config
from nbconvert.exporters.html import HTMLExporter

#-----------------------------------------------------------------------------
# Classes
#-----------------------------------------------------------------------------

class MyExporter(HTMLExporter):
    """
    My custom exporter
    """

    def _file_extension_default(self):
        """
        The new file extension is `.test_ext`
        """
        return '.test_ext'

    @property
    def template_path(self):
        """
        We want to inherit from HTML template, and have template under
        `./templates/` so append it to the search path. (see next section)
        """
        return super().template_path+[os.path.join(os.path.dirname(__file__),
↪"templates")]

    def _template_file_default(self):
        """
        We want to use the new template we ship with our library.
        """
        return 'test_template' # full
```

And the template file, that inherits from the html *full* template and prepend/append text to each markdown cell (see Jinja2 docs for template syntax):

```
{% extends "full.tpl" %}

{% block markdowncell -%}


## this is a markdown cell
{super()}
## THIS IS THE END
```

```
{% endblock markdowncell %}
```

Assuming you install this package locally, or from PyPI, you can now use:

```
jupyter nbconvert --to mypackage.MyEporter notebook.ipynb
```

CHAPTER 11

Architecture of nbconvert

This is a high-level outline of the basic workflow, structures and objects in nbconvert. Specifically, this exposition has a two-fold goal:

1. to alert you to the affordances available for customisation or direct contributions
2. to provide a map of where and when different events occur, which should aid in tracking down bugs.

A detailed pipeline exploration

Nbconvert takes in a notebook, which is a JSON object, and operates on that object.

This can include operations that take a notebook and return a notebook. For example, that operation could be to execute the notebook as though it were a continuous script; if it were executed `--in-place` then it would overwite the current notebook. Or it could be that we wish to systematically alter the notebook, for example by clearing all output cells. Format agnostic operations on cell content that do not violate the nbformat spec can be interpreted as a notebook to notebook conversion step; such operations can be performed as part of the preprocessing step.

But often we want to have the notebook's structured content in a different format. Importantly, in many cases the structure of the notebook should be reflected in the structure of the output, adapted to the output's format. For that purpose, the original JSON structure of the document is crucial scaffolding needed to support this kind of structured output. In order to maintain structured, it can be useful to apply our conversion programmatically on the structure itself. To do so, when converting to formats other than the notebook, we use the jinja templating engine.

The basic unit of structure in a notebook is the cell. Accordingly, since our templating engine is capable of expressing structure, the basic unit in our templates will often be specified at the cell level. Each cell has a certain type; the three most important cell types for our purposes are code, markdown, and raw NbConvert. Code cells can be split further into their input and their output. Operations can also occur separately on input and output and their respective subcomponents. Markdown cells and raw NbConvert cells do not have analogous substructure.

The template's structure then can be seen as a mechanism for selecting content on which to operate. Because the template operates on individual cells, this has some upsides and drawbacks. One upside is that this allows the template to have access to the individual cell's metadata, which enables intelligently transforming the appropriate content. The transformations occur as a series of replacement rules and filters. For many purposes these filters take the form of

external calls to pandoc, which is a utility for converting between many different document formats. One downside is that this makes operations that require global coördination (e.g., cross referencing across cells) somewhat challenging to implement as filters inside templates.

Note that all that we've described is happening in memory. This is crucial in order to ensure that this functionality is available when writing files is more challenging. Nonetheless, the reason for using nbconvert almost always involves producing some kind of output file. We take the in-memory object and write a file appropriate for the output type.

The entirety of heretofore described process can be described as part of an `Exporter`. `Exporters` often involves `Preprocessors`, `filters`, `templates` and `Writers`. These classes and functions are described in greater detail below.

Finally, one can apply a `Postprocessor` after the writing has occurred. For example, it is common when converting to slides to start a webserver and open a browser window with the newly created document (`--to slides --post serve`).

Classes

Exporters

The primary class in nbconvert is the *`Exporter`*. Exporters encapsulate the operation of turning a notebook into another format. There is one Exporter for each format supported in nbconvert. The first thing an Exporter does is load a notebook, usually from a file via `nbformat`. Most of what a typical Exporter does is select and configure preprocessors, filters, and templates. If you want to convert notebooks to additional formats, a new Exporter is probably what you are looking for.

See also:

Writing a custom Exporter

Once the notebook is loaded, it is preprocessed...

Preprocessors

A *`Preprocessor`* is an object that transforms the content of the notebook to be exported. The result of a preprocessor being applied to a notebook is always a notebook. These operations include re-executing the cells, stripping output, removing bundled outputs to separate files, etc. If you want to add operations that modify a notebook before exporting, a preprocessor is the place to start.

See also:

Custom Preprocessors

Once a notebook is preprocessed, it's time to convert the notebook into the destination format.

Templates and Filters

Most Exporters in nbconvert are a subclass of *`TemplateExporter`*, which means they use a jinja template to render a notebook into the destination format. If you want to change how an exported notebook looks in an existing format, a custom template is the place to start.

A jinja template is composed of blocks that look like this (taken from nbconvert's default html template):

```
{% block stream_stdout -%}
<div class="output_subarea output_stream output_stdout output_text">
<pre>
{{- output.text | ansi2html -}}
</pre>
</div>
{%- endblock stream_stdout %}
```

This block determines how text output on `stdout` is displayed in HTML. The `{{- output.text | ansi2html -}}` bit means "Take the output text and pass it through ansi2html, then include the result here." In this example, `ansi2html` is a filter. Filters are a jinja concept; they are Python callables which take something (typically text) as an input, and produce a text output. If you want to perform new or more complex transformations of particular outputs, a filter may be what you need. Typically, filters are pure functions. However, if you have a filter that itself requires some configuration, it can be an instance of a callable, configurable class.

See also:

- *Customizing nbconvert*
- Filters

Once it has passed through the template, an Exporter is done with the notebook, and returns the file data. At this point, we have the file data as text or bytes and we can decide where it should end up. When you are using nbconvert as a library, as opposed to the command-line application, this is typically where you would stop, take your exported data, and go on your way.

Writers

A `Writer` takes care of writing the resulting file(s) where they should end up. There are two basic Writers in nbconvert:

1. stdout - writes the result to stdout (for pipe-style workflows)
2. Files (default) - writes the result to the filesystem

Once the output is written, nbconvert has done its job.

Postprocessors

A `Postprocessor` is something that runs after everything is exported and written to the filesystem. The only postprocessor in nbconvert at this point is the *`ServePostProcessor`*, which is used for serving reveal.js HTML slideshows.

CHAPTER 12

Python API for working with nbconvert

Contents:

NbConvertApp

See also:

Configuration options Configurable options for the nbconvert application

class nbconvert.nbconvertapp.**NbConvertApp**(***kwargs*)

Application used to convert from notebook file type (`*.ipynb`)

init_notebooks()

Construct the list of notebooks.

If notebooks are passed on the command-line, they override (rather than add) notebooks specified in config files. Glob each notebook to replace notebook patterns with filenames.

convert_notebooks()

Convert the notebooks in the self.notebook traitlet

convert_single_notebook(*notebook_filename*, *input_buffer=None*)

Convert a single notebook.

Performs the following steps:

1. Initialize notebook resources
2. Export the notebook to a particular format
3. Write the exported notebook to file
4. (Maybe) postprocess the written file

Parameters

- **notebook_filename** (*str*) –

- **input_buffer** – If input_buffer is not None, conversion is done and the buffer is used as source into a file basenamed by the notebook_filename argument.

init_single_notebook_resources(*notebook_filename*)
Step 1: Initialize resources

This initializes the resources dictionary for a single notebook.

Returns

resources dictionary for a single notebook that MUST include the following keys:

- config_dir: the location of the Jupyter config directory
- unique_key: the notebook name
- output_files_dir: a directory where output files (not including the notebook itself) should be saved

Return type dict

export_single_notebook(*notebook_filename*, *resources*, *input_buffer=None*)
Step 2: Export the notebook

Exports the notebook to a particular format according to the specified exporter. This function returns the output and (possibly modified) resources from the exporter.

Parameters

- **notebook_filename** (*str*) – name of notebook file.
- **resources** (*dict*) –
- **input_buffer** – readable file-like object returning unicode. if not None, notebook_filename is ignored

Returns

- *output*
- *dict* – resources (possibly modified)

write_single_notebook(*output*, *resources*)
Step 3: Write the notebook to file

This writes output from the exporter to file using the specified writer. It returns the results from the writer.

Parameters

- **output** –
- **resources** (*dict*) – resources for a single notebook including name, config directory and directory to save output

Returns results from the specified writer output of exporter

Return type file

postprocess_single_notebook(*write_results*)
Step 4: Post-process the written file

Only used if a postprocessor has been specified. After the converted notebook is written to a file in Step 3, this post-processes the notebook.

Exporters

See also:

Configuration options Configurable options for the nbconvert application

nbconvert.exporters.**export**(*exporter*, *nb*, ***kw*)
Export a notebook object using specific exporter class.

Parameters

- **exporter** (Exporter class or instance) – Class or instance of the exporter that should be used. If the method initializes its own instance of the class, it is ASSUMED that the class type provided exposes a constructor (__init__) with the same signature as the base Exporter class.
- **nb** (NotebookNode) – The notebook to export.
- **config** (*config (optional, keyword arg)*) – User configuration instance.
- **resources** (*dict (optional, keyword arg)*) – Resources used in the conversion process.

Returns

output [str] The resulting converted notebook.

resources [dictionary] Dictionary of resources used prior to and during the conversion process.

Return type tuple

nbconvert.exporters.**get_exporter**(*name*)
Given an exporter name or import path, return a class ready to be instantiated

Raises ValueError if exporter is not found

nbconvert.exporters.**get_export_names**()
Return a list of the currently supported export targets

Exporters can be found in external packages by registering them as an nbconvert.exporter entrypoint.

Exporter base classes

class nbconvert.exporters.**Exporter**(*config=None*, ***kw*)
Class containing methods that sequentially run a list of preprocessors on a NotebookNode object and then return the modified NotebookNode object and accompanying resources dict.

__init__(*config=None*, ***kw*)
Public constructor

Parameters

- **config** (Config) – User configuration instance.
- ****kw** – Additional keyword arguments passed to parent __init__

from_notebook_node(*nb*, *resources=None*, ***kw*)
Convert a notebook from a notebook node instance.

Parameters

- **nb** (NotebookNode) – Notebook node (dict-like with attr-access)

- **resources** (*dict*) – Additional resources that can be accessed read/write by preprocessors and filters.
- ****kw** – Ignored

from_filename(*filename*, *resources=None*, ***kw*)
Convert a notebook from a notebook file.

Parameters

- **filename** (*str*) – Full filename of the notebook file to open and convert.
- **resources** (*dict*) – Additional resources that can be accessed read/write by preprocessors and filters.
- ****kw** – Ignored

from_file(*file_stream*, *resources=None*, ***kw*)
Convert a notebook from a notebook file.

Parameters

- **file_stream** (*file-like object*) – Notebook file-like object to convert.
- **resources** (*dict*) – Additional resources that can be accessed read/write by preprocessors and filters.
- ****kw** – Ignored

register_preprocessor(*preprocessor*, *enabled=False*)
Register a preprocessor. Preprocessors are classes that act upon the notebook before it is passed into the Jinja templating engine. preprocessors are also capable of passing additional information to the Jinja templating engine.

Parameters

- **preprocessor** (*Preprocessor*) – A dotted module name, a type, or an instance
- **enabled** (*bool*) – Mark the preprocessor as enabled

class nbconvert.exporters.**TemplateExporter**(*config=None*, ***kw*)
Exports notebooks into other file formats. Uses Jinja 2 templating engine to output new formats. Inherit from this class if you are creating a new template type along with new filters/preprocessors. If the filters/ preprocessors provided by default suffice, there is no need to inherit from this class. Instead, override the template_file and file_extension traits via a config file.

Filters available by default for templates:

•add_anchor

•add_prompts

•ansi2html

•ansi2latex

•ascii_only

•citation2latex

•comment_lines

•convert_pandoc

•escape_latex

•filter_data_type

- get_lines
- get_metadata
- highlight2html
- highlight2latex
- html2text
- indent
- ipython2python
- json_dumps
- markdown2asciidoc
- markdown2html
- markdown2latex
- markdown2rst
- path2url
- posix_path
- prevent_list_blocks
- strip_ansi
- strip_dollars
- strip_files_prefix
- wrap_text

__init__(*config=None*, ***kw*)
Public constructor

Parameters

- **config** (*config*) – User configuration instance.
- **extra_loaders** (*list[of Jinja Loaders]*) – ordered list of Jinja loader to find templates. Will be tried in order before the default FileSystem ones.
- **template** (*str (optional, kw arg)*) – Template to use when exporting.

from_notebook_node(*nb*, *resources=None*, ***kw*)
Convert a notebook from a notebook node instance.

Parameters

- **nb** (NotebookNode) – Notebook node
- **resources** (*dict*) – Additional resources that can be accessed read/write by preprocessors and filters.

from_filename(*filename*, *resources=None*, ***kw*)
Convert a notebook from a notebook file.

Parameters

- **filename** (*str*) – Full filename of the notebook file to open and convert.
- **resources** (*dict*) – Additional resources that can be accessed read/write by preprocessors and filters.

- ****kw** – Ignored

from_file (*file_stream*, *resources=None*, ***kw*)
Convert a notebook from a notebook file.

Parameters

- **file_stream** (*file-like object*) – Notebook file-like object to convert.
- **resources** (*dict*) – Additional resources that can be accessed read/write by preprocessors and filters.
- ****kw** – Ignored

register_preprocessor (*preprocessor*, *enabled=False*)
Register a preprocessor. Preprocessors are classes that act upon the notebook before it is passed into the Jinja templating engine. preprocessors are also capable of passing additional information to the Jinja templating engine.

Parameters

- **preprocessor** (*Preprocessor*) – A dotted module name, a type, or an instance
- **enabled** (*bool*) – Mark the preprocessor as enabled

register_filter (*name*, *jinja_filter*)
Register a filter. A filter is a function that accepts and acts on one string. The filters are accessible within the Jinja templating engine.

Parameters

- **name** (*str*) – name to give the filter in the Jinja engine
- **filter** (*filter*) –

Specialized exporter classes

The *NotebookExporter* inherits directly from *Exporter*, while the other exporters listed here inherit either directly or indirectly from *TemplateExporter*.

class nbconvert.exporters.**NotebookExporter** (*config=None*, ***kw*)
Exports to an IPython notebook.

This is useful when you want to use nbconvert's preprocessors to operate on a notebook (e.g. to execute it) and then write it back to a notebook file.

class nbconvert.exporters.**HTMLExporter** (*config=None*, ***kw*)
Exports a basic HTML document. This exporter assists with the export of HTML. Inherit from it if you are writing your own HTML template and need custom preprocessors/filters. If you don't need custom preprocessors/filters, just change the 'template_file' config option.

class nbconvert.exporters.**SlidesExporter** (*config=None*, ***kw*)
Exports HTML slides with reveal.js

class nbconvert.exporters.**LatexExporter** (*config=None*, ***kw*)
Exports to a Latex template. Inherit from this class if your template is LaTeX based and you need custom tranformers/filters. Inherit from it if you are writing your own HTML template and need custom tranformers/filters. If you don't need custom tranformers/filters, just change the 'template_file' config option. Place your template in the special "/latex" subfolder of the "../templates" folder.

class nbconvert.exporters.**MarkdownExporter** (*config=None*, ***kw*)
Exports to a markdown document (.md)

class nbconvert.exporters.**PDFExporter**(*config=None*, ***kw*)
Writer designed to write to PDF files.

This inherits from *LatexExporter*. It creates a LaTeX file in a temporary directory using the template machinery, and then runs LaTeX to create a pdf.

class nbconvert.exporters.**PythonExporter**(*config=None*, ***kw*)
Exports a Python code file.

class nbconvert.exporters.**RSTExporter**(*config=None*, ***kw*)
Exports reStructuredText documents.

Preprocessors

See also:

Configuration options Configurable options for the nbconvert application

class nbconvert.preprocessors.**Preprocessor**(***kw*)
A configurable preprocessor

Inherit from this class if you wish to have configurability for your preprocessor.

Any configurable traitlets this class exposed will be configurable in profiles using c.SubClassName.attribute = value

you can overwrite *preprocess_cell()* to apply a transformation independently on each cell or *preprocess()* if you prefer your own logic. See corresponding docstring for informations.

Disabled by default and can be enabled via the config by 'c.YourPreprocessorName.enabled = True'

__init__(***kw*)
Public constructor

Parameters

- **config** (*Config*) – Configuration file structure
- ****kw** – Additional keyword arguments passed to parent

preprocess(*nb*, *resources*)
Preprocessing to apply on each notebook.

Must return modified nb, resources.

If you wish to apply your preprocessing to each cell, you might want to override preprocess_cell method instead.

Parameters

- **nb** (*NotebookNode*) – Notebook being converted
- **resources** (*dictionary*) – Additional resources used in the conversion process. Allows preprocessors to pass variables into the Jinja engine.

preprocess_cell(*cell*, *resources*, *index*)
Override if you want to apply some preprocessing to each cell. Must return modified cell and resource dictionary.

Parameters

- **cell** (*NotebookNode cell*) – Notebook cell being processed

- **resources** (*dictionary*) – Additional resources used in the conversion process. Allows preprocessors to pass variables into the Jinja engine.
- **index** (*int*) – Index of the cell being processed

Specialized preprocessors

class nbconvert.preprocessors.**ConvertFiguresPreprocessor**(***kw*)
Converts all of the outputs in a notebook from one format to another.

class nbconvert.preprocessors.**SVG2PDFPreprocessor**(***kw*)
Converts all of the outputs in a notebook from SVG to PDF.

class nbconvert.preprocessors.**ExtractOutputPreprocessor**(***kw*)
Extracts all of the outputs from the notebook file. The extracted outputs are returned in the 'resources' dictionary.

class nbconvert.preprocessors.**LatexPreprocessor**(***kw*)
Preprocessor for latex destined documents.

Mainly populates the *latex* key in the resources dict, adding definitions for pygments highlight styles.

class nbconvert.preprocessors.**CSSHTMLHeaderPreprocessor**(**pargs*, ***kwargs*)
Preprocessor used to pre-process notebook for HTML output. Adds IPython notebook front-end CSS and Pygments CSS to HTML output.

class nbconvert.preprocessors.**HighlightMagicsPreprocessor**(*config=None*, ***kw*)
Detects and tags code cells that use a different languages than Python.

class nbconvert.preprocessors.**ClearOutputPreprocessor**(***kw*)
Removes the output from all code cells in a notebook.

class nbconvert.preprocessors.**ExecutePreprocessor**(***kw*)
Executes all the cells in a notebook

preprocess(*nb*, *resources*)
Preprocess notebook executing each code cell.

The input argument *nb* is modified in-place.

Parameters

- **nb** (*NotebookNode*) – Notebook being executed.
- **resources** (*dictionary*) – Additional resources used in the conversion process. For example, passing `{'metadata': {'path': run_path}}` sets the execution path to `run_path`.

Returns

- **nb** (*NotebookNode*) – The executed notebook.
- **resources** (*dictionary*) – Additional resources used in the conversion process.

preprocess_cell(*cell*, *resources*, *cell_index*)
Executes a single code cell. See base.py for details.

To execute all cells see *preprocess()*.

nbconvert.preprocessors.**coalesce_streams**(*cell*, *resources*, *index*)
Merge consecutive sequences of stream output into single stream to prevent extra newlines inserted at flush calls

Parameters

- **cell** (*NotebookNode cell*) – Notebook cell being processed

- **resources** (*dictionary*) – Additional resources used in the conversion process. Allows transformers to pass variables into the Jinja engine.
- **index** (*int*) – Index of the cell being processed

Filters

Filters are for use with the *TemplateExporter* exporter. They provide a way for you transform notebook contents to a particular format depending on the template you are using. For example, when converting to HTML, you would want to use the *ansi2html()* function to convert ANSI colors (from e.g. a terminal traceback) to HTML colors.

See also:

Exporters API documentation for the various exporter classes

nbconvert.filters.**add_anchor**(*html*)
Add an id and an anchor-link to an html header

For use on markdown headings

nbconvert.filters.**add_prompts**(*code*, *first='>>> '*, *cont='... '*)
Add prompts to code snippets

nbconvert.filters.**ansi2html**(*text*)
Convert ANSI colors to HTML colors.

Parameters **text** (*unicode*) – Text containing ANSI colors to convert to HTML

nbconvert.filters.**ansi2latex**(*text*)
Convert ANSI colors to LaTeX colors.

Parameters **text** (*unicode*) – Text containing ANSI colors to convert to LaTeX

nbconvert.filters.**ascii_only**(*s*)
ensure a string is ascii

nbconvert.filters.**citation2latex**(*s*)
Parse citations in Markdown cells.

This looks for HTML tags having a data attribute names *data-cite* and replaces it by the call to LaTeX cite command. The tranformation looks like this:

<cite data-cite="granger">(Granger, 2013)</cite>

Becomes

cite{granger}

Any HTML tag can be used, which allows the citations to be formatted in HTML in any manner.

nbconvert.filters.**comment_lines**(*text*, *prefix='# '*)
Build a Python comment line from input text.

Parameters

- **text** (*str*) – Text to comment out.
- **prefix** (*str*) – Character to append to the start of each line.

nbconvert.filters.**escape_latex**(*text*)
Escape characters that may conflict with latex.

Parameters **text** (*str*) – Text containing characters that may conflict with Latex

class nbconvert.filters.**DataTypeFilter**(***kw*)
Returns the preferred display format

nbconvert.filters.**get_lines**(*text*, *start=None*, *end=None*)
Split the input text into separate lines and then return the lines that the caller is interested in.

Parameters

- **text** (*str*) – Text to parse lines from.
- **start** (*int, optional*) – First line to grab from.
- **end** (*int, optional*) – Last line to grab from.

nbconvert.filters.**convert_pandoc**(*source*, *from_format*, *to_format*, *extra_args=None*)
Convert between any two formats using pandoc.

This function will raise an error if pandoc is not installed. Any error messages generated by pandoc are printed to stderr.

Parameters

- **source** (*string*) – Input string, assumed to be valid in from_format.
- **from_format** (*string*) – Pandoc format of source.
- **to_format** (*string*) – Pandoc format for output.

Returns **out** – Output as returned by pandoc.

Return type string

class nbconvert.filters.**Highlight2HTML**(*pygments_lexer=None*, ***kwargs*)

class nbconvert.filters.**Highlight2Latex**(*pygments_lexer=None*, ***kwargs*)

nbconvert.filters.**html2text**(*element*)
extract inner text from html

Analog of jQuery's $(element).text()

nbconvert.filters.**indent**(*instr*, *nspaces=4*, *ntabs=0*, *flatten=False*)
Indent a string a given number of spaces or tabstops.

indent(str,nspaces=4,ntabs=0) -> indent str by ntabs+nspaces.

Parameters

- **instr** (*basestring*) – The string to be indented.
- **nspaces** (*int (default: 4)*) – The number of spaces to be indented.
- **ntabs** (*int (default: 0)*) – The number of tabs to be indented.
- **flatten** (*bool (default: False)*) – Whether to scrub existing indentation. If True, all lines will be aligned to the same indentation. If False, existing indentation will be strictly increased.

Returns **str|unicode**

Return type string indented by ntabs and nspaces.

nbconvert.filters.**ipython2python**(*code*)
Transform IPython syntax to pure Python syntax

Parameters **code** (*str*) – IPython code, to be transformed to pure Python

nbconvert.filters.**markdown2html**(*source*)
Convert a markdown string to HTML using mistune

nbconvert.filters.**markdown2latex**(*source*, *markup='markdown'*, *extra_args=None*)
Convert a markdown string to LaTeX via pandoc.

This function will raise an error if pandoc is not installed. Any error messages generated by pandoc are printed to stderr.

Parameters

- **source** (*string*) – Input string, assumed to be valid markdown.
- **markup** (*string*) – Markup used by pandoc's reader default : pandoc extended markdown (see http://pandoc.org/README.html#pandocs-markdown)

Returns **out** – Output as returned by pandoc.

Return type string

nbconvert.filters.**markdown2rst**(*source*, *extra_args=None*)
Convert a markdown string to ReST via pandoc.

This function will raise an error if pandoc is not installed. Any error messages generated by pandoc are printed to stderr.

Parameters **source** (*string*) – Input string, assumed to be valid markdown.

Returns **out** – Output as returned by pandoc.

Return type string

nbconvert.filters.**path2url**(*path*)
Turn a file path into a URL

nbconvert.filters.**posix_path**(*path*)
Turn a path into posix-style path/to/etc

Mainly for use in latex on Windows, where native Windows paths are not allowed.

nbconvert.filters.**prevent_list_blocks**(*s*)
Prevent presence of enumerate or itemize blocks in latex headings cells

nbconvert.filters.**strip_ansi**(*source*)
Remove ANSI escape codes from text.

Parameters **source** (*str*) – Source to remove the ANSI from

nbconvert.filters.**strip_dollars**(*text*)
Remove all dollar symbols from text

Parameters **text** (*str*) – Text to remove dollars from

nbconvert.filters.**strip_files_prefix**(*text*)
Fix all fake URLs that start with *files/*, stripping out the *files/* prefix. Applies to both urls (for html) and relative paths (for markdown paths).

Parameters **text** (*str*) – Text in which to replace 'src="files/real...' with 'src="real...'

nbconvert.filters.**wrap_text**(*text*, *width=100*)
Intelligently wrap text. Wrap text without breaking words if possible.

Parameters

- **text** (*str*) – Text to wrap.
- **width** (*int, optional*) – Number of characters to wrap to, default 100.

Writers

See also:

Configuration options Configurable options for the nbconvert application

class nbconvert.writers.**WriterBase**(*config=None*, ***kw*)
Consumes output from nbconvert export...() methods and writes to a useful location.

__init__(*config=None*, ***kw*)
Constructor

write(*output*, *resources*, ***kw*)
Consume and write Jinja output.

Parameters

- **output** (*string*) – Conversion results. This string contains the file contents of the converted file.
- **resources** (*dict*) – Resources created and filled by the nbconvert conversion process. Includes output from preprocessors, such as the extract figure preprocessor.

Specialized writers

class nbconvert.writers.**DebugWriter**(*config=None*, ***kw*)
Consumes output from nbconvert export...() methods and writes usefull debugging information to the stdout. The information includes a list of resources that were extracted from the notebook(s) during export.

class nbconvert.writers.**FilesWriter**(***kw*)
Consumes nbconvert output and produces files.

class nbconvert.writers.**StdoutWriter**(*config=None*, ***kw*)
Consumes output from nbconvert export...() methods and writes to the stdout stream.

Postprocessors

See also:

Configuration options Configurable options for the nbconvert application

class nbconvert.postprocessors.**PostProcessorBase**(***kw*)

postprocess(*input*)
Post-process output from a writer.

Specialized postprocessors

class nbconvert.postprocessors.**ServePostProcessor**(***kw*)
Post processor designed to serve files

Proxies reveal.js requests to a CDN if no local reveal.js is present

postprocess(*input*)
Serve the build directory with a webserver.

CHAPTER 13

Changes in nbconvert

5.1.1

5.1.1 on GitHub

- fix version numbering because of incomplete previous version number

5.1

5.1 on GitHub

- improved CSS (specifically tables, in line with notebook) #498
- improve in-memory templates handling #491
- test improvements #516 #509 #505
- new configuration option: IOPub timeout #513
- doc improvements #489 #500 #493 #506
- newly customizable: output prompt #500
- more python2/3 compatibile unicode handling #502

5.0

5.0 on GitHub

- Use **`xelatex`** by default for latex export, improving unicode and font support.
- Use entrypoints internally to access Exporters, allowing for packages to declare custom exporters more easily.
- New ASCIIDoc Exporter.

- New preprocessor for sanitised html output.
- New general `convert_pandoc` filter to reduce the need to hard-code lists of filters in templates.
- Use pytest, nose dependency to be removed.
- Refactored Exporter code to avoid ambiguity and cyclic dependencies.
- Update to traitlets 4.2 API.
- Fixes for Unicode errors when showing execution errors on Python 2.
- Default math font matches default Palatino body text font.
- General documentation improvements. For example, testing, installation, custom exporters.
- Improved link handling for LaTeX output
- Refactored the automatic id generation.
- New kernel_manager_class configuration option for allowing systems to be set up to resolve kernels in different ways.
- Kernel errors now will be logged for debugging purposes when executing notebooks.

4.3

4.3 on GitHub

- added live widget rendering for html output, nbviewer by extension

4.2

4.2 on GitHub

- *Custom Exporters* can be provided by external packages, and registered with nbconvert via setuptools entry-points.
- allow nbconvert reading from stdin with `--stdin` option (write into `notebook` basename)
- Various ANSI-escape fixes and improvements
- Various LaTeX/PDF export fixes
- Various fixes and improvements for executing notebooks with `--execute`.

4.1

4.1 on GitHub

- setuptools fixes for entrypoints on Windows
- various fixes for exporters, including slides, latex, and PDF
- fixes for exceptions met during execution
- include markdown outputs in markdown/html exports

4.0

4.0 on GitHub

CHAPTER 14

Indices and tables

- genindex
- modindex
- search

Python Module Index

n

Index

Symbols

A

C

D

E

F

G

H

I