Jupytext

May 24, 2020

Contents

1 Contents 3

- jupy
+ text

CHAPTER 1

Contents

1.1 Introduction

Have you always wished Jupyter notebooks were plain text documents? Wished you could edit them in your favorite IDE? And get clear and meaningful diffs when doing version control? Then... Jupytext may well be the tool you're looking for!

Jupytext can save Jupyter notebooks as

- Markdown and R Markdown documents,
- Scripts in many languages.

The languages that are currently supported by Jupytext are: Julia, Python, R, Bash, Scheme, Clojure, Matlab, Octave, C++, q/kdb+, IDL, TypeScript, Javascript, Scala, Rust/Evxcr, PowerShell, C#, F#, Robot Framework, Script of Script. Extending Jupytext to more languages should be easy - read more at CONTRIBUTING.md. In addition, jupytext users can choose between two formats for notebooks as scripts:

- The `percent` format, compatible with several IDEs, including Spyder, Hydrogen, VScode and PyCharm. In that format, cells are delimited with a commented `%%`.
- The `light` format, designed for this project. Use that format to open standard scripts as notebooks, or to save notebooks as scripts with few cell markers - none when possible.

Jupytext can also convert these formats **into Jupyter Notebooks**, allowing for two-directional syncing between formats. See below for a quick demo.

1.1.1 How to use Jupytext

There are multiple ways to use `jupytext`:

- **Directly from Jupyter Notebook or JupyterLab.** Jupytext provides a *contents manager* that allows Jupyter to save your notebook to your favorite format (`.py`, `.R`, `.jl`, `.md`, `.Rmd`...) in addition to (or in place of) the traditional `.ipynb` file. The text representation can be edited in your favorite editor. When you're done, refresh the notebook in Jupyter: inputs cells are loaded from the text file, while output cells are reloaded from the `.ipynb` file if present. Refreshing preserves kernel variables, so you can resume your work in the notebook and run the modified cells without having to rerun the notebook in full.
- **On the *command line***. `jupytext` converts Jupyter notebooks to their text representation, and back. The command line tool can act on notebooks in many ways. It can synchronize multiple representations of a notebook, pipe a notebook into a reformatting tool like `black`, etc... It can also work as a pre-commit hook if you wish to automatically update the text representation when you commit the `.ipynb` file.
- **in Vim**: edit your Jupyter notebooks, represented as a Markdown document, or a Python script, with jupytext.vim.

1.1.2 Jupytext formats

Jupytext implements a series of text formats for notebooks, which are documented *here*.

In short: the Markdown representation of notebooks fits well the notebooks that contain narratives, while notebooks that mostly contain code are conveniently saved as scripts. The most popular formats for notebooks as scripts are:

- the `percent` format (in which cells are delimited with `# %%`) also used by Spyder, VSCode, PyCharm, and others,
- and the `light` format which was developed to support this project. `light` uses as few cell markers as possible and is particularly suited for importing a pre-existing python script as a notebook with cell divisions automatically inferred from paragraph breaks in the source code.

1.1.3 Demo time

Looking for a demo?

- Read the original announcement in *Towards Data Science* (Sept. 2018),
- Watch the PyParis talk (Nov. 2018),
- Read our article on Jupytext and Papermill in *CFM Insights* (Sept. 2019)
- See how you can edit Jupyter Notebooks in VS Code or PyCharm with (or without!) Jupytext (Jan. 2020)
- or, try Jupytext online with binder!

1.1.4 Want to contribute?

Contributions are welcome. Please let us know how you use `jupytext` and how we could improve it. You think the documentation could be improved? Go ahead! And stay tuned for more demos on medium and twitter!

1.2 Usecases for Jupytext

1.2.1 Writing notebooks as plain text

You like to work with scripts? The good news is that plain scripts, which you can draft and test in your favorite IDE, open transparently as notebooks in Jupyter when using Jupytext. Run the notebook in Jupyter to generate the outputs, associate an `.ipynb` representation, save and share your research as either a plain script or as a traditional Jupyter notebook with outputs.

1.2.2 Collaborating on Jupyter Notebooks

With Jupytext, collaborating on Jupyter notebooks with Git becomes as easy as collaborating on text files.

The setup is straightforward:

- Open your favorite notebook in Jupyter notebook
- Associate a `.py` representation (for instance) to that notebook
- Save the notebook, and put the Python script under Git control. Sharing the `.ipynb` file is possible, but not required.

Collaborating then works as follows:

- Your collaborator pulls your script.
- The script opens as a notebook in Jupyter, with no outputs (in JupyterLab right-click the script and use the open-with context menu).
- They run the notebook and save it. Outputs are regenerated, and a local `.ipynb` file is created.
- Note that, alternatively, the `.ipynb` file could have been regenerated with `jupytext --sync notebook.py`.
- They change the notebook, and push their updated script. The diff is nothing else than a standard diff on a Python script.
- You pull the changed script, and refresh your browser. Input cells are updated. The outputs from cells that were changed are removed. Your variables are untouched, so you have the option to run only the modified cells to get the new outputs.

1.2.3 Code refactoring

In the animation below we propose a quick demo of Jupytext. While the example remains simple, it shows how your favorite text editor or IDE can be used to edit your Jupyter notebooks. IDEs are more convenient than Jupyter for navigating through code, editing and executing cells or fractions of cells, and debugging.

- We start with a Jupyter notebook.
- The notebook includes a plot of the world population. The plot legend is not in order of decreasing population, we'll fix this.
- We want the notebook to be saved as both a `.ipynb` and a `.py` file: we select *Pair Notebook with a light Script* in either the Jupytext menu in Jupyter Notebook, or in the Jupytext commands in JupyterLab. This has the effect of adding a `"jupytext": {"formats": "ipynb,py:light"},` entry to the notebook metadata.
- The Python script can be opened with PyCharm:
  - Navigating in the code and documentation is easier than in Jupyter.

  - The console is convenient for quick tests. We don't need to create cells for this.
  - We find out that the columns of the data frame were not in the correct order. We update the corresponding cell, and get the correct plot.
- The Jupyter notebook is refreshed in the browser. Modified inputs are loaded from the Python script. Outputs and variables are preserved. We finally rerun the code and get the correct plot.

1.2.4 Importing Jupyter Notebooks as modules

Jupytext allows to import code from other Jupyter notebooks in a very simple manner. Indeed, all you need to do is to pair the notebook that you wish to import with a script, and import the resulting script.

If the notebook contains demos and plots that you don't want to import, mark those cells as either

- *active* only in the `ipynb` format, with the `{"active": "ipynb"}` cell metadata, or with an `active-ipynb` tag (you may use the jupyterlab-celltags extension for this). Use a `{"active": "ipynb,py"}` metadata or a an `active-ipynb-py` tag if you want the cell to be active only in the `ipynb` and `py` formats, but not in the R Markdown format.
- *frozen*, using the freeze extension for Jupyter notebook.

1.3 Installation

Jupytext is available on pypi and on conda-forge. Run either of

```
pip install jupytext --upgrade
```

or

```
conda install -c conda-forge jupytext
```

If you want to use Jupytext within Jupyter Notebook or JupyterLab, make sure you install Jupytext in the Python environment where the Jupyter server runs. If that environment is read-only, for instance if your server is started using JupyterHub, install Jupytext in user mode with:

```
/path_to_your_jupyter_environment/python -m pip install jupytext --upgrade --user
```

1.3.1 Jupytext's contents manager

Jupytext provides a contents manager for Jupyter that allows Jupyter to open and save notebooks as text files. When Jupytext's content manager is active in Jupyter, scripts and Markdown documents have a notebook icon.

In most cases, Jupytext's contents manager is activated automatically by Jupytext's server extension. When you restart either `jupyter lab` or `jupyter notebook`, you should see a line that looks like:

```
[I 10:28:31.646 LabApp] [Jupytext Server Extension] Changing NotebookApp.contents_
↪manager_class from LargeFileManager to jupytext.TextFileContentsManager
```

If you don't have the notebook icon on text documents after a fresh restart of your Jupyter server, you can either enable our server extension explicitly (with `jupyter serverextension enable jupytext`), or install the contents manager manually. Append

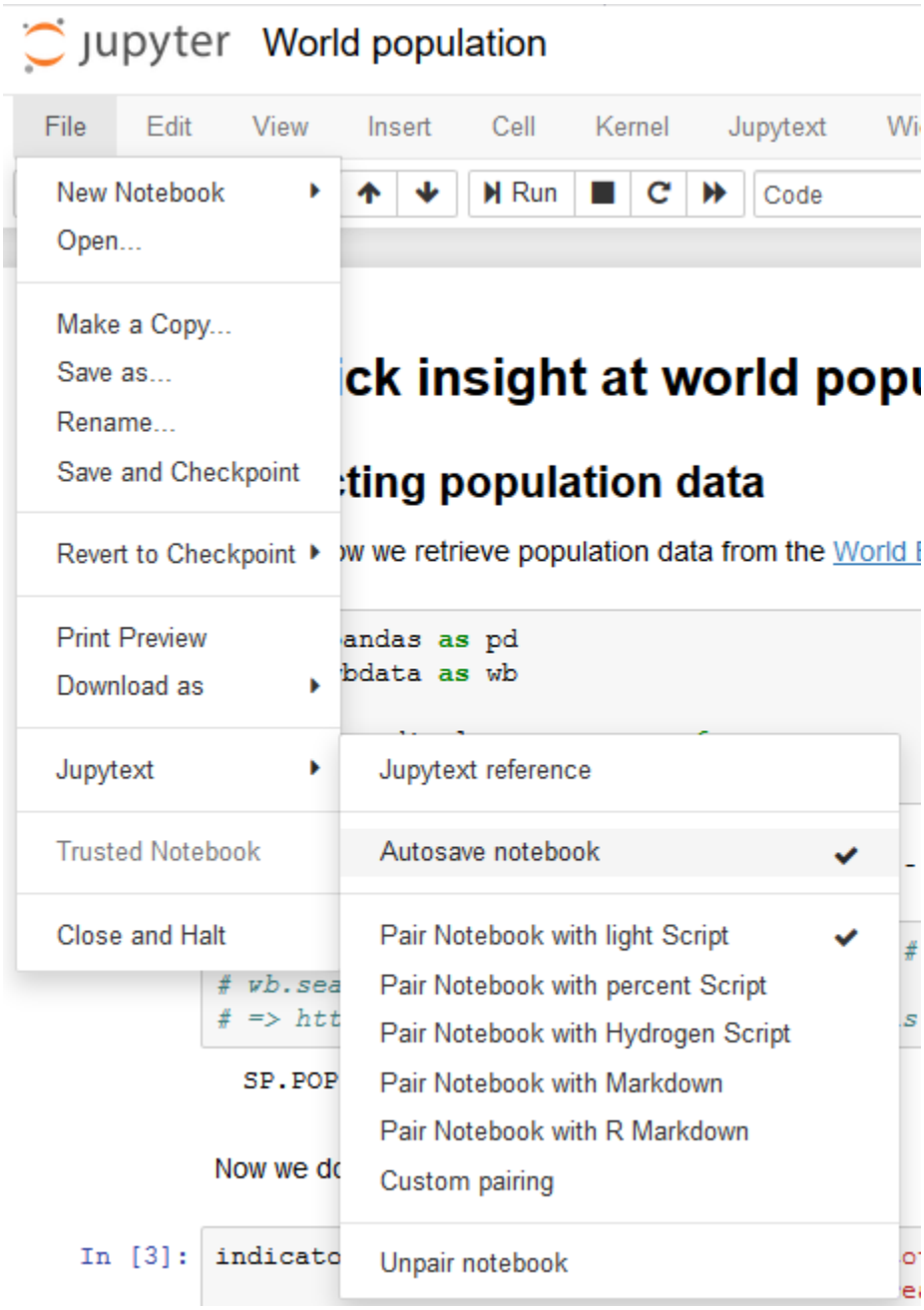
```
c.NotebookApp.contents_manager_class = "jupytext.TextFileContentsManager"
```

to your `.jupyter/jupyter_notebook_config.py` file (generate a Jupyter config, if you don't have one yet, with `jupyter notebook --generate-config`). Our contents manager accepts a few options: default formats, default metadata filter, etc. Then, restart Jupyter Notebook or JupyterLab, either from the JupyterHub interface or from the command line with

```
jupyter notebook # or lab
```

1.3.2 Jupytext menu in Jupyter Notebook

Jupytext includes an extensions for Jupyter Notebook that adds a Jupytext section in the File menu.

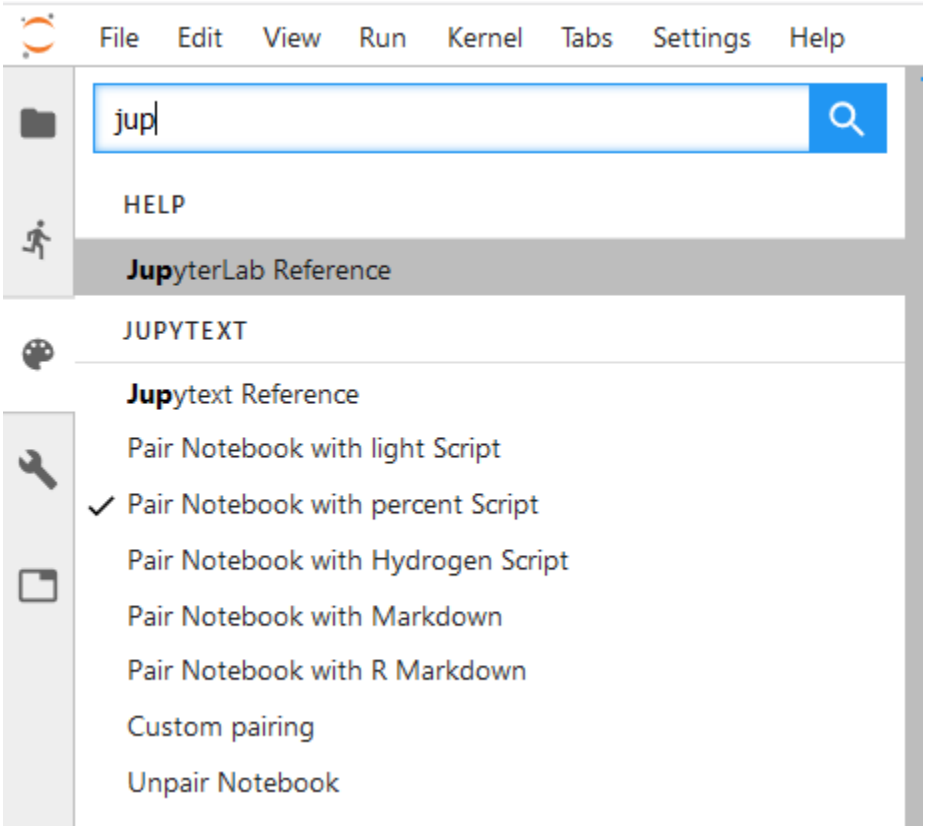


If the extension was not automatically installed, install and activate it with

```
jupyter nbextension install --py jupytext [--user]
jupyter nbextension enable --py jupytext [--user]
```

1.3.3 Jupytext commands in JupyterLab

In JupyterLab, Jupytext adds a set of commands to the command palette:



The Jupytext extension for JupyterLab is bundled with Jupytext. Installing Jupytext will trigger a build of JupyterLab the next time you open it. If you prefer, you can trigger the build manually with

```
jupyter lab build
```

The version of the extension that is shipped with Jupytext requires JupyterLab 2.0. If you prefer to continue using JupyterLab in version 1.x, you should install the version 1.1.1 of the extension:

```
jupyter labextension install jupyterlab-jupytext@1.1.1
```

1.4 Paired notebooks

Jupytext can write a given notebook to multiple files. In addition to the original notebook file, Jupytext can save the input cells to a text file — either a script or a Markdown document. Put the text file under version control for a clear commit history. Or refactor the paired script, and reimport the updated input cells by simply refreshing the notebook in Jupyter.

Select the pairing for a given notebook using either the Jupytext menu in Jupyter Notebook, or the Jupytext commands in JupyterLab.

These command simply add a `"jupytext": {"formats": "ipynb,md"}`-like entry in the notebook metadata. You could also set that metadata yourself with *Edit/Edit Notebook Metadata* in Jupyter Notebook. In JupyterLab, use this extension.

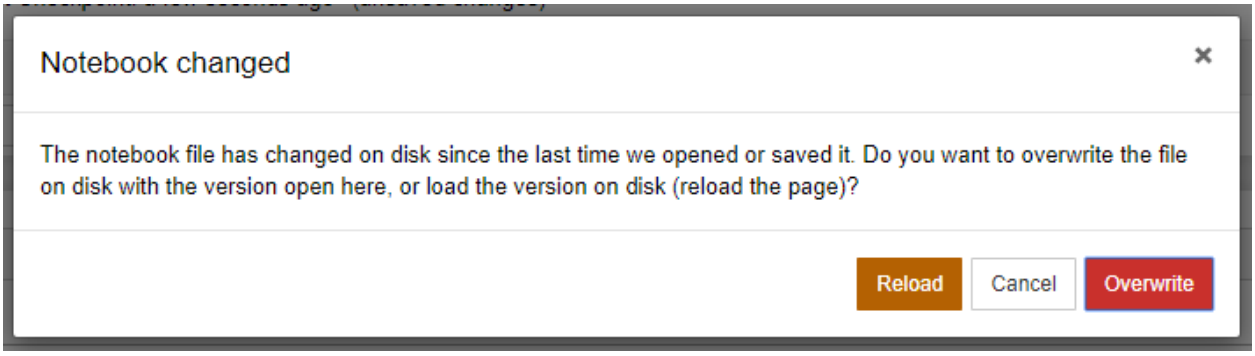
1.4.1 Can I edit a notebook simultaneously in Jupyter and in a text editor?

When saving a paired notebook using Jupytext's contents manager, Jupyter updates both the `.ipynb` and its text representation. The text representation can be edited outside of Jupyter. When the notebook is refreshed in Jupyter, the input cells are read from the text file, and the output cells from the `.ipynb` file.

It is possible (and convenient) to leave the notebook open in Jupyter while you edit its text representation. However, you don't want that the two editors save the notebook simultaneously. To avoid this:

- deactivate Jupyter's autosave, by either toggling the `"Autosave notebook"` menu entry or run `%autosave 0` in a cell of the notebook (see in the faq how to deactivate autosave permanently)
- and refresh the notebook when you switch back from the editor to Jupyter.

In case you forgot to refresh, and saved the Jupyter notebook while the text representation had changed, no worries: Jupyter will ask you which version you want to keep:

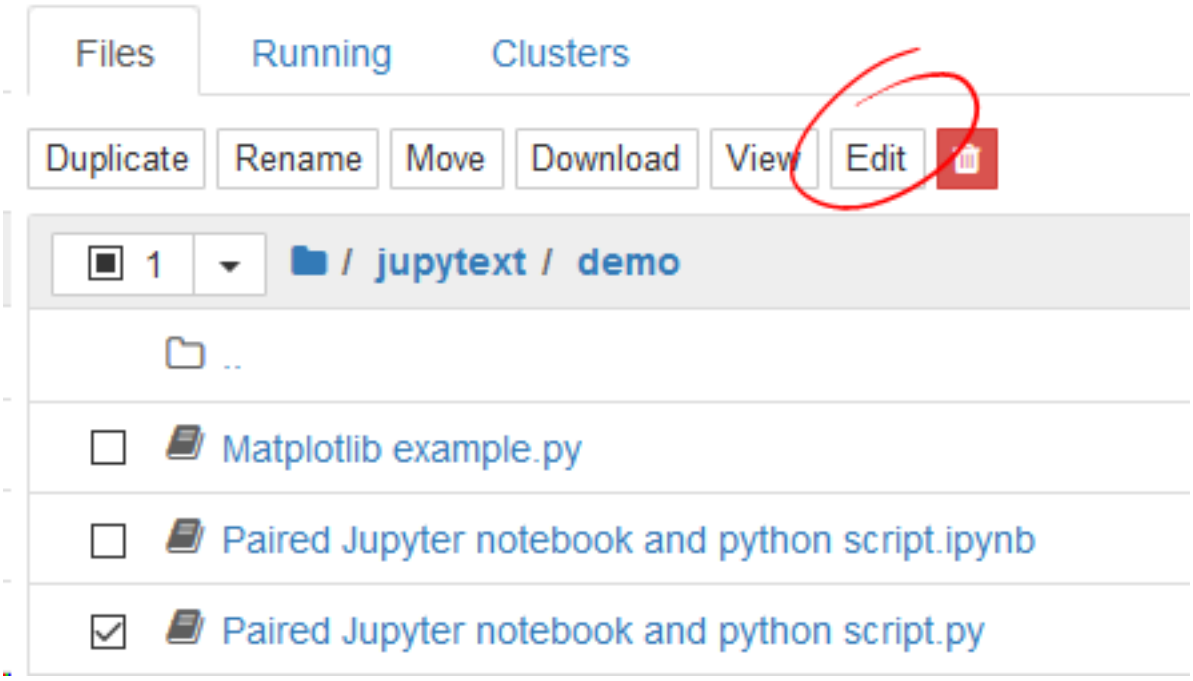


When that occurs, please choose the version in which you made the latest changes. And give a second look to our advice to deactivate the autosaving of notebooks in Jupyter.

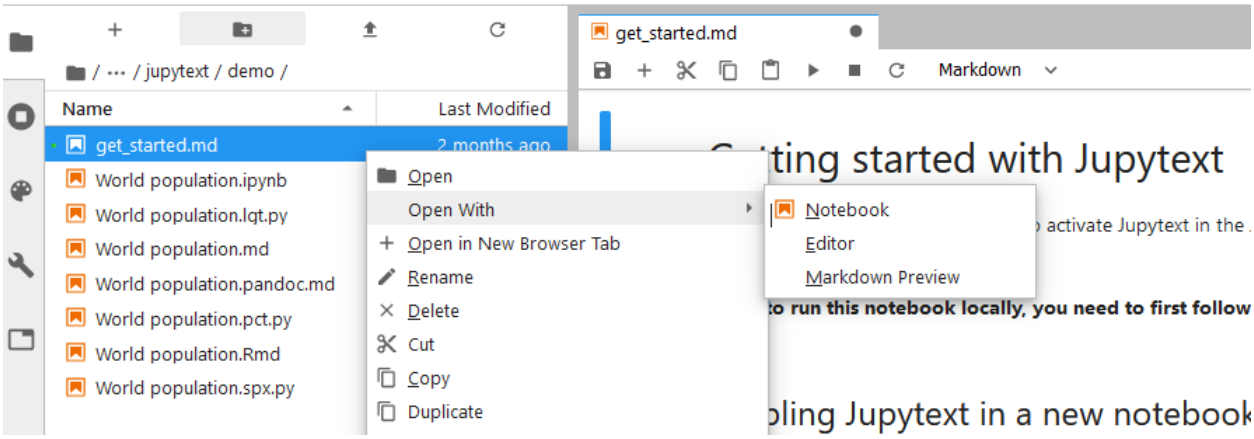
1.4.2 How to open scripts with either the text or notebook view in Jupyter?

With Jupytext's contents manager for Jupyter, scripts and Markdown documents gain a notebook icon. If you don't see the notebook icon, double check the contents manager configuration.

By default, Jupyter Notebook open scripts and Markdown documents as notebooks. If you want to open them with the text editor, select the document and click on *edit*:



In JupyterLab this is slightly different. Scripts and Markdown document also have a notebook icon. But they open as text by default. Open them as notebooks with the *Open With -> Notebook* context menu (available in JupyterLab 0.35 and above):



If do not want to classify scripts or Markdown documents as notebooks, please use the `notebook_extension` option. For instance, if you want to get the notebook icon only for `.ipynb` and `.Rmd` files, set

```
c.ContentsManager.notebook_extensions = "ipynb,Rmd"
```

Please note that, with the above setting, Jupyter will not let you open scripts as notebooks. If you still want to do so, use Jupytext command line (see below) to first convert or pair the script to an `.ipynb` notebook.

1.5 Using Jupytext at the command line

1.5.1 Command line conversion

The package provides a `jupytext` script for command line conversion between the various notebook extensions:

```
jupytext --to py notebook.ipynb                  # convert notebook.ipynb to a .py file
jupytext --to py:percent notebook.ipynb          # convert notebook.ipynb to a .py␣
↪file in the double percent format
jupytext --to py:percent --opt comment_magics=false notebook.ipynb   # same as above␣
↪+ do not comment magic commands
jupytext --to markdown notebook.ipynb            # convert notebook.ipynb to a .md file
jupytext --output script.py notebook.ipynb       # convert notebook.ipynb to a script.
↪py file

jupytext --to notebook notebook.py               # convert notebook.py to an .ipynb␣
↪file with no outputs
jupytext --update --to notebook notebook.py      # update the input cells in the .
↪ipynb file and preserve outputs and metadata

jupytext --to md --test notebook.ipynb           # Test round trip conversion

jupytext --to md --output - notebook.ipynb       # display the markdown version on␣
↪screen
jupytext --from ipynb --to py:percent            # read ipynb from stdin and write␣
↪double percent script on stdout
```

Jupytext has a `--sync` mode that updates all the paired representations of a notebook based on timestamps:

```
jupytext --set-formats ipynb,py notebook.ipynb  # Turn notebook.ipynb into a paired␣
↪ipynb/py notebook
jupytext --sync notebook.ipynb                  # Update whichever of notebook.ipynb/
↪notebook.py is outdated
```

For convenience, when creating a notebook from text you can execute it:

```
jupytext --set-kernel - notebook.md             # create a YAML header with kernel␣
↪metadata matching the current python executable
jupytext --set-formats md:myst notebook.md      # create a YAML header with an␣
↪explicit jupytext format
jupytext --to notebook --execute notebook.md    # convert notebook.md to an .ipynb␣
↪file and run it
```

If you wanted to convert a collection of Markdown files to paired notebooks, and execute them in the current Python environment, you could run:

```
jupytext --set-formats ipynb,md --execute *.md
```

You may also find useful to `--pipe` the text representation of a notebook into tools like `black`:

```
jupytext --sync --pipe black notebook.ipynb    # read most recent version of notebook,
↪ reformat with black, save
```

For programs that don't accept pipes, use `{}` as a placeholder for the name of a temporary file that will contain the text representation of the notebook. For instance, run `pytest` on your notebook with:

```
jupytext --check 'pytest {}' notebook.ipynb    # export the notebook in format␣
↪py:percent in a temp file, run pytest
```

Read more about running `pytest` on notebooks in our example `Tests in a notebook.md`. Note also that on Windows you need to use double quotes instead of single quotes and type e.g. `jupytext --check "pytest {}" notebook.ipynb`.

Execute `jupytext --help` to access the full documentation.

1.5.2 Notebook and cell metadata filters

If you want to preserve (or filter out) certain notebook or cell metadata, change the value of either `notebook_metadata_filter` or `cell_metadata_filter` with the `--update-metadata` option. For instance, if you wish to convert an `.ipynb` document to a `.md` file and preserve all the notebook metadata in that document, run

```
jupytext --to md --update-metadata '{"jupytext": {"notebook_metadata_filter":"all"}}'␣
↪notebook.ipynb
```

Read more on the default and possible values for the metadata filters in this section.

1.5.3 Jupytext as a Git pre-commit hook

Jupytext is also available as a Git pre-commit hook. Use this if you want Jupytext to create and update the `.py` (or `.md...`) representation of the staged `.ipynb` notebooks. All you need is to create an executable `.git/hooks/pre-commit` file with the following content:

```
#!/bin/sh
# For every ipynb file in the git index, add a Python representation
jupytext --from ipynb --to py:light --pre-commit
```

```
#!/bin/sh
# For every ipynb file in the git index:
# - apply black and flake8
# - export the notebook to a Python script in folder 'python'
# - and add it to the git index
jupytext --from ipynb --pipe black --check flake8 --pre-commit
jupytext --from ipynb --to python//py:light --pre-commit
```

If you don't want notebooks to be committed (and only commit the text representations), you can ask the pre-commit hook to unstage notebooks after conversion by adding the following line:

```
git reset HEAD **/*.ipynb
```

Note that these hooks do not update the `.ipynb` notebook when you pull. Make sure to either run `jupytext` in the other direction, or to use our paired notebook and our contents manager for Jupyter. Also, Jupytext does not offer a merge driver. If a conflict occurs, solve it on the text representation and then update or recreate the `.ipynb` notebook. Or give a try to nbdime and its merge driver.

1.5.4 Using Jupytext with the pre-commit package manager

Using Jupytext with the pre-commit package manager is another option. You could add the following to your `.pre-commit-config.yaml` file:

```
repos:
-   repo: local
    hooks:
    - id: jupytext
      name: jupytext
      entry: jupytext --to md
      files: .ipynb
      language: python
```

Here is another `.pre-commit-config.yaml` example that uses the –pre-commit mode of Jupytext to convert all `.ipynb` notebooks to `py:light` representation and unstage the `.ipynb` files before committing.

```
repos:
  -
    repo: local
    hooks:
      -
        id: jupytext
        name: jupytext
        entry: jupytext --from ipynb --to py:light --pre-commit
        pass_filenames: false
        language: python
      -
        id: unstage-ipynb
        name: unstage-ipynb
        entry: git reset HEAD **/*.ipynb
        pass_filenames: false
        language: system
```

1.5.5 Testing the round-trip conversion

Representing Jupyter notebooks as scripts requires a solid round trip conversion. You don't want your notebooks (nor your scripts) to be modified because you are converting them to the other form. Our test suite includes a few hundred tests to ensure that round trip conversion is safe.

You can easily test that the round trip conversion preserves your Jupyter notebooks and scripts. Run for instance:

```
# Test the ipynb -> py:percent -> ipynb round trip conversion
jupytext --test notebook.ipynb --to py:percent

# Test the ipynb -> (py:percent + ipynb) -> ipynb (à la paired notebook) conversion
jupytext --test --update notebook.ipynb --to py:percent
```

Note that `jupytext --test` compares the resulting notebooks according to its expectations. If you wish to proceed to a strict comparison of the two notebooks, use `jupytext --test-strict`, and use the flag `-x` to report with more details on the first difference, if any.

Please note that

- Scripts opened with Jupyter have a default metadata filter that prevents additional notebook or cell metadata to be added back to the script. Remove the filter if you want to store Jupytext's settings, or the kernel information, in the text file.
- Cell metadata are available in the `light` and `percent` formats, as well as in the Markdown and R Markdown formats. R scripts in `spin` format support cell metadata for code cells only. Sphinx Gallery scripts in `sphinx` format do not support cell metadata.

- By default, a few cell metadata are not included in the text representation of the notebook. And only the most standard notebook metadata are exported. Learn more on this in the sections for notebook specific and global settings for metadata filtering.

1.6 Configuration

1.6.1 Per-notebook configuration

The pairing information for one or multiple notebooks can be set on the command line:

```
jupytext --set-formats ipynb,py [--sync] notebook.ipynb
```

You can pair a notebook to as many text representations as you want (see our *World population* notebook in the demo folder). Format specifications are of the form

```
[[root_folder//][path/][prefix]/][suffix.]ext[:format_name]
```

where

- `ext` is one of `ipynb`, `md`, `Rmd`, `jl`, `py`, `R`, `sh`, `cpp`, `q`. Use the `auto` extension to have the script extension chosen according to the Jupyter kernel.
- `format_name` (optional) is either `light` (default for scripts), `nomarker`, `percent`, `hydrogen`, `sphinx` (Python only), `spin` (R only) — see the *format specifications*.
- `root_folder`, `path`, `prefix` and `suffix` allow to save the text representation to files with different names, or in a different folder.

If you want to pair a notebook to a python script in a subfolder named `scripts`, set the formats metadata to `ipynb,scripts//py`. If the notebook is in a `notebooks` folder and you want the text representation to be in a `scripts` folder at the same level, set the Jupytext formats to `notebooks//ipynb,scripts//py`. If you want to pair the notebooks in subtrees, use e.g. `notebooks///ipynb,scripts///py` (and make sure you don't use `notebooks` and trees in subfolder names).

Jupytext accepts a few additional options. These options should be added to the `"jupytext"` section in the metadata — use either the metadata editor or the `--opt/--format-options` argument on the command line.

- `comment_magics`: By default, Jupyter magics are commented when notebooks are exported to any other format than markdown. If you prefer otherwise, use this boolean option, or is global counterpart (see below).
- `notebook_metadata_filter`: By default, Jupytext only exports the `kernelspec` and `jupytext` metadata to the text files. Set `"jupytext": {"notebook_metadata_filter": "-all"}` if you want that the script has no notebook metadata at all. The value for `notebook_metadata_filter` is a comma separated list of additional/excluded (negated) entries, with `all` a keyword that allows to exclude all entries. Use dots to filter recursively the metadata. For instance, use `notebook_metadata_filter="-jupytext.text_representation.jupytext_version"` to remove the `jupytext_version` field in the `jupytext.text_representation` metadata.
- `cell_metadata_filter`: By default, cell metadata `autoscroll`, `collapsed`, `scrolled`, `trusted` and `ExecuteTime` are not included in the text representation. Add or exclude more cell metadata with this option.

1.6.2 Jupytext configuration file

Jupytext's contents manager, and the command line interface, can load some configuration options from a configuration file.

The configuration file should be either in the local or a parent directory, or in any directory listed in

```
from jupytext.config import global_jupytext_configuration_directories
global_jupytext_configuration_directories()
```

which include `XDG_CONFIG_HOME` (defaults to `$HOME/.config`) and `XDG_CONFIG_DIR`.

The name for the configuration file can be any of `jupytext.config.JUPYTEXT_CONFIG_FILES`, i.e. `.jupytext` (in TOML), `jupytext.toml`, `jupytext.yml`, `jupytext.yaml`, `jupytext.json` or `jupytext.py`, and their dot-file versions.

If you want to know, for a given directory, which configuration file Jupytext is using, please execute:

```
from jupytext.config import find_jupytext_configuration_file
find_jupytext_configuration_file('.')
```

If you want to limit the search for a configuration file to a given parent directory, you can create an empty `.jupytext` configuration file in that directory. Alternatively, you can set the search boundaries with an environment variable `JUPYTEXT_CEILING_DIRECTORIES` - a colon-separated list of absolute paths.

If `JUPYTEXT_CEILING_DIRECTORIES` is defined, Jupytext will stop searching for configuration files when it meets one of these path. This can be helpful to avoid searching for configuration files on slow filesystems. It can also be useful if you don't want to use a global configuration - for instance, when running `pytest` on Jupytext, we use `JUPYTEXT_CEILING_DIRECTORIES="/tmp"`.

1.6.3 Configuring paired notebooks globally

Say you want to always associate every `.ipynb` notebook with a `.md` file (and reciprocally). This is done by adding the following to your `jupytext.toml` or `.jupytext.toml` Jupyter configuration file:

```
# Always pair ipynb notebooks to md files
default_jupytext_formats = "ipynb,md"
```

If you prefer using a default `ipynb` - `py:percent` pairing, then that would be:

```
# Always pair ipynb notebooks to py:percent files
default_jupytext_formats = "ipynb,py:percent"
```

And if you prefer to use `jupytext.yml`, in YAML format, as your configuration file, then you could use:

```
# Always pair ipynb notebooks to py:percent files
default_jupytext_formats: "ipynb,py:percent"
```

To disable global pairing for an individual notebook, set formats to a single format, with e.g.:

```
jupytext --set-formats ipynb notebook.ipynb
```

1.6.4 Metadata filtering

You can specify which metadata to include or exclude in the text files created by Jupytext by setting `default_notebook_metadata_filter` (notebook metadata) and `default_cell_metadata_filter` (cell metadata) in the configuration file. They accept a string of comma separated keywords. A minus sign – in front of a keyword means exclusion.

Suppose you want to keep all the notebook metadata but `widgets` and `varInspector` in the YAML header. For cell metadata, you want to allow `ExecuteTime` and `autoscroll`, but not `hide_output`. You can set

```
default_notebook_metadata_filter = "all,-widgets,-varInspector"
default_cell_metadata_filter = "ExecuteTime,autoscroll,-hide_output"
```

If you want that the text files created by Jupytext have no metadata, you may use the global metadata filters below. Please note that with this setting, the metadata is only preserved in the `.ipynb` file.

```
default_notebook_metadata_filter = "-all"
default_cell_metadata_filter = "-all"
```

It is possible to filter nested metadata. For example, if you want to preserve the Jupytext metadata, but not the Jupytext version number, you can use:

```
default_notebook_metadata_filter = "-jupytext.text_representation.jupytext_version"
```

1.6.5 More options

There are a couple more options available - please have a look at the `JupytextConfiguration` class in config.py.

1.7 Using the Jupytext Python library

Jupytext provides the same `read`, `write`, `reads` and `writes` functions as `nbformat`. You can use `jupytext`'s functions as drop-in replacements for `nbformat`'s ones.

1.7.1 Reading notebooks from many text formats

To read text files as notebooks, simply provide the path to a Jupytext-supported format.

```
import jupytext

# Read a notebook from a file
ntbk = jupytext.read('notebook.md')

# Read a notebook from a string
jupytext.reads(text, fmt='md')
```

Jupytext will read in the content and infer metadata about the file from the YAML header (if there is one). If there is no Jupytext header, then Jupytext will make some assumptions about the format based on the file extension.

This function returns an instance of an `nbformat NotebookNode`. You can find more information for working with this notebook representation in the nbformat documentation.

1.7.2 Writing notebooks to many text files

You can also write in-memory notebooks to a variety of text formats by using `jupytext.write`.

Jupytext's implementation provides an additional `fmt` argument, which can be any of the accepted Jupytext extensions (e.g., `py`, `md`, `jl:percent`) If not explicitly provided, the argument is inferred from the file extension.

```
# Return the text representation of a notebook
jupytext.writes(notebook, fmt='py:percent')

# Write a notebook to a file in the desired format
jupytext.write(notebook, 'notebook.py')
jupytext.write(notebook, 'notebook.py', fmt='py:percent')
```

1.8 Supported formats

Jupytext supports conversion between the `.ipynb` format and many different formats. This page describes each format, as well as some considerations for each.

1.8.1 Markdown formats

Jupytext Markdown

Jupytext can save notebooks as Markdown documents. This format is well adapted to tutorials, books, or more generally notebooks that contain more text than code. Notebooks represented in this form are well rendered by most Markdown editors or renderers, including GitHub.

Like all the Jupytext formats, Jupytext Markdown notebooks start with an (optional) YAML header. This header is used to store selected notebook metadata like the kernel information, together with Jupytext's format and version information.

```
---
jupyter:
  jupytext:
    text_representation:
      extension: .md
      format_name: markdown
      format_version: '1.1'
      jupytext_version: 1.1.0
  kernelspec:
    display_name: Python 3
    language: python
    name: python3
---
```

You can add custom notebook metadata like `author` or `title` under the `jupyter:` section, it will be synchronized with the notebook metadata. And if you wish to export more metadata from the notebook, have a look at the paragraph on *metadata filtering*.

In the Markdown format, markdown cells are inserted verbatim and separated with two blank lines.

If you'd like that cell breaks also occurs on Markdown headers, add a `split_at_heading: true` entry in the `jupytext` section in the YAML header, or if you want that option to be the default for all Markdown documents in Jupyter, activate the option on Jupytext's content manager:

```
c.ContentsManager.split_at_heading = True
```

Code cells are encoded using the classical triple backticks, followed by the notebook language. Cell metadata are appended after the language information, with a `key=value` syntax, where `value` is encoded in JSON format. For instance, in a Python notebook, a simple code cell with a `parameters` tag is represented as:

```
```python tags=["parameters"]
param = 5
```
```

Code snippets are turned into code cells in Jupyter as soon as they have an explicit language, when that language is supported in Jupyter. Thus, you have a code snippet that you don't want to execute in Jupyter, you can either

- remove the language information,
- or, start the code snippet with a triple tilde, e.g. `~~~python`, instead of ```` ```python ````
- or, add an `active="md"` cell metadata, or a `.noeval` attribute after the language information, e.g. ```` ```python .noeval ````
- or, surround the code snippet with explicit Markdown cell markers (see below).

Raw cells are delimited with HTML comments, and accept cell metadata in the same key=value format:

```
<!-- #raw -->
raw text
<!-- #endraw -->

<!-- #raw key="value"-->
raw cell with metadata
<!-- #endraw -->
```

Markdown cells can also have explicit markers: use one of `<!-- #md -->` or `<!-- #markdown -->` or `<!-- #region -->` and the corresponding `<!-- #end... -->` counterpart. Note that the `<!-- #region -->` and `<!-- #endregion -->` cells markers are foldable in VS Code, and that you can also insert a title there, e.g. `<!-- #region This is a title for my protected cell -->`. Cell metadata are accepted in the format `key="value"` (`"value"` being encoded in JSON) as for the other cell types.

For a concrete example, see how our `World population.ipynb` notebook in the demo folder is represented in Markdown.

R Markdown

R Markdown is RStudio's format for notebooks, with support for R, Python, and many other languages.

Jupytext's implementation of R Markdown is very similar to that of the Markdown format. The major difference is on code cells, which use R Markdown's convention, i.e. the language and options are surrounded by curly brackets, and the cell metadata are encoded as R objects. For instance our cell with the `parameters` tags would be represented as:

```
```{python tags=c("parameters")}
param = 5
```
```

Python and R notebooks represented in the R Markdown format can run both in Jupyter and RStudio. Note that you can change the default Python environment in RStudio with `RETICULATE_PYTHON` in a `.Renviron` file, see here.

See how our `World population.ipynb` notebook in the demo folder is represented in R Markdown.

Pandoc Markdown

Pandoc, the *Universal document converter*, can read and write Jupyter notebooks - see Pandoc's documentation.

In Pandoc Markdown, all cells are marked with pandoc divs (`:::`). The format is therefore slightly more verbose than the Jupytext Markdown format.

See for instance how our `World population.ipynb` notebook is represented in the `md:pandoc` format.

If you wish to use that format, please install `pandoc` in version 2.7.2 or above, with e.g. `conda install pandoc -c conda-forge`.

MyST Markdown

MyST (Markedly Structured Text) is a markdown flavor that "implements the best parts of reStructuredText". It provides a way to call Sphinx directives and roles from within Markdown, using a *slight* extension of CommonMark markdown. MyST-NB builds on this markdown flavor, to offer direct conversion of Jupyter Notebooks into Sphinx documents.

Similar to the jupytext Markdown format, MyST Markdown uses code blocks to contain code cells. The difference though, is that the metadata is contained in a YAML block:

````
```{code-cell} ipython3
---
other:
  more: true
tags: [hide-output, show-input]
---

print("Hallo!")
```
````

The `ipython3` here is purely optional, as an aide for syntax highlighting. In the round-trip, it is copied from `notebook.metadata.language_info.pygments_lexer`.

Also, where possible the conversion will use the short-hand metadata format (see the MyST guide):

````
```{code-cell} ipython3
:tags: [hide-output, show-input]

print("Hallo!")
```
````

Raw cells are also represented in a similare fashion:

````
```{raw-cell}
:raw_mimetype: text/html

<b>Bold text<b>
```
````

Markdown cells are not wrapped. If a markdown cell has metadata, or directly proceeds another markdown cell, then a block break will be inserted above it, with an (optional) single line JSON representation of the metadata:

```
+++ {"slide": true}

This is a markdown cell with metadata

+++

This is a new markdown cell with no metadata
```

See for instance how our `World population.ipynb` notebook is represented in the `myst` format.

If you wish to use that format, please install `conda install -c conda-forge myst-parser`, or `pip install jupytext[myst]`.

Tip: You can use the myst-highlight VS Code extension to provide better syntax highlighting for this format.

1.8.2 Notebooks as scripts

The `light` format

The `light` format was created for this project. That format can read any script in one of these languages as a Jupyter notebook, even scripts which were never prepared to become a notebook.

When a script in the `light` format is converted to a notebook, Jupytext code paragraphs are turned into code cells, and comments that are not adjacent to code are converted to Markdown cells. Cell breaks occurs on blank lines outside of functions, classes or multiline comments.

For instance, in this example we have three cells:

```
# This is a multiline
# Markdown cell

# Another Markdown cell


# This is a code cell
class A():
    def one():
        return 1

    def two():
        return 2
```

Code cells can contain multiple code paragraphs. In that case Jupytext uses an explicit start-of-cell delimiter that is, by default, `# +` (`// +` in C++, etc). The default end of cell delimiter is `# -`, and can be omitted when followed by another explicit start of cell marker, or the end of the file:

```
# +
# A single code cell made of two paragraphs
a = 1


def f(x):
    return x+a
```

Metadata can be associated to a given cell using a key/value representation:

```
# + key="value"
# A code cell with metadata

# + [markdown] key="value"
# A Markdown cell with metadata
```

The `light` format can use custom cell markers instead of `# +` or `# -`. If you prefer to mark cells with VS Code/PyCharm (resp. Vim) folding markers, set `"cell_markers": "region,endregion"` (resp. `"{{{,}}}"`) in the jupytext section of the notebook metadata. If you want to configure this as a global default, add either

```
c.ContentsManager.default_cell_markers = "region,endregion"  # Use VS Code/PyCharm␣
↪region folding delimiters
```

or

```
c.ContentsManager.default_cell_markers = "{{{,}}}"           # Use Vim region folding␣
↪delimiters
```

to your `.jupyter/jupyter_notebook_config.py` file.

See how our `World population.ipynb` notebook is represented in that format.

The `nomarker` format

The `nomarker` format is a variation of the `light` format with no cell marker at all. Please note that this format does not provide round-trip consistency - code cells are split on code paragraphs. By default, the `nomarker` format still includes a YAML header - if you prefer to also remove the header, set `"notebook_metadata_filter": "-all"` in the jupytext section of your notebook metadata.

The `percent` format

The `percent` format is a representation of Jupyter notebooks as scripts, in which all cells are explicitely delimited with a commented double percent sign `# %%`. The `percent` format is currently available for these languages.

The format was introduced by Spyder in 2013, and is now supported by many editors, including

- Spyder IDE,
- Hydrogen, a package for Atom,
- VS Code,
- Python Tools for Visual Studio,
- and PyCharm Professional.

Our implementation of the `percent` format is as follows: cells can have

- a title
- a cell type (`markdown`, `md` or `raw`, omitted for code cells)
- and cell metadata like in this example:

```
# %% Optional title [cell type] key="value"
```

In the `percent` format, our previous example becomes:

```
# %% [markdown]
# This is a multiline
# Markdown cell

# %% [markdown]
# Another Markdown cell


# %%
# This is a code cell
class A():
```

(continues on next page)

(continued from previous page)

```
    def one():
        return 1

    def two():
        return 2
```

In the case of Python scripts, Markdown cells do accept multiline comments:

```
# %% [markdown]
"""
This is a Markdown cell
that uses multiline comments
"""
```

By default Jupytext will use line comments when it converts your Jupyter notebooks for `percent` scripts. If you prefer to use multiline comments for all text cells, add a `{"jupytext": {"cell_markers": "\"\"\""}}` metadata to your notebook, either with the notebook metadata editor in Jupyter, or at the command line:

```
jupytext --update-metadata '{"jupytext": {"cell_markers": "\"\"\""}}' notebook.ipynb -
↪-to py:percent
```

If you want to use multiline comments for all your paired notebooks, you could also add

```
c.ContentsManager.default_cell_markers = '"""'
```

to your `.jupyter/jupyter_notebook_config.py` file.

See how our `World population.ipynb` notebook is represented in the `percent` format.

The `hydrogen` format

By default, *Jupyter magics* are commented in the `percent` representation. If you run the percent scripts in Hydrogen, use the `hydrogen` format, a variant of the `percent` format that does not comment Jupyter magic commands.

Sphinx-gallery scripts

Another popular notebook-like format for Python scripts is the Sphinx-gallery format. Scripts that contain at least two lines with more than twenty hash signs are classified as Sphinx-Gallery notebooks by Jupytext.

Comments in Sphinx-Gallery scripts are formatted using reStructuredText rather than markdown. They can be converted to markdown for a nicer display in Jupyter by adding a `c.ContentsManager.sphinx_convert_rst2md = True` line to your Jupyter configuration file. Please note that this is a non-reversible transformation—use this only with Binder. Revert to the default value `sphinx_convert_rst2md = False` when you edit Sphinx-Gallery files with Jupytext.

Turn a GitHub repository containing Sphinx-Gallery scripts into a live notebook repository with Binder and Jupytext by adding only two files to the repo:

- `binder/requirements.txt`, a list of the required packages (including `jupytext`)
- `.jupyter/jupyter_notebook_config.py` with the following contents:

```
c.NotebookApp.contents_manager_class = "jupytext.TextFileContentsManager"
c.ContentsManager.preferred_jupytext_formats_read = "py:sphinx"
c.ContentsManager.sphinx_convert_rst2md = True
```

Our sample notebook is also represented in sphinx format here.

1.8.3 Jupytext format options

Metadata filtering

All the Jupytext formats (except Sphinx Gallery scripts) store a selection of the notebook metadata in a YAML header at the top of the text file. By default, Jupytext only includes the kernelspec and jupytext metadata (the remaining notebook metadata are preserved in the .ipynb document when you use paired notebook).

If you want to include more (or less) jupyter metadata here, add a notebook_metadata_filter option to the jupytext metadata. The additional metadata will be added to the jupyter: section in the YAML header (or, at the root of the YAML header for the md:pandoc and md:myst formats). The value for notebook_metadata_filter is a comma separated list of additional/excluded (negated) entries, with all a keyword that allows to exclude all entries. For instance, if you don't want to store any notebook metadata in the text file, use notebook_metadata_filter: -all. If you want to store the whole, unfiltered notebook metadata then use notebook_metadata_filter: all. And if you want the default, plus a few specific section, use e.g. notebook_metadata_filter: section_one,section_two.

Similarly, cell metadata can be filtered with the cell_metadata_filter option. To minimize the differences when a notebook is edited, Jupytext's default cell metadata filter does not include the autoscroll, collapsed, scrolled, trusted and ExecuteTime cell metadata in the text representation.

Magic commands

Jupyter notebooks often include *magic commands* like %load_ext or %matplotlib inline. These commands are Jupyter specific and cannot be executed by the classical interpreter.

By default, magic commands are commented out in all the Jupytext formats, with the exception of the Markdown format (not meant to be executed) and the Hydrogen format. You can change this by setting a comment_magics option (true or false) in the Jupytext section.

Active and inactive cells

Sometimes you want a specific cell to be executable only in the .ipynb files, or only in the .py or .Rmd representation. To this end Jupytext introduces the notion of *active* cells.

Mark a code cell with an "active": "ipynb" metadata or with an active-ipynb tag if you want it to be commented out in the paired script.

Mark a raw cell with an "active": "py" metadata or with an active-py tag if you want it to be inactive in the notebook but active in the script.

1.9 Frequently Asked Questions

1.9.1 What is Jupytext?

Jupytext is a Python package that provides *two-way* conversion between Jupyter Notebooks and several other text-based formats like Markdown documents or scripts.

1.9.2 Why would I want to convert my notebooks to text?

The text representation only contains the part of the notebook that you wrote (not the outputs). You get a cleaner diff history. Thanks to the *two-way* conversion, you can also act on the text file and then propagate the changes to the original `.ipynb` file. Refactor your code or merge multiple contributions easily!

1.9.3 How do I use Jupytext?

Open the notebook that you want to version control. *Pair* the notebook to a script or a Markdown file using either the Jupytext Menu in Jupyter Notebook or the Jupytext Commands in JupyterLab.

Save the notebook, and you get two copies of the notebook: the original `*.ipynb` file, together with its paired text representation.

Read more about how to use Jupytext in the *documentation*.

1.9.4 Which Jupytext format do you recommend?

Notebooks that contain more text than code are best represented as Markdown documents. These are conveniently edited in IDEs and are also well rendered on GitHub.

Saving notebooks as scripts is an appropriate choice when you want to act on the code (refactor the code, import it in another script or notebook, etc). Use the `percent` format if you prefer to get explicit cell markers (compatible with VScode, PyCharm, Spyder, Hydrogen...). And if you prefer to get the minimal amount of cell markers, go for the `light` format.

1.9.5 Can I see a sample of each format?

Go to our demo folder and see how our sample `World population` notebook is represented in each format.

1.9.6 Can I edit the paired text file?

Yes! When you're done, reload the notebook in Jupyter. There, you will see the updated input cells combined with the matching output cells from the `.ipynb` file.

1.9.7 Do I need to close my notebook in Jupyter?

No, you don't (*). You can edit the paired text file and simply refresh your navigator to reload the updated input cells. When you refresh the notebook, the kernel variables are preserved, so you can continue your work where you left it.

(*) Please read about Jupyter's autosave below.

1.9.8 How do paired notebooks work?

The `.ipynb` file contains the full notebook. The paired text file only contains the input cells and selected metadata. When the notebook is loaded by Jupyter, input cells are loaded from the text file, while the output cells and the filtered metadata are restored using the `.ipynb` file. When the notebook is saved in Jupyter, the two files are updated to match the current content of the notebook.

1.9.9 How do I remove pairing?

Paired Jupyter Notebooks contains specific `jupytext` metadata that you may want to remove. You may want to keep the pairing only while editing the files, and when it comes the time to distribute them, it may make sense to remove the pairing. To do so, you can update the metadata in the `.ipynb` files as follows:

```
jupytext --update-metadata '{"jupytext": null}' path/to/notebooks/*.ipynb
```

1.9.10 Can I create a notebook from a text file?

Certainly. Open your pre-existing scripts or Markdown files as notebooks with a click in Jupyter Notebook, and with the *Open as Notebook* menu in JupyterLab.

The text formats do not store the output cells. If you want to preserve these when you refresh the notebook, be sure to pair the text file to an `.ipynb` file.

If you want to convert text formats to notebooks programmatically, use one of

```
jupytext --to ipynb *.md                          # convert all .md files to notebooks␣
↪with no outputs
jupytext --to ipynb --execute *.md                # convert all .md files to notebooks␣
↪and execute them
jupytext --set-formats ipynb,md --execute *.md  # convert all .md files to paired␣
↪notebooks and execute them
```

1.9.11 I want a specific cell to be commented out in the paired script

That's possible! See how to activate or deactivate cells.

1.9.12 Which files should I version control?

Unless you want to version the outputs, you should version *only the text representation*. The paired `.ipynb` file can safely be deleted. It will be recreated locally the next time you open the notebook (from the text file) and save it.

Note that if you version both the `.md` and `.ipynb` files, you can configure `git diff` to ignore the diffs on the `.ipynb` files.

1.9.13 I have modified a text file, but git reports no diff for the paired `.ipynb` file

The synchronization between the two files happens when you reload and *save* the notebook in Jupyter, or when you explicitly run `jupytext --sync`. If you want to force the synchronization on every commit, create a file `.git\hooks\pre-commit` with the following content:

```
#!/bin/sh
jupytext --sync --pre-commit
```

and make it executable: `chmod u+x .git\hooks\pre-commit`.

Alternatively, VIM users can give a try to the jupytext.vim plugin.

1.9.14 Jupyter warns me that the file has changed on disk

By default, Jupyter saves your notebook every 2 minutes. Fortunately, it is also aware that you have edited the text file, yielding this message.

You should simply click on *Reload*.

Note you can deactivate Jupyter's autosave function with the Jupytext Menu in Jupyter Notebook, and with the *Autosave Document* setting in JupyterLab. If you want to permanently deactivate autosave in Jupyter Notebook, use a `custom.js` file:

```
mkdir -p ~/.jupyter/custom
echo "Jupyter.notebook.set_autosave_interval(0);" >> ~/.jupyter/custom/custom.js
```

1.9.15 When I reload, Jupyter warns me that my notebook has unsaved changes

Oh - you have edited both the notebook and the paired text file at the same time? If you know which version you want to keep, save it and reload the other. If you want to compare and merge both versions, backup the text file (with e.g. `git stash`), save the notebook, and merge the updated paired file with the backup (with e.g. `git stash pop`). Then, refresh the notebook in Jupyter.

If your IDE has the ability to compare the changes in memory versus on disk (like PyCharm), you can simply save the notebook and let your IDE do the merge.

1.9.16 Jupyter complains that the `.ipynb` file is more recent than the text representation

This happens if you have edited the `.ipynb` file outside of Jupyter. It is a safeguard to avoid overwriting the input cells of the notebook with an outdated text file.

Manual action is requested as the paired text representation may be outdated. Please edit (`touch`) the paired `.md` or `.py` file if it is not outdated, or if it is, delete it, or update it with

```
jupytext --sync notebook.ipynb
```

1.9.17 Can I use Jupytext with Jupyter Hub, Binder, Nteract, Colab, Saturn or Azure?

Jupytext is compatible with Jupyter Hub (execute `pip install jupytext --user` to install it in user mode) and with Binder (add `jupytext` to the project requirements and `jupyter lab build` to `postBuild`).

If you use another editor than Jupyter Notebook, Lab or Hub, you probably can't get Jupytext there. However you can still use Jupytext at the command line to manually sync the two representations of the notebook:

```
jupytext --set-formats ipynb,py:light notebook.ipynb    # Pair a notebook to a light␣
↪script
jupytext --sync notebook.ipynb                          # Sync the two representations
```

1.9.18 Can I re-write my git history to use text files instead of notebooks?

Indeed, you could substitute every `.ipynb` file in the project history with its Jupytext Markdown representation.

Technically this is available in just one command, which results in a complete rewrite of the history. Please experiment that in a branch, and think twice before pushing the result...

```
git filter-branch --tree-filter 'jupytext --to md */*.ipynb && rm -f */*.ipynb' HEAD
```

See the result and the cleaner diff history in the case of the Python Data Science Handbook.