nbformat Documentation

Release 5.3

Jupyter Development Team

Apr 04, 2022

Contents

Jupyter (né IPython) notebook files are simple JSON documents, containing text, source code, rich media output, and metadata. Each segment of the document is stored in a cell.

Contents:

CHAPTER 1

The Notebook file format

The official Jupyter Notebook format is defined with this JSON schema, which is used by Jupyter tools to validate notebooks.

This page contains a human-readable description of the notebook format.

Note: *All* metadata fields are optional. While the types and values of some metadata fields are defined, no metadata fields are required to be defined. Any metadata field may also be ignored.

1.1 Top-level structure

At the highest level, a Jupyter notebook is a dictionary with a few keys:

- metadata (dict)
- nbformat (int)
- nbformat_minor (int)
- cells (list)

```
{
  "metadata" : {
    "kernel_info": {
        # if kernel_info is defined, its name field is required.
        "name" : "the name of the kernel"
    },
    "language_info": {
        # if language_info is defined, its name field is required.
        "name" : "the programming language of the kernel",
        "version": "the version of the language",
        "codemirror_mode": "The name of the codemirror mode to use [optional]"
    }
```

(continues on next page)

(continued from previous page)

```
  },
  "nbformat": 4,
  "nbformat_minor": 0,
  "cells" : [
      # list of cell dictionaries, see below
  ],
}
```

Some fields, such as code input and text output, are characteristically multi-line strings. When these fields are written to disk, they **may** be written as a list of strings, which should be joined with `''` when reading back into memory. In programmatic APIs for working with notebooks (Python, Javascript), these are always re-joined into the original multi-line string. If you intend to work with notebook files directly, you must allow multi-line string fields to be either a string or list of strings.

1.2 Cell Types

There are a few basic cell types for encapsulating code and text. All cells have the following basic structure:

```
{
  "cell_type" : "type",
  "metadata" : {},
  "source" : "single string or [list, of, strings]",
}
```

Note: On disk, multi-line strings **MAY** be split into lists of strings. When read with the nbformat Python API, these multi-line strings will always be a single string.

1.2.1 Markdown cells

Markdown cells are used for body-text, and contain markdown, as defined in GitHub-flavored markdown, and implemented in marked.

```
{
  "cell_type" : "markdown",
  "metadata" : {},
  "source" : "[multi-line *markdown*]",
}
```

Changed in version nbformat: 4.0

Heading cells have been removed in favor of simple headings in markdown.

1.2.2 Code cells

Code cells are the primary content of Jupyter notebooks. They contain source code in the language of the document's associated kernel, and a list of outputs associated with executing that code. They also have an execution_count, which must be an integer or `null`.

```
{
  "cell_type" : "code",
  "execution_count": 1, # integer or null
  "metadata" : {
      "collapsed" : True, # whether the output of the cell is collapsed
      "scrolled": False, # any of true, false or "auto"
  },
  "source" : "[some multi-line code]",
  "outputs": [{
      # list of output dicts (described below)
      "output_type": "stream",
      ...
  }],
}
```

Changed in version nbformat: 4.0

`input` was renamed to `source`, for consistency among cell types.

Changed in version nbformat: 4.0

`prompt_number` renamed to `execution_count`

1.2.3 Code cell outputs

A code cell can have a variety of outputs (stream data or rich mime-type output). These correspond to messages produced as a result of executing the cell.

All outputs have an `output_type` field, which is a string defining what type of output it is.

stream output

```
{
  "output_type" : "stream",
  "name" : "stdout", # or stderr
  "text" : "[multiline stream text]",
}
```

Changed in version nbformat: 4.0

The `stream` key was changed to `name` to match the stream message.

display_data

Rich display outputs, as created by `display_data` messages, contain data keyed by mime-type. This is often called a mime-bundle, and shows up in various locations in the notebook format and message spec. The metadata of these messages may be keyed by mime-type as well.

```
{
  "output_type" : "display_data",
  "data" : {
    "text/plain" : "[multiline text data]",
    "image/png": "[base64-encoded-multiline-png-data]",
    "application/json": {
      # JSON data is included as-is
```

(continues on next page)

(continued from previous page)

```
      "key1": "data",
      "key2": ["some", "values"],
      "key3": {"more": "data"}
    },
    "application/vnd.exampleorg.type+json": {
      # JSON data, included as-is, when the mime-type key ends in +json
      "key1": "data",
      "key2": ["some", "values"],
      "key3": {"more": "data"}
    }
  },
  "metadata" : {
    "image/png": {
      "width": 640,
      "height": 480,
    },
  },
}
```

Changed in version nbformat: 4.0

`application/json` output is no longer double-serialized into a string.

Changed in version nbformat: 4.0

mime-types are used for keys, instead of a combination of short names (`text`) and mime-types, and are stored in a `data` key, rather than the top-level. i.e. `output.data['image/png']` instead of `output.png`.

execute_result

Results of executing a cell (as created by `displayhook` in Python) are stored in `execute_result` outputs. *execute_result* outputs are identical to `display_data`, adding only a `execution_count` field, which must be an integer.

```
{
  "output_type" : "execute_result",
  "execution_count": 42,
  "data" : {
    "text/plain" : "[multiline text data]",
    "image/png": "[base64-encoded-multiline-png-data]",
    "application/json": {
      # JSON data is included as-is
      "json": "data",
    },
  },
  "metadata" : {
    "image/png": {
      "width": 640,
      "height": 480,
    },
  },
}
```

Changed in version nbformat: 4.0

`pyout` renamed to `execute_result`

Changed in version nbformat: 4.0

`prompt_number` renamed to `execution_count`

error

Failed execution may show an error:

```
{
    'output_type': 'error',
    'ename' : str,   # Exception name, as a string
    'evalue' : str,  # Exception value, as a string

    # The traceback will contain a list of frames,
    # represented each as a string.
    'traceback' : list,
}
```

Changed in version nbformat: 4.0

`pyerr` renamed to `error`

1.2.4 Raw NBConvert cells

A raw cell is defined as content that should be included *unmodified* in nbconvert output. For example, this cell could include raw LaTeX for nbconvert to pdf via latex, or restructured text for use in Sphinx documentation.

The notebook authoring environment does not render raw cells.

The only logic in a raw cell is the *format* metadata field. If defined, it specifies which nbconvert output format is the intended target for the raw cell. When outputting to any other format, the raw cell's contents will be excluded. In the default case when this value is undefined, a raw cell's contents will be included in any nbconvert output, regardless of format.

```
{
  "cell_type" : "raw",
  "metadata" : {
    # the mime-type of the target nbconvert format.
    # nbconvert to formats other than this will exclude this cell.
    "format" : "mime/type"
  },
  "source" : "[some nbformat output text]"
}
```

1.2.5 Cell attachments

Markdown and raw cells can have a number of attachments, typically inline images that can be referenced in the markdown content of a cell. The `attachments` dictionary of a cell contains a set of mime-bundles (see display_data) keyed by filename that represents the files attached to the cell.

Note: The `attachments` dictionary is an optional field and can be undefined or empty if the cell does not have any attachments.

```
{
  "cell_type" : "markdown",
  "metadata" : {},
  "source" : ["Here is an *inline* image ![inline image](attachment:test.png)"],
  "attachments" : {
    "test.png": {
        "image/png" : "base64-encoded-png-data"
    }
  }
}
```

1.2.6 Cell ids

Since the 4.5 schema release, all cells have an `id` field which must be a string of length 1-64 with alphanumeric, -, and _ as legal characters to use. These ids must be unique to any given Notebook following the nbformat spec.

The full rules and guidelines for using cells ids is captured in the corresponding JEP Proposal.

If attempting to add similiar support to other languages supporting notebooks specs, this Example PR can be used as a reference to follow.

1.3 Backward-compatible changes

The notebook format is an evolving format. When backward-compatible changes are made, the notebook format minor version is incremented. When backward-incompatible changes are made, the major version is incremented.

As of nbformat 4.x, backward-compatible changes include:

- new fields in any dictionary (notebook, cell, output, metadata, etc.)
- new cell types
- new output types

New cell or output types will not be rendered in versions that do not recognize them, but they will be preserved.

Because the nbformat python package used to be less strict about validating notebook files, two features have been backported from nbformat 4.x to nbformat 4.0. These are:

- `attachment` top-level keys in the Markdown and raw cell types (backported from nbformat 4.1)
- Mime-bundle attributes are JSON data if the mime-type key ends in `+json` (backported from nbformat 4.2)

These backports ensure that any valid nbformat 4.4 file is also a valid nbformat 4.0 file.

1.4 Metadata

Metadata is a place that you can put arbitrary JSONable information about your notebook, cell, or output. Because it is a shared namespace, any custom metadata should use a sufficiently unique namespace, such as *metadata.kaylees_md.foo = "bar"*.

Metadata fields officially defined for Jupyter notebooks are listed here:

1.4.1 Notebook metadata

The following metadata keys are defined at the notebook level:

Key	Value	Interpretation
kernelspec	dict	A kernel specification
authors	list of dicts	A list of authors of the document

A notebook's authors is a list of dictionaries containing information about each author of the notebook. Currently, only the name is required. Additional fields may be added.

```
nb.metadata.authors = [
    {
        'name': 'Fernando Perez',
    },
    {
        'name': 'Brian Granger',
    },
]
```

1.4.2 Cell metadata

Official Jupyter metadata, as used by Jupyter frontends should be placed in the *metadata.jupyter* namespace, for example *metadata.jupyter.foo = "bar"*.

The following metadata keys are defined at the cell level:

Key	Value	Interpretation
collapsed	bool	Whether the cell's output container should be collapsed
scrolled	bool or 'auto'	Whether the cell's output is scrolled, unscrolled, or autoscrolled
deletable	bool	If False, prevent deletion of the cell
editable	bool	If False, prevent editing of the cell (by definition, this also prevents deleting the cell)
format	'mime/type'	The mime-type of a *Raw NBConvert Cell*
name	str	A name for the cell. Should be unique across the notebook. Uniqueness must be verified outside of the json schema.
tags	list of str	A list of string tags on the cell. Commas are not allowed in a tag
jupyter	dict	A namespace holding jupyter specific fields. See docs below for more details
execution	dict	A namespace holding execution specific fields. See docs below for more details

The following metadata keys are defined at the cell level within the *jupyter* namespace

Key	Value	Interpretation
source_hidden	bool	Whether the cell's source should be shown
outputs_hidden	bool	Whether the cell's outputs should be shown

The following metadata keys are defined at the cell level within the *execution* namespace. These are lower level fields capturing common kernel message timestamps for better visibility in applications where needed. Most users will not look at these directly.

Key	Value	Interpretation
iopub.execute_input	ISO 8601 format	Indicates the time at which the kernel broadcasts an execute_input message. This represents the time when request for work was received by the kernel.
iopub.status.busy	ISO 8601 format	Indicates the time at which the iopub channel's kernel status message is 'busy'. This represents the time when work was started by the kernel.
shell.execute_reply	ISO 8601 format	Indicates the time at which the shell channel's execute_reply status message was created. This represents the time when work was completed by the kernel.
iopub.status.idle	ISO 8601 format	Indicates the time at which the iopub channel's kernel status message is 'idle'. This represents the time when the kernel is ready to accept new work.

1.4.3 Output metadata

The following metadata keys are defined for code cell outputs:

Key	Value	Interpretation
isolated	bool	Whether the output should be isolated into an IFrame

CHAPTER 2

Supported markup formats

The Jupyter Notebook format supports Markdown in text cells. There is not a strict specification for the flavor of markdown that is supported, but this page should help guide the user / developer in understanding what behavior to expect with Jupyter interfaces and markup languages.

2.1 What flavor of Markdown does the notebook format support?

Most Jupyter Notebook interfaces use the marked.js JavaScript library for rendering markdown. This supports markdown in the following markdown flavors:

- CommonMark.
- GitHub Flavored Markdown

See the Marked.js specification page for more information.

Note: Currently, as the Marked.js specification changes, so to will the behavior of Markdown in many notebook interfaces.

2.2 MathJax configuration

There are a few extra modifications that Jupyter interfaces tend to use for rendering markdown. Specifically, they automatically render mathematical equations using MathJax.

This is currently the MathJax configuration that is used:

```
{
    tex2jax: {
        inlineMath: [ ['$','$'], ["\\(","\\)"] ],
        displayMath: [ ['$$','$$'], ["\\[","\\]"] ],
```

(continues on next page)

(continued from previous page)

```
        processEscapes: true,
        processEnvironments: true
    },
    MathML: {
        extensions: ['content-mathml.js']
    },
    displayAlign: 'center',
    "HTML-CSS": {
        availableFonts: [],
        imageFont: null,
        preferredFont: null,
        webFont: "STIX-Web",
        styles: {'.MathJax_Display': {"margin": 0}},
        linebreaks: { automatic: true }
    },
}
```

See the MathJax script for the classic Notebook UI for one example.

CHAPTER 3

Python API for working with notebook files

3.1 Reading and writing

The reading functions require you to pass the *as_version* parameter. Your code should specify the notebook format that it knows how to work with: for instance, if your code handles version 4 notebooks:

```
nb = nbformat.read('path/to/notebook.ipynb', as_version=4)
```

This will automatically upgrade or downgrade notebooks in other versions of the notebook format to the structure your code knows about.

`nbformat.`**`NO_CONVERT`**
: This special value can be passed to the reading and writing functions, to indicate that the notebook should be loaded/saved in the format it's supplied.

`nbformat.`**`current_nbformat`**
`nbformat.`**`current_nbformat_minor`**
: These integers represent the current notebook format version that the nbformat module knows about.

3.2 NotebookNode objects

The functions in this module work with `NotebookNode` objects, which are like dictionaries, but allow attribute access (`nb.cells`). The structure of these objects matches the notebook format described in *The Notebook file format*.

3.3 Other functions

3.4 Constructing notebooks programmatically

These functions return `NotebookNode` objects with the necessary fields.

3.5 Notebook signatures

This machinery is used by the notebook web application to record which notebooks are *trusted*, and may show dynamic output as soon as they're loaded. See Security in notebook documents for more information.

3.5.1 Signature storage

Signatures are stored using a pluggable `SignatureStore` subclass. To implement your own, override the methods below and configure `NotebookNotary.store_factory`.

By default, `NotebookNotary` will use an SQLite based store if SQLite bindings are available, and an in-memory store otherwise.

CHAPTER 4

Changes in nbformat

4.1 In Development

4.2 5.3.0

- Use *f'astjsonschema'* by default
- Adopt `pre-commit` and auto-formatters
- Increase minimum `jsonschema` to 2.6, handle warnings

4.3 5.2.0

- Add ability to capture validation errors
- Update supported python versions
- Ensure nbformat minor version is present when upgrading
- Only fix cell ID validation issues if asked
- Return the notebook when no conversion is needed
- Catch AttributeErrors stemming from ipython_genutils as ValidationErrors on read
- Don't list pytest-cov as a test dependency
- Remove dependency on IPython genutils
- Include tests in sdist but not wheel

4.4 5.1.3

- Change id generation to be hash based to avoid problematic word combinations
- Added tests for python 3.9
- Fixed setup.py build operations to include package data

4.5 5.1.2

- Fixed missing file in manifest

4.6 5.1.1

- Changes convert.upgrade to upgrade minor 4.x versions to 4.5

4.7 5.1.0

- Implemented CellIds from JEP-62
- Fixed a regression introduced when using fastjsonschema, which does not directly support to validate a "reference"/"subschema"
- Removed unreachable/unneeded code
- Added CI workflow for package release on tag push

4.8 5.0.8

- Add optional support for using *fastjsonschema* as the JSON validation library. To enable fast validation, install *fastjsonschema* and set the environment variable *NBFORMAT_VALIDATOR* to the value *fastjsonschema*.

4.9 5.0.7

- Fixed a bug where default values for validator.get_validator() failed with an import error

4.10 5.0.6

- nbformat.read() function has a better duck-type interface and will raise more meaningful error messages if it can't parse a notebook document.

4.11 5.0.5

- Allow notebook format 4.0 and 4.1 to have the arbitrary JSON mimebundles from format 4.2 for pragmatic purposes.
- Support reading/writing path-like objects has been added to read operations.

4.12 5.0.4

- Fixed issue causing python 2 to pick up 5.0.x releases.

4.13 5.0.3

- Removed debug print statements from project.

4.14 5.0.2

- Added schema validation files for older versions. This was breaking notebook generation.

4.15 5.0.1

4.16 5.0

5.0 on GitHub

- Starting with 5.0, `nbformat` is now Python 3 only (>= 3.5)
- Add execution timings in code cell metadata for v4 spec. `"metadata": { "execution": {...}}` should be populated with kernel-specific timing information.
- Documentation for how markup is used in notebooks added
- Link to json schema docs from format page added
- Documented the editable metadata flag
- Update description for collapsed field
- Documented notebook format versions 4.0-4.3 with accurate json schema specification files
- Clarified info about name's meaning for cells
- Added a default execution_count of None for new_output_cell('execute_result')
- Added support for handling nbjson kwargs
- Wheels now correctly have a LICENSE file
- Travis builds now have a few more execution environments

4.17 4.4

4.4 on GitHub

- Explicitly state that metadata fields can be ignored.
- Introduce official jupyter namespace inside metadata (`metadata.jupyter`).
- Introduce `source_hidden` and `outputs_hidden` as official front-end metadata fields to indicate hiding source and outputs areas. **NB**: These fields should not be used to hide elements in exported formats.
- Fix ending the redundant storage of signatures in the signature database.
- `nbformat.validate()` can be set to not raise a ValidationError if additional properties are included.
- Fix for errors with connecting and backing up the signature database.
- Dict-like objects added to NotebookNode attributes are now transformed to be NotebookNode objects; transformation also works for *.update()*.

4.18 4.3

4.3 on GitHub

- A new pluggable `SignatureStore` class allows specifying different ways to record the signatures of trusted notebooks. The default is still an SQLite database. See *Signature storage* for more information.
- `nbformat.read()` and `nbformat.write()` accept file paths as bytes as well as unicode.
- Fix for calling `nbformat.validate()` on an empty dictionary.
- Fix for running the tests where the locale makes ASCII the default encoding.
- Include nbformat-schema files (v3 and v4) in nbformat-schema npm package.
- Include configuration for appveyor's continuous integration service.

4.19 4.2

4.19.1 4.2.0

4.2 on GitHub

- Update nbformat spec version to 4.2, allowing JSON outputs to have any JSONable type, not just `object`, and mime-types of the form `application/anything+json`.
- Define basics of `authors` in notebook metadata. `nb.metadata.authors` shall be a list of objects with the property `name`, a string of each author's full name.
- Update use of traitlets API to require traitlets 4.1.
- Support trusting notebooks on stdin with `cat notebook | jupyter trust`

4.20 4.1

4.20.1 4.1.0

4.1 on GitHub

- Update nbformat spec version to 4.1, adding support for attachments on markdown and raw cells.
- Catch errors opening trust database, falling back on `:memory:` if the database cannot be opened.

4.21 4.0

4.0 on GitHub

The first release of nbformat as its own package.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

n

Index

C

N