nbformat Documentation

Release 4.0

Jupyter Development Team

Mar 29, 2019

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

Some general points about the notebook format:

Note: *All* metadata fields are optional. While the type and values of some metadata are defined, no metadata values are required to be defined.

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" : {
    "signature": "hex-digest", # used for authenticating unsafe outputs on load
    "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]"
    }
  },
  "nbformat": 4,
```

(continues on next page)

(continued from previous page)

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

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" : ["some *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
      "autoscroll": False, # any of true, false or "auto"
  },
  "source" : ["some code"],
```

(continues on next page)

(continued from previous page)

```
  "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-png-data"],
    "application/json": {
      # JSON data is included as-is
      "json": "data",
    },
  },
  "metadata" : {
    "image/png": {
      "width": 640,
      "height": 480,
```

(continues on next page)

(continued from previous page)

```
    },
  },
}
```

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-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 a traceback

```
{
  '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.
```

(continues on next page)

(continued from previous page)

```
  '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 mime-type data"]
}
```

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.

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

1.4.2 Cell metadata

The following metadata keys are defined at the cell level:

Key	Value	Interpretation
collapsed	bool	Whether the cell's output container should be collapsed
autoscroll	bool or 'auto'	Whether the cell's output is scrolled, unscrolled, or autoscrolled
deletable	bool	If False, prevent deletion of the cell
format	'mime/type'	The mime-type of a *Raw NBConvert Cell*
name	str	A name for the cell. Should be unique
tags	list of str	A list of string tags on the cell. Commas are not allowed in a tag

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

Python API for working with notebook files

2.1 Reading and writing

nbformat.**read**(*fp*, *as_version*, ***kwargs*)

Read a notebook from a file as a NotebookNode of the given version.

The string can contain a notebook of any version. The notebook will be returned *as_version*, converting, if necessary.

Notebook format errors will be logged.

Parameters

fp [file or str] A file-like object with a read method that returns unicode (use `io.open()` in Python 2), or a path to a file.

as_version: int The version of the notebook format to return. The notebook will be converted, if necessary. Pass nbformat.NO_CONVERT to prevent conversion.

Returns

nb [NotebookNode] The notebook that was read.

nbformat.**reads**(*s*, *as_version*, ***kwargs*)

Read a notebook from a string and return the NotebookNode object as the given version.

The string can contain a notebook of any version. The notebook will be returned *as_version*, converting, if necessary.

Notebook format errors will be logged.

Parameters

s [unicode] The raw unicode string to read the notebook from.

as_version [int] The version of the notebook format to return. The notebook will be converted, if necessary. Pass nbformat.NO_CONVERT to prevent conversion.

Returns

nb [NotebookNode] The notebook that was read.

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.**write**(*nb*, *fp*, *version=nbformat.NO_CONVERT*, ***kwargs*)
Write a notebook to a file in a given nbformat version.

The file-like object must accept unicode input.

Parameters

nb [NotebookNode] The notebook to write.

fp [file or str] Any file-like object with a write method that accepts unicode, or a path to write a file.

version [int, optional] The nbformat version to write. If nb is not this version, it will be converted. If unspecified, or specified as nbformat.NO_CONVERT, the notebook's own version will be used and no conversion performed.

nbformat.**writes**(*nb*, *version=nbformat.NO_CONVERT*, ***kwargs*)
Write a notebook to a string in a given format in the given nbformat version.

Any notebook format errors will be logged.

Parameters

nb [NotebookNode] The notebook to write.

version [int, optional] The nbformat version to write. If unspecified, or specified as nbformat.NO_CONVERT, the notebook's own version will be used and no conversion performed.

Returns

s [unicode] The notebook as a JSON string.

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.

2.2 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*.

class nbformat.**NotebookNode**(**args*, ***kw*)
A dict-like node with attribute-access

nbformat.**from_dict**(*d*)
Convert dict to dict-like NotebookNode

Recursively converts any dict in the container to a NotebookNode. This does not check that the contents of the dictionary make a valid notebook or part of a notebook.

2.3 Other functions

nbformat.**convert**(*nb*, *to_version*)

Convert a notebook node object to a specific version. Assumes that all the versions starting from 1 to the latest major X are implemented. In other words, there should never be a case where v1 v2 v3 v5 exist without a v4. Also assumes that all conversions can be made in one step increments between major versions and ignores minor revisions.

Parameters

nb [NotebookNode]

to_version [int] Major revision to convert the notebook to. Can either be an upgrade or a downgrade.

nbformat.**validate**(*nbjson*, *ref=None*, *version=None*, *version_minor=None*)

Checks whether the given notebook JSON conforms to the current notebook format schema.

Raises ValidationError if not valid.

class nbformat.**ValidationError**(*message*, *validator=<unset>*, *path=()*, *cause=None*, *context=()*, *validator_value=<unset>*, *instance=<unset>*, *schema=<unset>*, *schema_path=()*, *parent=None*)

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

n

Index