nbtoolbelt Documentation

Release 2024.1.dev0

Tom Verhoeff

Jan 04, 2024

NBTOOLBELT ON COMMAND LINE

Version: 2024.1.dev0

CHAPTER

ONE

INTRODUCTION

When you work with many Jupyter notebooks (e.g., when writing a book or teaching a course), you will need tools to do bulk operations on notebooks.

`nbtoolbelt` is an integrated collection of tools to work with Jupyter notebooks.

We use `nbtoolbelt` together with Momotor, a highly-configurable LTI Tool Provider that automatically processes digital content submitted in our Learning Management System (in our case, students submit Jupyter notebooks), and returns various forms of feedback.

1.1 nbtoolbelt

With `nbtoolbelt` you get tools and libraries to

- **validate** notebooks (a wrapper for `nbformat.validate()`)
- inspect **head** or tail cells of notebooks
- **dump** notebooks compactly on the terminal
- summarize notebooks, with statistics (**stats**)
- **view** notebooks in the browser
- **catenate** multiple notebooks into one notebook
- **clean** notebooks
- **run** notebooks, with pre/post cleaning (a wrapper for `nbconvert.PreProcessor.execute()`)
- **split** notebooks
- **punch** notebooks (shoot holes in them, to produce frameworks useful for exercises)

The next sections briefly discuss some other tools.

1.2 nbconvert

You can use Jupyter's `nbconvert` on the command line:

```
jupyter nbconvert --to notebook --execute --allow-errors notebook.ipynb
```

to **execute** `notebook.nbconvert.ipynb` capturing the execution results, including cells that produce errors. See the `nbconvert` documentation for further details.

1.3 nbdime

You can use `nbdime` for selectively **showing**, **diffing**, and **merging** of notebooks. For instance, when you have installed `nbdime`, the command

```
nbshow -O nb.ipynb
```

will show the notebook without outputs of executed code cells. See the `nbdime` documentation for further details.

CHAPTER

TWO

INSTALLATION

To install `nbtoolbelt` from PyPI (the Python Package Index):

```
pip install nbtoolbelt
```

For users who install from a locally saved *wheel*:

```
pip install --no-index --find-links=<path/to/wheel/dir> nbtoolbelt
```

where `<path/to/wheel/dir>` is the (absolute or relative) path to the directory where you saved the wheel. The wheel has a name of the form `nbtoolbelt-*.whl`.

2.1 Dependencies

`nbtoolbelt` depends on

- `nbformat` for notebook structure, and reading and writing of notebooks;
- `nbconvert` for notebook validation and execution;
- `pandas` and `numpy` for summary statistics in the `stats` tool.

For testing, it depends on `pytest` and `pytest-mock`.

2.2 Upgrading

To upgrade your `nbtoolbelt` installation, add the `--upgrade` flag to the above commands:

```
pip install --upgrade nbtoolbelt

pip install --upgrade --no-index --find-links=<path/to/wheel/dir> nbtoolbelt
```

2.3 Uninstall

To uninstall `nbtoolbelt`:

```
pip uninstall nbtoolbelt
```

2.4 Developers

To install for testing:

```
pip install nbtoolbelt[test]
```

CHAPTER

THREE

USAGE INSTRUCTIONS

3.1 Command-Line Usage

Each tool of `nbtoolbelt` is invoked as a subcommand of `nbtb`:

```
nbtb <tool> ...
```

`nbtb` handles the following arguments itself:

- `nbtb -version` to obtain the current version, and exit
- `nbtb -h` or `nbtb --help` to get a usage message, and exit
- `nbtb config` to show the location of the embedded configuration file and the start-up configuration
- `nbtb config <tool>` to show the start-up configuration for a specific tool

3.1.1 Processing Multiple Notebooks

When more than one notebook is given, or in verbose mode, the output related to each notebook `nb.ipynb` is preceded by a header of the form

```
::::::::::::
nb.ipynb
::::::::::::
```

Some data is collected over all notebooks and aggregated (also see the option `output-json` to *Write JSON Output* and the **Examples** below).

3.1.2 Common Tool Options

Many options, also for the tools themselves, have both a positive and a negated form. This is useful, because option values can also be read from configuration files.

The following options apply to multiple tools (and many of them apply to all tools):

```
-h, --help            show this help message and exit
-v, --verbose, -V, --no-verbose
                      verbose mode produces extra output (default: False)
-q, --quiet, -Q, --no-quiet
```

(continues on next page)

(continued from previous page)

```
                      quiet mode produces less output (default: False)
--assert, --no-assert
                      assert mode: when processing fails, abort with exit
                      code 1 (default: True)
--validate, --no-validate
                      validate notebook before processing (default: False)
--run, --no-run       run notebook before processing (default: False)
--inplace, --no-inplace
                      replace original notebooks with processing result
                      (default: False)
--write-files, --no-write-files
                      do write result files (default: True)
-n                    short for --no-write-files: do processing but do not
                      write result files (dry run)
--config FILE.json    read configuration from FILE.json (in JSON)
--output-json FILE.json
                      write statistical output to FILE.json
-d, --debug           debug mode produces diagnostic output (default: False)
```

Help

Each command provides its own help option:

- `nbtb <tool> -h` or `nbtb command --help`

The default option values shown, are after loading of the start-up configuration (see below).

Verbose Mode

Each command supprts verbose mode:

- `nbtb <tool> -v` or `nbtb <tool> --verbose`

In verbose mode, the tool's option values are shown, and the total number of processed notebooks is shown at the end, e.g.:

```
Notebooks processed: 2
```

Quiet Mode

Each command supprts quiet mode:

- `nbtb <tool> -q` or `nbtb <tool> --quiet`

In quiet mode, output is reduced to a minimum.

Assert Mode

In assert mode (option `-a` or `--assert`), whenever an error occurs during processing, all further processing will be interrupted and the tool aborts.

Validate Mode

In validate mode (option `--validate`), each notebook is validated before processing.

Validation options can be set under the specific tool in a configuration file, but not on the command line.

Run Mode

In run mode (option `--run`), each notebook is run before processing (and after validation, if that applies).

Run options can be set under the specific tool in a configuration file, but not on the command line.

Inplace

For some tools (currently: `clean` and `run`), the option `--inplace` can be used to *replace* the input notebook with the resulting notebook.

Write Files and Dry Run

In dry-run mode (option `--no-write-files`, or `-n` for short), all processing and reporting is done, but resulting notebooks are not written to their destination file.

This allows one te experiment with options. In particular, this is useful in combination with verbose mode.

Read Configuration File

Initial option settings can be read from a configuration file (option `--config FILE.json`).

At start up, each tool gathers an initial configuration as follows.

1. The configuration file embedded in the package is loaded.
2. If present, the configuration file `/etc/nbtoolbelt.json` is loaded on top of this.
3. Finally, if present, the configuration file `~/.nbtoolbelt.json` is loaded on top of that. Here, `~` stands for the user's home directory.

The invokation

```
$ nbtb config [tool]
```

will show which configuration files are loaded, and what the resulting configuration is.

Each tool can read options from a configuration file:

- `nbtb <tool> ... --config file.json ...`

The options from the configuration file override the preceding command-line options, and are, in turn, overridden by any subsequent command-line options.

See *Configuration Files* for details about configuration files.

Write JSON Output

Most tools gather some statistics as they process notebooks. These statistics are printed as notebooks are processed. For use with other tooling, such tool output can also be written to a file in JSON format (option `--output-json FILE.json`).

This JSON output file contains one *array*, where

- item 0 concerns data aggregated over all notebooks that were processed;
- item *i* (*i* from 1) concerns data collected for the *i*-th notebook argument;
- each item is an object with further data appearing in various members, documented per tool;
- one common member is present with name `"file_name"`, and the file's name as value.

Debug Mode

In debug mode, diagnostic output is produced, and in particular, all option settings are shown.

3.1.3 Available Tools

The following tools are available:

- `nbtb validate ...` *Notebook Validation*
- `nbtb head ...` *Notebook Heads*
- `nbtb dump ...` *Notebook Dump*
- `nbtb stats ...` *Notebook Statistics*
- `nbtb view ...` *Notebook Viewing*
- `nbtb cat ...` *Notebook Catenation*
- `nbtb clean ...` *Notebook Cleaning*
- `nbtb run ...` *Notebook Execution*
- `nbtb split ...` *Notebook Splitting*
- `nbtb punch ...` *Notebook Punching*

3.1.4 Examples

The following invocation shows

- the options loaded at start up from the various configuration files,
- the location where these files were found, and
- what the resulting configuration is:

```
$ nbtb config run
Loading configuration for nbrun from: /Users/.../nbtoolbelt/data/nbtoolbelt.
↪json
Skipping configuration in: /etc/nbtoolbelt.json (not present)
Loading configuration for nbrun from: /Users/.../.nbtoolbelt.json
Configuration accumulated for nbrun:
  allow_errors (bool): True
  append_cell (bool): False
  appended_cell (str): '# Automatically added code cell: lists all global␣
↪names defined by this notebook.\n%whos'
  assert (bool): True
  clean_after (bool): True
  clean_after_metadata (list): ['collapsed', 'scrolled']
  clean_before (bool): True
  clean_before_metadata (list): ['ExecuteTime', 'execution']
  debug (bool): False
  inplace (bool): False
  interrupt_on_timeout (bool): True
  ipc (str): ''
  kernel_name (str): ''
  notebooks (list): []
  output_json (NoneType): None
  quiet (bool): False
  record_timing (bool): True
  run (bool): False
  run_path (str): ''
  run_result_name (str): '-run'
  streams_head (int): -1
  streams_truncate_message (str): '*** Output truncated ***'
  timeout (int): -1
  validate (bool): False
  verbose (bool): False
  write_files (bool): True
```

The following invocation prints statistics for two notebooks, and writes JSON output to file `nbtb-stats-output.json`.

```
$ nbtb stats short.ipynb test.ipynb --output-json nbtb-stats-output.json

::::::::::::::
short.ipynb
::::::::::::::
```

(continues on next page)

(continued from previous page)

```
Notebook metadata:
         4.2 format version
     python3 kernel
  python 3.6.1 language
Cell types:
           1 code
           1 markdown
           2 total cell count

::::::::::::::
test.ipynb
::::::::::::::
Notebook metadata:
         4.2 format version
     python3 kernel
  python 3.6.1 language
Cell types:
          12 code
           7 markdown
          19 total cell count

Totals
======

Cell types:
          13 code
           8 markdown
          21 total cell count
```

The file `nbtb-stats-output.json` then contains:


```
[
  {
    "cell_types": {
      "code": {
        "25%": 3.75,
        "50%": 6.5,
        "75%": 9.25,
        "count": 2.0,
        "max": 12.0,
        "mean": 6.5,
        "min": 1.0,
        "std": 7.7781745930520225,
        "total": 13.0
      },
      "markdown": {
        "25%": 2.5,
        "50%": 4.0,
        "75%": 5.5,
        "count": 2.0,
```


(continues on next page)

(continued from previous page)


```
        "max": 7.0,
        "mean": 4.0,
        "min": 1.0,
        "std": 4.242640687119285,
        "total": 8.0
      },
      "total cell count": {
        "25%": 6.25,
        "50%": 10.5,
        "75%": 14.75,
        "count": 2.0,
        "max": 19.0,
        "mean": 10.5,
        "min": 2.0,
        "std": 12.020815280171307,
        "total": 21.0
      }
    },
    "file_count": 2,
    "file_name": {
      "count": 2,
      "freq": 1,
      "top": "short.ipynb",
      "unique": 2
    },
    "notebook_metadata": {
      "format version": {
        "count": 2,
        "freq": 2,
        "top": "4.2",
        "unique": 1
      },
      "kernel": {
        "count": 2,
        "freq": 2,
        "top": "python3",
        "unique": 1
      },
      "language": {
        "count": 2,
        "freq": 2,
        "top": "python 3.6.1",
        "unique": 1
      }
    }
  },
  {
    "cell_types": {
      "code": 1,
```


(continues on next page)

(continued from previous page)


```
      "markdown": 1,
      "total cell count": 2
    },
    "file_name": "short.ipynb",
    "notebook_metadata": {
      "format version": "4.2",
      "kernel": "python3",
      "language": "python 3.6.1"
    }
  },
  {
    "cell_types": {
      "code": 12,
      "markdown": 7,
      "total cell count": 19
    },
    "file_name": "test.ipynb",
    "notebook_metadata": {
      "format version": "4.2",
      "kernel": "python3",
      "language": "python 3.6.1"
    }
  }
]
```


3.2 Notebook Validation

`validate` validates notebooks, using `nbformat.validate()`. This takes the notebook version into account.

When the notebook is invalid, a message explains the problem.

Only in verbose mode, will `validate` report explicitly that a notebook is valid. Otherwise, it will quietly continue.

3.2.1 Options

`validate` does not take any special options.

3.2.2 Examples

Check two notebooks, one valid and the other not, with verbose mode:

```
$ nbtb validate test.ipynb test-bad.ipynb -v

  ::::::::::::::
  test.ipynb
  ::::::::::::::
  test.ipynb notebook structure is valid

  ::::::::::::::
  test-bad.ipynb
  ::::::::::::::
  test-bad.ipynb notebook structure is INVALID

  Additional properties are not allowed ('BAD_src' was unexpected)

  Failed validating 'additionalProperties' in markdown_cell:

  On instance['cells'][1]:
  {'BAD_src': ['This is a notebook for testing purposes;\n',
               'in particular, for testing the various scripts.\n',
               '\n',
               'It has a few formatted _MarkDown_ cells.\n',
               'And some simple code cells, with various execution effects;\n
↪',
               'some (not only the last) has an error.\n',
               'Various tags have been set.'],
   'cell_type': 'markdown',
   'metadata': {}}

  Files processed: 2
```

3.3 Notebook Heads

`head` shows the first/last *n* lines of the first/last cell in Jupyter notebooks.

`head` is also a good tool to see how you can extend `nbtoolbelt` with another tool. Also see *Adding Tools to nbtoolbelt*.

3.3.1 Options

The following options are supported by `head`:

```
-# NUMBER, --number NUMBER
                      number of head/tail lines to show of first/last cell;
                      < 0 for tail (default: 5)
```

Number

By default, `head` shows the first 5 lines of the first cell (or fewer, if the cell is shorter). The `number` option determines the desired number of lines, and whether this is from the head of the first cell (when the number is positive), or from the tail of the last cell (when the number is negative). The actual number of lines shown is smaller, when the cell has fewer lines than desired.

3.3.2 Examples

Show head of notebook `test.ipynb`:

```
$ nbtb head test.ipynb
# Test Notebook

This is a notebook for testing purposes;
in particular,
for testing the various scripts.
```

Show 10 tail lines in last cell of notebook `test.ipynb`:

```
$ nbtb head -# -10 test.ipynb -v
Options for nbhead:
  Show last 10 source lines of last cell
::::::::::::::
test.ipynb
::::::::::::::
Last
cell
with
more
than
five
lines.

Notebooks processed: 1
```

The last cell apparently contains fewer than 10 lines.

3.4 Notebook Dump

`dump` shows notebook and cell information, and cell sources of Jupyter notebooks in a compact way on stdout (more compact than `nbshow` of the `nbdime` package).

Which elements of a notebook are dumped can be selected through options. The output format can also be adjusted via options.

3.4.1 Options

The following options are supported by `dump`:

```
-g, --notebook-info, -G, --no-notebook-info
                      show global notebook information (default: True)
-c, --cell-info, -C, --no-cell-info
                      show cell information (default: True)
-s, --sources, -S, --no-sources
                      show cell sources (default: True)
-t TYPES, --cell-types TYPES
                      comma-separated list of cell types to dump (default:
                      'markdown,code,raw')
-p STRING, --prefix STRING
                      pre/postfix for information lines (default: '=====')
-i INT, --indent INT  indentation level for source lines (default: 2)
-l, --line-numbers, -L, --no-line-numbers
                      show source line numbers (default: False)
-e INT, --cell-spacing INT
                      number of empty lines between cells (default: 0)
```

3.4.2 Examples

A dump of notebook `short.ipynb` in verbose mode:

```
$ nbtb dump -v short.ipynb
Options for nbdump:
  Dump notebook info: True
  Dump cell info: True
  Dump cell sources: True
  Dump cell types: markdown,code,raw
  Prefix for information lines: =====
  Source line indentation level: 2
  Show source line numbers: False
  Cell spacing lines: 0
::::::::::::::
short.ipynb
::::::::::::::
===== nbformat: 4.2 | kernel: python3 | language: python 3.6.1 =====
===== cells: 2 =====
===== cell 0: markdown | metadata: collapsed =====
```

(continues on next page)

(continued from previous page)

```
  # Short Notebook for Testing

  * **Bold**
  * _Italic_
  * `Typewriter`
  * $e^{\pi i} + 1 = 0$
===== cell 1: code | outputs: 2 =====
  print(6 * 7)
  1 + 1

Notebooks processed: 1
```

A dump that shows notebook and cell information only (no sources):

```
$ nbtb dump -S test.ipynb
===== nbformat: 4.2 | kernel: python3 | language: python 3.6.1 =====
===== cells: 19 | metadata: celltoolbar,toc =====
===== cell 0: markdown =====
===== cell 1: markdown =====
===== cell 2: markdown =====
===== cell 3: code | metadata: collapsed | outputs: 1 =====
===== cell 4: code | metadata: collapsed | outputs: 1 =====
===== cell 5: code | metadata: collapsed | outputs: 1 =====
===== cell 6: code | outputs: 1 =====
===== cell 7: code | outputs: 2 =====
===== cell 8: code | outputs: 2 =====
===== cell 9: markdown | tags: YourTurn =====
===== cell 10: code | metadata: scrolled | outputs: 1 =====
===== cell 11: code | metadata: collapsed =====
===== cell 12: code | metadata: scrolled | tags: YourTurn | outputs: 1 =====
===== cell 13: code | metadata: collapsed =====
===== cell 14: markdown =====
===== cell 15: code | tags: YourTurn,Test | outputs: 2 =====
===== cell 16: code | metadata: collapsed =====
===== cell 17: markdown | attachments: 1 =====
===== cell 18: markdown =====
```

A dump of MarkDown sources only, without indentation:

```
$ nbtb dump -G -C -t markdown -i 0 test.ipynb
# Test Notebook

This is a notebook for testing purposes;
in particular,
for testing the various scripts.

It has a few formatted _MarkDown_ cells.
And some simple code cells, with various execution effects;
some (not only the last) has an error.
Various tags have been set.
```

(continues on next page)

(continued from previous page)

```
## A Section

Demonstrating **bold face**.
This one has a $\LaTeX$ formula: $e^{\pi i} + 1 = 0$.
Print the first one hundred numbers, their squares, and their cubes.
Produce a line plot of `df`.
![jupyter.png](attachment:jupyter.png)
Last
cell
with
more
than
five
lines.
```

A dump of code cells without global notebook information, without indentation, with cell spacing 1, and info prefix `#####`:

```
$ nbtb dump -G -t code -i 0 -l 1 -p '#####' test.ipynb
##### cell 3: code | metadata: collapsed | outputs: 1 #####
print([i ** 3 for i in range(6)])  # output as stream in stdout

##### cell 4: code | metadata: collapsed | outputs: 1 #####
# NameError
undefined

##### cell 5: code | metadata: collapsed | outputs: 1 #####
6 * 7  # output as execution result

##### cell 6: code | outputs: 1 #####
6 / 0  # ZeroDivisionError

##### cell 7: code | outputs: 2 #####
# stdout and execution result
print('Hello, World!')
'Thanks'

##### cell 8: code | outputs: 2 #####
import sys
print('Message on stdout')
print('Message on stderr', file=sys.stderr)

##### cell 10: code | metadata: scrolled | outputs: 1 #####
for i in range(100):  # output as stream in stdout
    print(i, i ** 2, i ** 3)

##### cell 11: code | metadata: collapsed #####
import pandas as pd

##### cell 12: code | metadata: scrolled | tags: YourTurn | outputs: 1 #####
```

(continues on next page)

(continued from previous page)

```
df = pd.DataFrame([3, 1, 4, 1, 5, 9])
df

##### cell 13: code | metadata: collapsed #####
# import seaborn  # commented out to avoid depencency on seaborn
# %matplotlib inline  # commented out to avoid dependcy on matplotlib

##### cell 15: code | tags: YourTurn,Test | outputs: 2 #####
# df.plot(kind='line')  # commented out to avoid dependency on matplotlib

##### cell 16: code | metadata: collapsed #####
print(42)  # output to be deleted afterwards
```

A dump of a notebook with source line numbers:

```
$ nbtb dump -l short.ipynb
===== nbformat: 4.2 | kernel: python3 | language: python 3.6.1 =====
===== cells: 2 =====
===== cell 0: markdown | metadata: collapsed =====
   1| # Short Notebook for Testing
   2|
   3| * **Bold**
   4| * _Italic_
   5| * `Typewriter`
   6| * $e^{\pi i} + 1 = 0$
===== cell 1: code | outputs: 2 =====
   1| print(6 * 7)
   2| 1 + 1
```

3.5 Notebook Statistics

`stats` can show the following statistics:

- summary of required notebook metadata (always shown)
- list of other notebook metadata fields
- list of additional notebook fields (outside metadata; should normally not be present)
- cell statistics:
  - count per cell type (`markdown`, `code`, `raw`), and their total
  - size statistics for cell sources for all cells, and for markdown, code, and raw cells separately:
    - number of empty cells
    - total number of lines (non-empty), words, characters (non-whitespace).
  - count per cell metadata field, and
    - count per tag in `tags` metadata
  - count attachments, and

        * count per attachment MIME type
    - count outputs for code cells (note that there can be multiple outputs per code cell)
        * count of code cells without output
        * count per output type (`execute_result`, `stream`, `display_data`, `error`), and their total
        * count per `stream` (`stdout`, `stderr`)
        * count per `error` (by `ename`)
    - execution counts for code cells:
        * count executed
        * count executed *in linear order*
        * maximum execution count (max # in In[#])
        * count not executed
        * count not executed *in linear order*
    - list of additional cell fields (outside metadata; should normally not be present)

Use options to select reporting of specific statistics.

3.5.1 Options

The following options are supported by `stats`:

```
--all, --no-all       show all statistics (default: False)
-c, --cell-types, -C, --no-cell-types
                      count cell types (default: True)
-s, --sources, -S, --no-sources
                      statistics for cell sources (default: False)
-m, --metadata, -M, --no-metadata
                      show notebook metadata and count cell metadata
                      (default: False)
-t, --tags, -T, --no-tags
                      count individual cell tags (default: False)
-a, --attachments, -A, --no-attachments
                      count cell attachment MIME types (default: False)
-o, --outputs, -O, --no-outputs
                      count code cell outputs (default: False)
--streams, --no-streams
                      count code cell output stream names (default: False)
-e, --errors, -E, --no-errors
                      count code cell error names (default: False)
-x, --execution, -X, --no-execution
                      statistics for code execution (default: False)
--extra, --no-extra   report extra fields outside metadata (default: False)
```

3.5.2 JSON Output

See *Write JSON Output* for general information about JSON output.

`stats` produces the following members in the JSON output, where (*) refers to details below:

Name	Value
`"notebook_metadata"`	object with required notebook metadata
`"notebook_other_meta`	object with 1 for each other notebook metadata field
`"notebook_extra_fiel`	object with 1 for each extra notebook fields
`"cell_types"`	object with count per cell type (*)
`"sources"`	object with empty/line/word/char/total source counts per cell type and totals (*)
`"cell_metadata"`	object with counts of cell metadata fields
`"cell_attachments"`	object counts of cells attachments per type and totals
`"code_execution"`	object with counts of for cell execution (*)
`"code_outputs"`	object with counts of code cell output per type and totals (*)
`"cell_extra"`	object with count for each extra cell field

Details for `"cell_types"`:

Name	Value
`"markdown"`	count of markdown cells
`"code"`	count of code cells
`"raw"`	count of raw cells
`"total cell count"`	count of all cells

Details for `"sources"`:

Name	Value
`"CT source W"`	count of `W` for cell type `CT`
`"CT empty sources"`	count of empty sources for cell type `CT`

Where cell type `CT` is on of `markdown`, `code`, `raw`, or `total`, and what `W` is one of `chars`, `lines`, `words`.

Details for `"code_execution"`:

Name	Value
`"executed"`	count of executed code cells
`"executed in linear order"`	count of code cells executed in linear order from beginning
`"maximum In[#]"`	maximum execution code of executed code cells
`"not executed"`	count of code cells not executed
`"not executed in linear order"`	count of code cells not executed in linear order from beginning

Details for "code_outputs":

Name	Value
"empty outputs"	count of code cells without output
"display_data"	count of display_data output
"error"	count of error output (total)
"error E"	count of error output with exception E
"execute_result"	count of code cells with an execution result as output
"stream"	count of code cells with stream output (total)
"stream S"	count of code cells with stream S output
"total output count"	count of all output items (total over all code cells)

Note that each code cell can have zero or more output items in its outputs array.

3.5.3 Examples

Report required notebook metadata and cell types for two notebooks:

```
$ nbtb stats short.ipynb test.ipynb --output-json nbtb-stats-output.json

::::::::::::::
short.ipynb
::::::::::::::
Notebook metadata:
         4.2 format version
     python3 kernel
  python 3.6.1 language
Cell types:
           1 code
           1 markdown
           2 total cell count

::::::::::::::
test.ipynb
::::::::::::::
Notebook metadata:
         4.2 format version
     python3 kernel
  python 3.6.1 language
Cell types:
          12 code
           7 markdown
          19 total cell count

Totals
======

Cell types:
          13 code
```

(continues on next page)

(continued from previous page)

```
           8 markdown
          21 total cell count
```

All statistics for one notebook:

```
$ nbtb stats --all test.ipynb
Notebook metadata:
         4.2 format version
     python3 kernel
  python 3.6.1 language
Other notebook metadata fields:
           1 celltoolbar
           1 toc
Cell types:
          12 code
           7 markdown
          19 total cell count
Cell sources:
         561 code source chars
          20 code source lines
         115 code source words
         454 markdown source chars
          21 markdown source lines
          89 markdown source words
        1015 total source chars
          41 total source lines
         204 total source words
Cell metadata fields:
           6 collapsed
           2 scrolled
           1 tag Test
           3 tag YourTurn
           3 tags
Cell attachments:
           1 image/png
           1 total attachments count
           1 total count of cells with attachments
Code cell outputs:
           3 code cells without outputs
           1 display_data
           2 error
           1 error NameError
           1 error ZeroDivisionError
           4 execute_result
           5 stream
           1 stream stderr
           4 stream stdout
          12 total output count
Code cell execution:
          11 executed
```

(continues on next page)

(continued from previous page)

```
  3 executed in linear order
 15 maximum In[#]
  1 not executed
  9 not executed in linear order
```

3.6 Notebook Viewing

`view` generates HTML from notebooks, and opens them in your browswer. It uses the `nbconvert.HTMLExporter` and in-lines *embedded attachments* to make them visible.

3.6.1 Result File Name

The resulting HTML for viewing notebook `nb.ipynb` is written to `nb-view.html`, unless the `--inplace` options is used, in which case it is written to `nb.html`.

The result name addition can be adjusted in the configuration file by setting `view_result_name`; see *Configuration Files*.

3.6.2 Options

The following options are supported by `view`:

```
-t NAME, --template NAME
                      template to use (default: classic)
-b, --browser, -B, --no-browser
                      open html in browser (default: True)
-w, --wait_delete, -W, --no-wait_delete
                      interactively wait for user confirmation to delete
                      html files (default: False)
```

Template File

The option `template-file` is deprecated, because `nbconvert` 6 has a template mechanism that uses directories instead of single template files.

Template

By default, `view` uses the `classic` template.

The `template` option can be used to select another template. For instance, `nbconvert` supports the `basic` template, without any CSS and JavaScript, and `lab` to give you a JypterLab look. Also see nbconvert documentation for HTML output.

Note that Jupyter notebook extensions provide additional templates, such as `toc2.tpl` for an enhanced Table of Contents. However, these templates can no longer be used, since they have not been converted to the `nbconvert` 6 style.

Browswer

By default, `view` opens the generated HTML in a new tab in your default browser.

The `browser` option can be used to override this behavior.

Wait-Delete

By default, `view` exits after generating HTML files, and opening them in your browser. The generated HTML files are not deleted.

The `wait-delete` option makes `view` wait after generating the HTML files, until the user confirms deletion of the HTML files:

```
$ nbtb view -w test.ipynb
Delete created html files ([y]/n)? y
Deleting: test-view.html
```

3.7 Notebook Catenation

`cat` catenates the cells of multiple notebooks into a single notebook. The resulting notebook takes its notebook-level properties from the first notebook.

This tool can be used to prepend and append cells to a notebook (also see the `append-cell` option of the `run` tool). Prepended cells can provide (re)definitions, and appended cells can show further output.

3.7.1 Educational Use

In an educational setting, it can be useful to redefine the built-in function `input()` to either be blocked:

```
def input(*args, **kwargs):
    import sys
    print('input() is forbidden in notebooks', file=sys.stderr)
    sys.exit(1)
```

Or to deliver strings read from a file:

```
def __yield_from_file(file_name):
    with open(file_name) as f:
        yield from f.readlines()

def input(prompt=None, __gen=__yield_from_file('input.txt')):
    if prompt:
        print(prompt, end='')
    line = next(__gen)
    # suppress terminating end-of-line
    if line and line[-1] == '\n':
        return line[:-1]
    return line
```

By putting this code in a notebook, it can be prepended to every student notebook, via `cat`.

3.7.2 Result File Name

The result of catenating notebooks `nb.ipynb ...` is written to `nb-cat.ipynb` (that is, the name of the first notebook argument is used), unless the option `--inplace` is applied, in which case the result is written to `nb.ipynb`.

The result name addition can be adjusted in the configuration file by setting `cat_result_name`; see *Configuration Files*.

3.7.3 Options

`cat` does not have any tool-specific options.

3.7.4 JSON Output

See *Write JSON Output* for general information about JSON output.

`cat` produces the following members in the JSON output:

Name	Value
`"cell_types"`	object with counts per cell type (see *Notebook Statistics*)

Note that members are absent when count is zero.

3.8 Notebook Cleaning

`clean` can delete elements from Jupyter notebooks:

- selected notebook-level non-required metadata fields
- selected cell-level metadata fields
- selected cell tags
- empty cells
- all outputs of code cells

When code cell outputs are cleaned, also the execution counts are cleaned (reset to not-executed).

3.8.1 Result File Name

The result of cleaning notebook `nb.ipynb` is written to `nb-clean.ipynb`, unless the option `--inplace` is applied.

The result name addition can be adjusted in the configuration file by setting `clean_result_name`; see *Configuration Files*.

3.8.2 Options

The default behavior is to clean nothing. The following options are supported by `clean`:

```
-g FIELDS, --global FIELDS
                      comma-separated list of fields to remove from metadata
                      of notebook (default: '')
-m FIELDS, --metadata FIELDS
                      comma-separated list of fields to remove from metadata
                      of all cells (default: '')
-t TAGS, --tags TAGS  comma-separated list of tags to remove from all cells;
                      use '-m tags' to remove all tags (default: '')
-e, --empty-cells, -E, --no-empty-cells
                      delete cells with empty source, i.e. with whitespace
                      only (default: False)
-o, --outputs, -O, --no-outputs
                      clean all outputs from all code cells (default: False)
```

3.8.3 JSON Output

See *Write JSON Output* for general information about JSON output.

`clean` produces the following members in the JSON output:

Name	Value
`"global F"`	count of global metadata fields `F` cleaned (0 or 1)
`"cell F"`	count of cell metadata fields `F` cleaned
`"tag T"`	count of cell tags `T` cleaned
`"empty_cells"`	count of empty cells cleaned
`"outputs"`	count of cell outputs cleaned

Note that members are absent when count is zero.

3.8.4 Examples

Remove all `collapsed` and `scrolled` metadata fields from cells:

```
$ nbtb clean -m collapsed,scrolled test.ipynb -v
Options for nbclean:
  Deleting cell metadata fields: ['collapsed', 'scrolled']
  Clearing outputs from all code cells

::::::::::::::
test.ipynb
::::::::::::::
Counts:
           6 cell collapsed
           2 cell scrolled
```

(continues on next page)

(continued from previous page)

```
          9 outputs cleaned
  Files written: {'test-clean.ipynb'}

Notebooks processed: 1
```

3.9 Notebook Execution

`run` executes all codes cells in the notebook, from first to last, comparable to `Kernel > Restart & Run All` in the *Jupyter Notebook Editor*. Each cell is executed separately, in the context of the global state, which is empty at the start and updated as cells execute.

Unless in quiet mode, `run` will report some execution statistics on `stdout`:

- number of code cells
- number of executed code cells
- number of executions that resulted in an error

3.9.1 Result File Name

The result of running notebook `nb.ipynb` is written to `nb-run.ipynb`, unless the option `--inplace` is applied.

The result name addition can be adjusted in the configuration file by setting `run_result_name`; see *Configuration Files*.

3.9.2 Options

The following options are supported by `run`:

```
-e, --allow-errors, -E, --no-allow-errors
                      continue on errors (default: True)
-b, --clean-before, -B, --no-clean-before
                      clean code output before running (default: True
-a, --clean-after, -A, --no-clean-after
                      clean code metadata after running (default: True)
-k KERNEL_NAME, --kernel-name KERNEL_NAME
                      name of kernel to use, e.g. python3; for kernel
                      specified in notebook; default: '')
-p RUN_PATH, --run-path RUN_PATH
                      working directory for execution ;"" for current
                      working directory (default: '')
-t TIMEOUT, --timeout TIMEOUT
                      timeout time in seconds per cell; -1 for unlimited
                      (default: -1)
-i, --interrupt-on-timeout, -I, --no-interrupt-on-timeout
                      interrupt execution on timeout (default: False)
-r, --record-timing, -R, --no-record-timing
```

(continues on next page)

(continued from previous page)

```
                      record execution times (default: True)
-w, --append-cell, -W, --no-append-cell
                      append special code cell (default: '# Automatically
                      added code cell: lists all global names defined by
                      this notebook.\n%whos') before executing notebooks
                      (default: False)
--streams-head STREAMS_HEAD
                      limit output streams to a maximum number of lines per
                      cell; -1 for unlimited (default: 1000)
```

Dry Run

By default, the result of executing notebook `nb.ipynb` is written to `nb-run.ipynb`, unless the `dry-run` option is in effect.

The `dry-run` option suppresses writing of the resulting notebook. It does not suppress execution.

Allow Errors

By default, `run` executes *all* code cells, even when errors occur. Such errors typically give rise to an `error` element among the cell outputs in the notebook (which the *Jupyter Notebook Editor* shows on a red background).

The `no-allow-errors` option makes `run` abort execution at the first error. Error details are then reported on `stderr`.

Timeout

By default, `run` does not limit the execution time. The `timeout` option sets a limit in seconds on the execution time *per cell*. Use `-1` as value to remove the limit.

Interrupt on Timeout

By default, `run` interrupts the kernel with a `KeyboardInterrupt` when a timeout occurs (which does *not* stop further execution when allowing errors). When *not* interrupting on timeout, `run` aborts execution with a `CellTimeoutError` when a timeout occurs (which stops further execution even when allowing errors).

Record Timing

By default, `run` records cell execution times in the resulting cell metadata under the key `execution`. See nbformat, Cell metadata for details.

Kernel Name

By default, `run` uses the kernel associated with the notebook (as specified in its `metadata > kernel` attribute). The `kernel_name` option overrides this default behavior and sets a specific kernel. Typical choices are

- `python2`
- `python3`

Run Path

By default, the notebook is executed in the current working directory where the tool was invoked. The `run-path` option sets a specific path to be the working directory.

Clean Before

By default, `run` cleans the notebook before execution:

- the outputs of all code cells are removed,
- their execution counts are cleared, and
- `ExecuteTime` and `execution` metadata is removed.

This ensures that the the notebook is run in a clean environment. In particular, when executing until the first error, no spurious outputs and metadata from a previous run will be left in the notebook.

The `not-before` option suppresses this pre-cleaning step.

Clean After

By default, `run` cleans the notebook after execution:

- the `collapased` and `scrolled` metadata are removed from all code cells

Currently, it appears that execution through `nbconvert` will make all output cells collapsed and scrolled. When such a notebook is subsequently opened in the *Jupyter Notebook Editor*, no outputs will be visible, until they are actively opened by the user (either by clicking to the left of the output area of a cell, or through the `Cell` menu to *toggle* the output cell state.

The `not-after` option suppresses this post-cleaning step.

Append Cell

The `append-cell` option appends to the notebook (before execution) a code cell containing:

```
# Automatically added code cell: lists all global names defined by this␣
↪notebook.
%whos
```

This will show an overview of all names defined a the global level of the notebook, their types, and some data/info. Also see the example below.

The content of the appended code cell can be changed in a configuration file.

Using the `cat` tool, you can prepend and append entire notebooks.

Streams Head

The `streams-head` option will limit the amount of data in output streams (both *stdout* and *stderr*) that is written to the resulting notebook, by showing only the *head*. The number of lines to show is a parameter to this option (use -1 for unlimited output). It applies per cell.

This option serves as a protection to generate excessively large notebooks, when the input notebook contains 'overzealous' code. An example of 'overzealous' code is an infinite loop that contains a call to *print*.

3.9.3 JSON Output

See *Write JSON Output* for general information about JSON output.

`run` produces the following members in the JSON output:

Name	Value
`"code cells"`	count of code cells
`"executed"`	count of executed code cells
`"with errors"`	count of executed code cells with errors

Note that members are absent when count is zero.

3.9.4 Examples

Example output when running a single notebook without options:

```
$ nbtb run test.ipynb
Execution statistics:
     12 code cells
     12 executed
      2 with errors
```

The execution result is written to `test-run.ipynb`.

Example output when running to first error in verbose mode without writing the result:

```
$ nbtb run -E tool-faq.ipynb test.ipynb -v -n
Dry run (no files written)
Options for nbrun:
  Cleaning code output and ['ExecuteTime', 'execution'] metadata before␣
↪running
  Timeout: unlimited
  Kernel name: from notebook
  Working dir: current directory
  Stopping at first error
```

(continues on next page)

(continued from previous page)

```
  Cell execution does not time out
  Recording execution times
  Not appending code cell with %whos
  Cleaning ['collapsed', 'scrolled'] code metadata after running
  Not limiting output streams

::::::::::::::
short.ipynb
::::::::::::::
Execution statistics:
           1 code cells
           0 code cells with empty source
           1 executed
           0 with errors
No file written

::::::::::::::
test.ipynb
::::::::::::::

Notebooks processed: 1
Processing of "test.ipynb" failed (CellExecutionError): An error occurred␣
↪while executing the following cell:
------------------
# NameError
undefined
------------------

NameError: name 'undefined' is not defined

Tool 'run' aborted.
```

Here is an example with the `append-cell` (`-w`) option. The execution result is written to `test-run.ipynb`. Next, we use `nbshow` to inspect the output of the last cell (Note that `nbshow` is part of the `nbdime` package; see documentation of `nbdime`.)

```
$ nbtb run -w test.ipynb
Execution statistics:
          13 code cells
           0 code cells with empty source
          13 executed
           2 with errors

$ nbshow test-run.ipynb | tail -n -16
code cell 19:
  execution_count: 13
  source:
    # Automatically added code cell: lists all global names defined by this␣
↪notebook.
    %whos
```

(continues on next page)

(continued from previous page)

```
  outputs:
    output 0:
      output_type: stream
      name: stdout
      text:
        Variable   Type         Data/Info
        ----------------------------------
        df         DataFrame       0\n0  3\n1  1\n2  4\n3  1\n4  5\n5  9
        i          int          99
        pd         module       <module 'pandas' from '/o<...>ages/pandas/__
↪init__.py'>
        sys        module       <module 'sys' (built-in)>
```

3.10 Notebook Splitting

`split` splits Jupyter notebooks by writing each cell type (markdown, code, and raw) to its own notebook.

Unless in quiet mode, `split` shows cell counts: markdown, code, raw, and total.

3.10.1 Result File Names

The results of splitting notebook `nb.ipynb` are named:

- `nb-markdown.ipynb`
- `nb-code.ipynb`
- `nb-raw.ipynb`

When a particular type of cell does not occur, that result notebook is not written.

The result name additions can be adjusted in the configuration file by setting `split_markdown_result_name`, `split_code_result_name`, and `split_raw_result_name`; see *Configuration Files*.

3.10.2 Options

The following options are supported by `split`:

```
-t TYPES, --split-cell-types TYPES
                      comma-separated list of cell types to split (default:
                      'markdown,code,raw')
```

3.10.3 JSON Output

See *Write JSON Output* for general information about JSON output.

`split` produces the following members in the JSON output:

Name	Value
"CT"	count of cells with type CT

Cell types distinguished: `markdown`, `code`, `raw`

3.10.4 Examples

Split notebook `test.ipynb` in verbose mode without writing the results:

```
$ nbtb split test.ipynb -v -n
Dry run (no files written)
Options for nbsplit:
  (None)

::::::::::::::
test.ipynb
::::::::::::::
Split cell statistics:
          12 code
           7 markdown
           0 raw
          19 total
No files written

Notebooks processed: 1
```

3.11 Notebook Punching

`punch` punches holes in notebooks, supporting two different approaches to defining the holes (see below). The resulting holes can be **filled** either with fixed content, or from a source notebook (see options `fill` and `source`).

`punch` can also collect the content that has been punched out, the so-called **chads** (see option `chads`).

3.11.1 Result File Names

The result of punching notebook `nb.ipynb` is written to `nb-punched.ipynb`, and the content that got punched out, the so-called *chads*, is written to `nb-chads.ipynb`.

The result name additions can be adjusted in the configuration file by setting `punch_punched_result_name` and `punch_chads_result_name`; see *Configuration Files*.

3.11.2 Holes and Filling

The two ways of defining holes are:

- Via **tags**: any cell with a tag contained in the trigger set (see option `tags`), will be punched as a whole.
- Via **marker lines**: any source content appearing between *begin* and *end* marker lines will be punched.

Via Tags

This approach is selected by providing a list of tags to the `tags` option.

In the tags approach, `punch` removes the source content of selected cells. No cells will be removed; only their source content will be emptied.

Cells are selected through the presence of cell metadata tags in a trigger set, defined with the option `tags`.

When the source of a *code* cell is cleared, also its outputs and execution count are cleared.

Note that `nbgrader` uses the tag `YourTurn` to mark cells where students write their solutions; see documentation of `nbgrader`.

Via Marker Lines

This approach applies when no tags are given (the default).

Lines to be punched are marked by enclosing them between **begin** and **end** marker lines.

Such marker lines are recognized by a special character sequence (default: `#//` for all cell types; this can be overridden per cell type in the configuration file). Marker lines are parsed through a regular expression (per cell type) into

- a **transition**, to distinguish *begin* and *end* marker lines (from outside to inside, and from inside to outside);
- a **label**, to distinguish holes, especially needed when filling them from a source notebook;
- an optional **description**, mainly as documentation, or as hint to students.

Replacement Content

When filling, the default replacement content is defined in the embedded configuration file as

```
{
  "filling": {
    "markdown": "\n<div class='alert alert-warning' role='alert'>Replace this␣
↪line by your text.</div>\n",
    "code": "\n# ===== =====> Replace this line by your code. <===== ===== #\n
↪",
    "raw": "\n% ===== =====> Replace this line by your content. <===== =====\n
↪"
  }
}
```

These replacement strings can be redefined in your own configuration file; see *Configuration Files*.

In the replacement content, two substitutions are done:

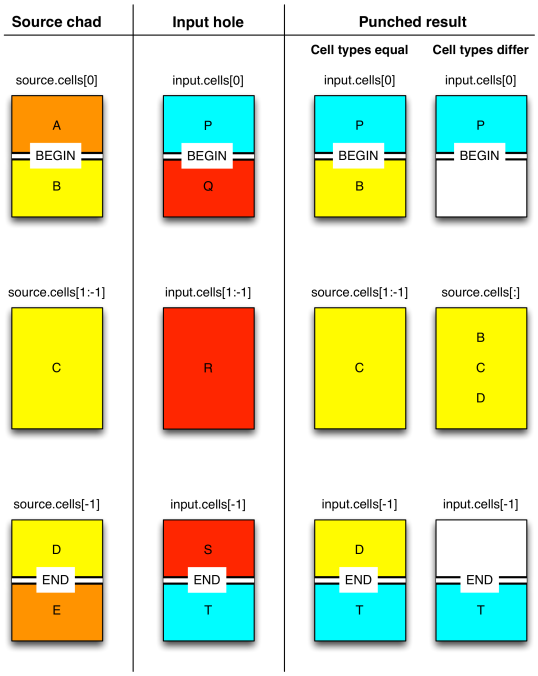
- `{label}` is replaced by the hole's label,
- `{description}` is replaced by the hole's description.

This is especially useful with the `no-keep-marker-lines` option.

With the `source` option, the replacement content is taken from a *source* notebook. This can only be used with *marker lines*. The replacement chad in the source notebook is selected by matching the *labels*. In this case, there can be a mismatch between cell types in the hole to be filled and in the selected source chad to be used as filling.

`punch` ensures that content from source cells ends up in cells of the same type in the punched notebook. In particular, when the `source` option is *not* used, the punched notebook will have the same cells as the input notebook, but possibly with different content inside the holes.

The following diagram illustrates how `punch` uses cells and cell source content when filling from a source notebook.

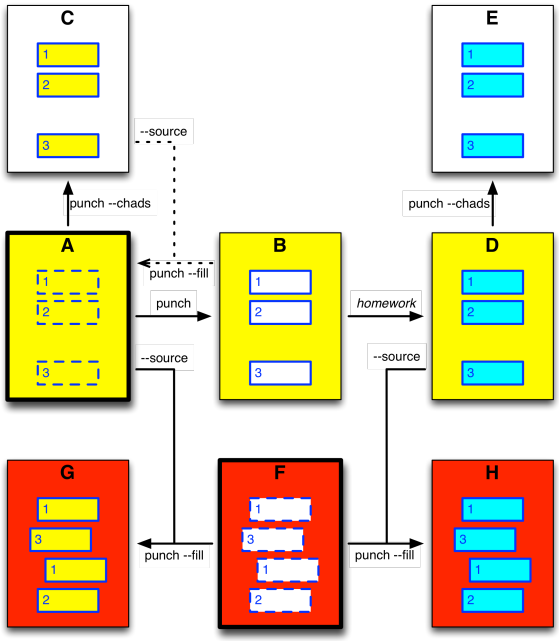


Some notes:

- The number of cells in the source chad and input hole can differ. Both have at least one cell.
- The begin and end marker line can be in the same cell (in which case `cells[0]` and `cells[-1]` are identical). In fact, a cell can contain multiple holes.
- The cell types of the begin cells can differ, and so can the types of the end cells. For instance, the hole could start with a code cell, whereas the replacement source chad starts with a markdown cell. In such cases, `punch` will split cells as necessary.
- All cell metadata, attachments, and outputs are carried along.

3.11.3 Educational Use

`punch` is intended for use in an educational setting. The teacher prepares a *master notebook* **A** with questions and tagged or marked solutions (see diagram).



Using `punch`, the teacher then produces a *punched notebook* **B** (a framework, in software terms), that can be given to students, who need to fill the holes. The teach can also produce a *chads notebook* **C**, with just the solutions, that can be given to assistants. Students deliver their completed notebooks **D**. When punching a student notebook **D**, the *chads notebook* **E** will just contain the student's work.

With reference to the diagram, we have:

- **A** = `master.ipynb` with questions and solutions
- `nbtb punch mater.ipynb --chads` produces
- **B** = `master-punched.ipynb`
- **C** = `master-chads.ipynb`
- student produces
- **D** = `student.ipynb` delivered by student (from `master-punched.ipynb` with work in holes)
- `nbtb punch student.ipynb --chads` produces

- **E** = `student-chads.ipynb` with just the student's work

punch can optionally fill the punched holes either with fixed content (taken from the configuration file), or (in the case of the approach via marker lines) with content taken from another *source notebook* with correspondingly marked content.

The teacher can also prepare a *test notebook* **G**, with correspondingly marked holes, and punch it while using the student's notebook **D** as source to fill the holes. In that way, the student's work is copied into the test notebook in the appropriate places. The resulting notebook **H** can be executed to provide feedback.

punch can also be used to transfer the solutions from the master notebook **A** into the test notebook **G**, obtaining a verification notebook **F**.

With reference to the diagram, we have:

- **F** = *test_frame.ipynb* ` a test notebook to assess work
- `nbtb punch test_frame.ipynb --fill --source master.ipynb` produces
- **G** = `test_frame-punched.ipynb` tes notebook with teacher solutions for verification
- `nbtb punch test_frame.ipynb --fill --source student.ipynb` produces
- **H** = `test_frame-punched.ipynb` test notebook with student work

When filling the template notebook **B** with the solutions (chads) **C**, the original master notebook **A** is obtained:

- `nbtb punch master-punched.ipynb --fill --source master-chads.ipynb` produces
- `master-punched-punched.ipynb` which is equivalent to `master.ipynb`

3.11.4 Options

The following options are supported by punch:

```
-t TAGS, --tags TAGS  comma-separated list of tags that trigger removal of
                      source from cells; e.g. YourTurn (default: '')
-p, --punched, -P, --no-punched
                      whether to write punched notebook (default: True)
-c, --chads, -C, --no-chads
                      whether to write chads notebook (default: False)
-m, --keep-marker-lines, -M, --no-keep-marker-lines
                      whether to keep marker lines (default: True)
-f, --fill, -F, --no-fill
                      whether to fill punched holes (default: False)
-s PUNCH_SOURCE, --source PUNCH_SOURCE
                      source notebook for filling holes; implies --fill
                      (default: "")
-l, --list, -L, --no-list
                      list marker labels and descriptions (default: False)
-e, --allow-errors, -E, --no-allow-errors
                      continue on parsing errors in marker lines (default:
                      False)
--label-regex REGEX
```

(continues on next page)

(continued from previous page)

```
                    only process marker lines whose labels match REGEX
                    (default: ''; empty regex matches all)
```

Tags

The `tags` option defines the trigger set, using a comma-separated list. It can also be defined as a list in the configuration file in your home directory, with the key `tags`.

Punched

By default, `punch` writes the punched notebook. The option `punched` controls this.

Chads

By default, `punch` does not write out the chads notebook. The option `chads` controls this.

Keep Marker Lines

By default, the marker lines are copied to both the punched notebook and the chads notebook. The option `keep-marker-lines` controls this.

Fill

By default, `punch` will not fill the created holes. The option `fill` controls whether `punch` fills holes. Also see option `source` below.

Source

When filling holes, `punch` will take fixed replacement content from the configuration file if option `source` is not supplied; otherwise, it will take replacement content from the source notebook, by matching hole labels. The hole labels must be unique in the source notebook.

The `source` option cannot be used together with the `tags` option.

The `source` options implies the `fill` option.

List

The option `list` will list labels and descriptions of all holes.

Allow Errors

By default, `punch` will abort execution at the first error that is detected in the marker structure. The option `allow-errors` can be used to let `punch` continue processing by assuming a corrected marker structure. In particular,

- when a bad marker is encountered outside a hole, it is ignored, and
- when a bad marker is encountered inside a hole, a correct end marker is assumed, or inserted in case a begin marker was detected.

The number of errors is reported. Do note, however, that when errors occurred, the notebooks involved still need to be fixed. This option is intended only as an aid to finding as many problems in a single run of the tool.

Label Regex

The option `label-regex` can be used to restrict processing to marker lines whose label matches a given regular expression. This filtering affects production of punched and chads files, and to reading of source chads. Regular expressions use Python re syntax. An empty string is interpreted as no filtering (processing all labels).

3.11.5 JSON Output

See *Write JSON Output* for general information about JSON output.

`punch` produces the following members in the JSON output. Note that members are absent when count is zero.

With option `tags`:

Name	Value
`"CT"`	count of cells with type `CT`
`"CT holes"`	count of cells with type `CT` punched

Cell types distinguished: `markdown`, `code`, `raw`

With options `punched` and `chads`:

Name	Value
`"errors"`	count of errors in marker lines
`"holes"`	count of holes detected
`"labels with duplicates"`	count of labels with duplicates
`"unfilled holes"`	count of unfilled holes (with option `fill`)
`"unused source chads"`	count of unused source chads (with option `source`)

3.11.6 Examples

The following example punches all cells having a tag `YourTurn`, in verbose mode.

```
$ nbtb punch -t YourTurn test.ipynb -v
  Options for nbpunch:
    Tags that trigger punch: ['YourTurn']
    Writing punched notebook: True
    Writing chads notebook: False
    Keeping marker lines: True
    Do not fill punched holes
  ::::::::::::::
  test.ipynb
  ::::::::::::::
  Punch cell statistics:
            12 code
             2 code holes
             7 markdown
             1 markdown holes
  No file written

  Notebooks processed: 1
```

The following example uses a source notebook. However, not all holes in the input notebook have replacements in this source notebook. Also the chads notebook is written.

```
$ nbtb punch test-template.ipynb --chads --source test-source.ipynb
  Punch cell statistics:
             8 holes
             2 unfilled holes
```

To find out which holes were not filled, use verbose mode:

```
$ nbtb punch test-template.ipynb --chads --source test-source.ipynb -v
Options for nbpunch:
  Writing punched notebook: True
  Writing chads notebook: True
  Keeping marker lines: True
  Fill holes from source notebook: test-source.ipynb
::::::::::::::
test-template.ipynb
::::::::::::::
Labels of unfilled holes: ['Label_7', 'Label_8']
Punch cell statistics:
           8 holes
           2 unfilled holes
  Files written: {'test-template-punched.ipynb', 'test-template-chads.ipynb'}

Notebooks processed: 1
```

To verify the marker structure, use the options `dry-run`, `list`, `allow-errors`.

```
$ nbtb punch -nle test-template.ipynb test-code-marker-duplicate-label.ipynb␣
↪test-code-marker-bad-transition-missing-end.ipynb
Dry run (no files written)
::::::::::::::
test-template.ipynb
::::::::::::::
Listing labels and descriptions for "test-template.ipynb"
  [Label_1] Do first thing
  [Label_2] Do second, longer, thing
  [Label_3] Do third, even longer, thing
  [Label_4] Write a short text
  [Label_5] Write a medium text
  [Label_6] Write longer text
  [Label_7] Do seventh, mixed, thing.
  [Label_8] Do another mixed problem.
Punch cell statistics:
           8 holes
No file written
::::::::::::::
test-code-marker-duplicate-label.ipynb
::::::::::::::
Listing labels and descriptions for "test-code-marker-duplicate-label.ipynb"
  [Label_1] Begin of first hole
  [Label_1] Begin of second hole, with duplicate label
Punch cell statistics:
           2 holes
           1 labels with duplicates
No file written
::::::::::::::
test-code-marker-bad-transition-missing-end.ipynb
::::::::::::::
Listing labels and descriptions for "test-code-marker-bad-transition-missing-
↪end.ipynb"
  [Label_1] Begin of first hole
Unexpected transition in marker line of cell 2 on line 1:
  #// BEGIN_TODO [Label_2] Begin of second hole
  Expected END [Label_1], but found BEGIN.
  Inserting correct end marker
  [Label_2] Begin of second hole
Punch cell statistics:
           1 errors
           2 holes
No file written
```

3.12 Configuration Files

`nbtoolbelt` can load options from a JSON configuration file: see *Read Configuration File*.

Such a configuration file holds one object in JSON (JavaScript Object Notation), with optional members for global options and options for each individual tool:

- `nbtoolbelt`: global options that apply to multiple tools
- `nbvalidate`: options for *Notebook Validation*
- `nbheads`: options for *Notebook Heads*
- `nbdump`: options for *Notebook Dump*
- `nbstats`: options for *Notebook Statistics*
- `nbview`: options for *Notebook Viewing*
- `nbcat`: options for *Notebook Catenation*
- `nbclean`: options for *Notebook Cleaning*
- `nbrun`: options for *Notebook Execution*
- `nbsplit`: options for *Notebook Splitting*
- `nbpunch`: options for *Notebook Punching*

The options for a tool are contained in a single object, with a member per option. The member name is the same as the long command-line option, in which dashes ('-') have been replaced by underscores ('_').

Whenever a configuration file is loaded,

1. the common options from that file are applied (with attribute `"nbtoolbelt"`),
2. those options are applied that concern the selected tool.

Thus, tool-specific options override common options in the same configuration file, but not the other way round.

Note: Note

- that JSON syntax is very strict;
- that `nbtb config [tool]` will report the configuration using **Python syntax**;
- that JSON *booleans* are spelled in **all lower case**: `false` and `true`;
- that JSON *strings* must be surrounded by **double quotes** (`"..."`).
- that JSON does *not* allow trailing commas in arrays and objects;
- that JSON does *not* support any form of *comments*.

Errors when loading a configuration file will be sent to `stderr`, and execution is aborted immediately.

3.12.1 Example

Here is the `embedded configuration file`:

```
{
  "nbtoolbelt": {
    "notebooks": [],
    "verbose": false,
    "quiet": false,
    "assert": true,
    "validate": false,

    "run": false,
    "kernel_name": "",
    "run_path": "",
    "timeout": -1,
    "interrupt_on_timeout": true,
    "record_timing": true,
    "ipc": "",
    "allow_errors": true,
    "clean_before": false,
    "clean_after": false,
    "clean_after_metadata": [
    ],
    "streams_head": -1,
    "streams_truncate_message": "*** Output truncated ***",
    "append_cell": false,
    "appended_cell": "# Automatically added code cell: lists all global names␣
↪defined by this notebook.\n%whos",

    "inplace": false,
    "write_files": true,
    "output_json": null,
    "debug": false
  },

  "nbvalidate": {

  },

  "nbhead": {
    "number": 5
  },

  "nbdump": {
    "dump_notebook_info": true,
    "dump_cell_info": true,
    "dump_sources": true,
    "dump_cell_types": [
      "markdown",
      "code",
```

(continues on next page)

(continued from previous page)

```
      "raw"
    ],
    "dump_info_prefix": "=====",
    "dump_source_line_indent": 2,
    "dump_source_line_numbers": false,
    "dump_cell_spacing": 0
  },

  "nbstats": {
    "all_stats": false,
    "cell_types": true,
    "sources": false,
    "metadata": false,
    "tags": false,
    "attachments": false,
    "outputs": false,
    "streams": false,
    "errors": false,
    "execution": false,
    "extra": false
  },

  "nbview": {
    "browser": true,
    "template_file": "",
    "template": "classic",
    "wait_delete": false,
    "view_result_name": "-view"
  },

  "nbcat": {
    "cat_result_name": "-cat"
  },

  "nbrun": {
    "kernel_name": "",
    "run_path": "",
    "timeout": -1,
    "interrupt_on_timeout": true,
    "allow_errors": true,
    "record_timing": true,
    "clean_before": true,
    "clean_before_metadata": [
      "ExecuteTime",
      "execution"
    ],
    "clean_after": true,
    "clean_after_metadata": [
      "collapsed",
```

(continues on next page)

(continued from previous page)

```
      "scrolled"
    ],
    "streams_head": -1,
    "append_cell": false,
    "run_result_name": "-run"
  },

  "nbclean": {
    "clean_notebook_metadata_fields": [],
    "clean_cell_metadata_fields": [],
    "clean_tags": [],
    "clean_empty_cells": false,
    "clean_outputs": false,
    "clean_result_name": "-clean"
  },

  "nbsplit": {
    "split_cell_types": [
      "markdown",
      "code",
      "raw"
    ],
    "split_markdown_result_name": "-markdown",
    "split_code_result_name": "-code",
    "split_raw_result_name": "-raw"
  },

  "nbpunch": {
    "tags": [
    ],
    "punched": true,
    "chads": false,
    "keep_marker_lines": true,
    "marker_regex": {
      "markdown": "#//.*_TODO",
      "code": "#//.*_TODO",
      "raw": "#//.*_TODO"
    },
    "marker_line_parsing_regex": {
      "markdown": "`#// (?P<transition>.*?)_TODO \\[(?P<label>.*?)\\] ?(?P
↪<description>.*?)`",
      "code": "#// (?P<transition>.*?)_TODO \\[(?P<label>.*?)\\] ?(?P
↪<description>.*)",
      "raw": "#// (?P<transition>.*?)_TODO \\[(?P<label>.*?)\\] ?(?P
↪<description>.*)"
    },
    "marker_transitions": {
      "begin": "BEGIN",
      "end": "END"
```

(continues on next page)

(continued from previous page)

```
    },
    "fill": false,
    "filling": {
      "markdown": "\n<div class='alert alert-warning' role='alert'>Replace␣
↪this line by your text.</div>\n",
      "code": "\n# ===== =====> Replace this line by your code. <===== =====
↪#\n",
      "raw": "\n% ===== =====> Replace this line by your content. <=====␣
↪=====\n"
    },
    "list": false,
    "allow_errors": false,
    "punch_source": "",
    "source_chads": {},
    "punch_punched_result_name": "-punched",
    "punch_chads_result_name": "-chads",
    "label_regex": ""
  }
}
```

Here is an `example configuration file` with one global option (for verbose mode) and options for running and punching. It can either be loaded through the command-line option `--config nbtoolbelt.json` or by putting it in your home directory as `~/.nbtoolbelt.json` (note the *dot*, which makes it a hidden file).

```
{
  "nbtoolbelt": {
    "verbose": true
  },
  "nbrun": {
    "kernel_name": "python3",
    "timeout": 5,
    "allow_errors": false
  },
  "nbpunch": {
    "tags": [ "YourTurn" ]
  }
}
```

3.13 Library Usage

`nbtoolbelt` can be used in Python 3 programs as a library.

3.13.1 Example

An `example script using nbtoolbelt as library`:

```
#! /usr/bin/env python
"""
Example script that illustrates use of nbtoolbelt as a library

It expects one argument on the command-line, opens it as a Jupyter notebook,␣
↪and
shows its global metadata and counts per cell type.
"""

import sys
import nbformat
import nbtoolbelt as nbtb


def main() -> int:
    """Main entry point.

    :return: exit code
    """
    try:
        nb = nbformat.read(sys.argv[1], as_version=4)
    except Exception as e:
        print('{}: {}'.format(type(e).__name__, e), sys.stderr)
        return 1

    nbtb.print_dict(nbtb.nb_metadata(nb), 'Notebook metadata')

    nbtb.print_dict(nbtb.nb_cell_stats(nb)['cell_types'], 'Cell types')

    return 0


if __name__ == '__main__':
    sys.exit(main())
```

Here is a sample run of this script:

```
$ example.py test.ipynb
Notebook metadata:
         4.2 format version
     python3 kernel
  python 3.6.1 language
```

(continues on next page)

(continued from previous page)

```
Cell types:
          12 code
           7 markdown
          19 total cell count
```

3.14 API Documentation

3.15 Adding Tools to `nbtoolbelt`

`nbtoolbelt` is set up in a modular way, so that it easy to add new tools.

The **base class** `Tool` is defined in toolbaseapp.py.

Every tool extends `Tool`, and thereby inherits the common code for argument parsing, looping over all argmuments, reading and writing of notebooks. The tool only needs to **override**

- the constructor `__init__()` [it **must call** `super()` at the beginning],

and following **hook methods**:

- `config_tool_args_parsing()` to configure the parser for tool-specific arguments [optional; no need to call `super()`]
- `check_and_adjust_arguments()` to check and adjust tool-specific arguments, e.g. to enforce consistence [optional; no need to call `super()`]
- `print_tool_args()` to print tool-specific arguments, called in verbose mode [optional; no need to call `super()`]
- `process_nb()` to process one notebook, returning produced notebooks [**must NOT call** `super()`]
- `process_collected_data()` to process data collected for all notebook [**must call** `super()` at the beginning]

For a simple example, see the `head` tool.

- documentation: *Notebook Heads*
- source file: nbheadapp.py

3.16 Testing `nbtoolbelt`

`nbtoolbelt` comes with automated test cases using `pytest` and `pytest-mock`.

For the purpose of testing, the command-line script `nbtb` has a *hidden command* `base`. This command invokes the base class `Tool` (also see *Adding Tools to nbtoolbelt*), which copies notebooks:

- notebook `nb.ipynb` is copied to `nb-copy.ipynb`

```
$ nbtb base test.ipynb
Copying: test.ipynb
```

Test cases for concrete tools need not test functionality of the base class.

3.17 Change Log

3.17.1 2024.1.dev0

- Update the Read-the-Docs YAML file with required build keys.
- Add generation of `pytest` code coverage information.
- Add some developer information.

3.17.2 2022.4

- Explicitly dropped support for Python 3.6.
- Improved declaration of dependencies.
- Removed use of `pandas.DataFrame.append` (which is deprecated), and replaced it by `pandas.concat`.

3.17.3 2022.3

- Added the `record-timing` option to `nbrun` (default `True`).
- Added `clean-before-metadata` option to clean code metadata before running. By default, execution code metadata is cleaned before running.
- Updated `nbview` to support `nbconvert` 6 templates, by deprecating the `template-file` option and adding the `template` option, which defaults to `classic`.
- Added explicit version dependencies in `setup.py`.
- Changed development status to production/stable, since `nbtoolbelt` has been used successfully for more than 4 years in production. In that time frame, `nbtoolbelt` processed over ten million notebooks.

3.17.4 2022.3.dev0

- Fixed and added test cases for nbrun with cell timeout.
- Removed duplicate keys in default configuration file.
- Documented nbrun `interrupt-on-timeout` option. Fixed typo in nbrun documentation.
- Changed nbrun default for `interrupt-on-timeout` option to `True`.
- Show nbrun `interrupt-on-timeout` option value in verbose mode.
- Fixed type hints for main methods of the tools.

3.17.5 2020.7.dev2

- Removed dependency on `pypandoc` and used `README.md` directly.

3.17.6 2020.7.dev1

- Mentioned dependency on `pypandoc` for packaging (to get the long project description from `README.md`).

3.17.7 2020.7.dev0

- Added more documentation in source code.
- Exported also version info and package name.
- In `run`, only set kernel name if non-empty.
- In `punch`, added rudimentary support for reading of chads from multiple source notebooks.
- In `punch`, added option `--label-regex` to process only marker lines with label matching given regex.
- Added more test cases for `run`.
- In top-level `REAMDE`, listed steps to add a feature to `nbtoolbelt`.
- Instructed Read the Docs to generate documentation in all formats.

3.17.8 2020.2.dev0

- Fixed inlining of attachments with names containing (quoted) spaces.

3.17.9 2018.2.dev1

- Improved documentation; in particular, documented JSON output for each tool.
- Report `empty outputs` under `Code cell outputs` instead of `Code cell execution` using the label `code cells without outputs`.
- Fixed `tests_require` in `setup.py`.

3.17.10 2018.2.dev0

- Improved help text in `punch` for option `allow-errors`.
- Fixed `json` output for `punch`; it now includes statistics gathered during processing.

3.17.11 2018.1.dev0

- By default, `clean` does nothing (previously it would clean all outputs by default).
- Improved error reporting in `punch` when bad marker lines are detected.
- Added option `--list` (`-l`) to `punch` that list all labels and descriptions.
- Added option `--allow-errors` (`-e`) to `punch` that continues on marker errors.
- Tool `stats` now also shows counts of empty sources per cell type; minor improvements in labeling of statistics.
- Tool `view` now has option `template-file` to select a template file other than default `full`.

3.17.12 2017.11.dev1

- Fixed code to comply with Python 3.5 and PEP 8

3.17.13 2017.10.dev5

- Fixed generation of documentation for Read the Docs (had to switch it to Python 3.5)
- Fixed defect in `punch` with frequencies when writing chads notebook but not punched notebook
- Updated README now that the source code repository has gone public

3.17.14 2017.10.dev4

- Was not (intended to be) released

3.17.15 2017.10.dev3

- Converted README to reST
- Improved problem reporting in `punch`

3.17.16 2017.10.dev2

Initial release on PyPI

3.17.17 2017.9.dev3

Initial private release

CHAPTER

FOUR

REFERENCES

- The structure of Jupyter notebooks is specified in `nbformat` (documentation). The library `nbformat` facilitates programmatic manipulation of notebooks.
- Tool to convert notebooks: `nbconvert` (documentation); can be used on the command line and as a library for conversion to various other formats, including notebook execution and other notebook versions. It also provides a set of notebook preprocessors.
- Tools for showing, diffing, and merging Jupyter notebooks, including Git integration: `nbdime` (documentation)

CHAPTER

FIVE

LICENSE AND SOURCE CODE

This software is made available under the terms of the `MIT License`.

Copyright (c) 2017-2020 - Eindhoven University of Technology, The Netherlands

The source code and issue tracker are at <https://gitlab.tue.nl/jupyter-projects/nbtoolbelt/>.

5.1 Source Code

5.1.1 nbtoolbelt

nbtoolbelt package

Submodules

nbtoolbelt.arguments module

nbtoolbelt.cleaning module

nbtoolbelt.config module

nbtoolbelt.counting module

nbtoolbelt.inline_attachments module

nbtoolbelt.nbcatapp module

nbtoolbelt.nbcleanapp module

nbtoolbelt.nbdumpapp module

nbtoolbelt.nbheadapp module

nbtoolbelt.nbpunchapp module

nbtoolbelt.nbrunapp module

nbtoolbelt.nbsplitapp module

nbtoolbelt.nbstatsapp module

nbtoolbelt.nbvalidateapp module

nbtoolbelt.nbviewapp module

nbtoolbelt.notebook_io module

nbtoolbelt.printing module

nbtoolbelt.processing module

nbtoolbelt.punching module

nbtoolbelt.rendering module

nbtoolbelt.running module

nbtoolbelt.toolbaseapp module

nbtoolbelt.validating module

Module contents

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex