nbclient Documentation

Release 0.10.0

jupyter team

Mar 13, 2024

DOCUMENTATION

—

NBClient lets you **execute** notebooks.

A client library for programmatic notebook execution, **NBClient** is a tool for running Jupyter Notebooks in different execution contexts, including the command line. NBClient was spun out of nbconvert's former `ExecutePreprocessor`.

CHAPTER

ONE

DEMO

To demo **NBClient** interactively, click the Binder link below:

CHAPTER

TWO

ORIGINS

This library used to be part of nbconvert and was extracted into its ownlibrary for easier updating and importing by downstream libraries and applications.

CHAPTER

THREE

PYTHON VERSION SUPPORT

This library currently supports python 3.6+ versions. As minor python versions are officially sunset by the python org, nbclient will similarly drop support in the future.

CHAPTER

FOUR

DOCUMENTATION

These pages guide you through the installation and usage of nbclient.

4.1 Installation

4.1.1 Installing nbclient

From the command line:

```
python3 -m pip install nbclient
```

See also:

Installing Jupyter
NBClient is part of the Jupyter ecosystem.

4.2 Executing notebooks

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

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

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

4.2.1 Using the Python API interface

This section will illustrate the Python API interface.

Example

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

Import: First we import nbformat and the `NotebookClient` class:

```
import nbformat
from nbclient import NotebookClient
```

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

```
nb = nbformat.read(notebook_filename, as_version=4)
```

Configure: Next, we configure the notebook execution mode:

```
client = NotebookClient(nb, timeout=600, kernel_name='python3', resources={'metadata': {
↪'path': 'notebooks/'}})
```

We specified two (optional) arguments `timeout` and `kernel_name`, which define respectively the cell execution timeout and the execution kernel. Usually you don't need to set these options, but these and other options are available to control execution context. Note that `path` specifies in which folder to execute the notebook.

Execute/Run: To actually run the notebook we call the method `execute`:

```
client.execute()
```

Hopefully, we will not get any errors during the notebook execution (see the last section for error handling). This notebook will now have its cell outputs populated with the result of running each cell.

Save: Finally, save the resulting notebook with:

```
nbformat.write(nb, 'executed_notebook.ipynb')
```

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

4.2.2 Execution arguments (traitlets)

The arguments passed to `NotebookClient` are configuration options called traitlets. There are many cool things about traitlets. For example, they enforce the input type, and they can be accessed/modified as class attributes.

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

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

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

4.2.3 Hooks before and after notebook or cell execution

There are several configurable hooks that allow the user to execute code before and after a notebook or a cell is executed. Each one is configured with a function that will be called in its respective place in the execution pipeline. Each is described below:

Notebook-level hooks: These hooks are called with a single extra parameter:

- `notebook=NotebookNode`: the current notebook being executed.

Here is the available hooks:

- `on_notebook_start` will run when the notebook client is initialized, before any execution has happened.
- `on_notebook_complete` will run when the notebook client has finished executing, after kernel cleanup.
- `on_notebook_error` will run when the notebook client has encountered an exception before kernel cleanup.

Cell-level hooks: These hooks are called with at least two parameters:

- `cell=NotebookNode`: a reference to the current cell.
- `cell_index=int`: the index of the cell in the current notebook's list of cells.

Here are the available hooks:

- `on_cell_start` will run for all cell types before the cell is executed.
- `on_cell_execute` will run right before the code cell is executed.
- `on_cell_complete` will run after execution, if the cell is executed with no errors.
- `on_cell_executed` will run right after the code cell is executed.
- `on_cell_error` will run if there is an error during cell execution.

`on_cell_executed` and `on_cell_error` are called with an extra parameter `execute_reply=dict`.

4.2.4 Handling errors and exceptions

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

Execution until first error

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

```
nbformat.write(nb, 'executed_notebook.ipynb')
```

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

Handling errors

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

```
from nbclient.exceptions import CellExecutionError

try:
    client.execute()
except CellExecutionError:
    msg = 'Error executing the notebook "%s".\n\n' % notebook_filename
    msg += 'See notebook "%s" for the traceback.' % notebook_filename_out
    print(msg)
    raise
finally:
    nbformat.write(nb, notebook_filename_out)
```

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

Execute and save all errors

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

4.2.5 Widget state

If your notebook contains any Jupyter Widgets, the state of all the widgets can be stored in the notebook's metadata. This allows rendering of the live widgets on for instance nbviewer, or when converting to html.

We can tell nbclient to not store the state using the `store_widget_state` argument:

```
client = NotebookClient(nb, store_widget_state=False)
```

This widget rendering is not performed against a browser during execution, so only widget default states or states manipulated via user code will be calculated during execution. `%%javascript` cells will execute upon notebook rendering, enabling complex interactions to function as expected when viewed by a UI.

If you can't view widget results after execution, you may need to select *Trust Notebook* under the *File* menu.

4.2.6 Using a command-line interface

This section will illustrate how to run notebooks from your terminal. It supports the most basic use case. For more sophisticated execution options, consider the papermill library.

This library's command line tool is available by running `jupyter execute`. It expects notebooks as input arguments and accepts optional flags to modify the default behavior.

Running a notebook is this easy.:

```
jupyter execute notebook.ipynb
```

You can pass more than one notebook as well.:

```
jupyter execute notebook.ipynb notebook2.ipynb
```

By default, notebook errors will be raised and printed into the terminal. You can suppress them by passing the `--allow-errors` flag.:

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

Other options allow you to modify the timeout length and dictate the kernel in use. A full set of options is available via the help command.:

```
jupyter execute --help

An application used to execute notebook files (*.ipynb)

Options
=======
The options below are convenience aliases to configurable class-options,
as listed in the "Equivalent to" description-line of the aliases.
To see all configurable class-options for some <cmd>, use:
    <cmd> --help-all

--allow-errors
    Errors are ignored and execution is continued until the end of the notebook.
    Equivalent to: [--NbClientApp.allow_errors=True]
--timeout=<Int>
    The time to wait (in seconds) for output from executions. If a cell
    execution takes longer, a TimeoutError is raised. ``-1`` will disable the
    timeout.
    Default: None
    Equivalent to: [--NbClientApp.timeout]
--startup_timeout=<Int>
    The time to wait (in seconds) for the kernel to start. If kernel startup
    takes longer, a RuntimeError is raised.
    Default: 60
    Equivalent to: [--NbClientApp.startup_timeout]
--kernel_name=<Unicode>
    Name of kernel to use to execute the cells. If not set, use the kernel_spec
    embedded in the notebook.
    Default: ''
    Equivalent to: [--NbClientApp.kernel_name]

To see all available configurables, use `--help-all`.
```

4.3 Changes in NBClient {#changelog}

4.3.1 0.10.0

(Full Changelog)

Enhancements made

- Optionally write out executed notebook in jupyter-execute #307 (@wpk-nist-gov)

Contributors to this release

(GitHub contributors page for this release)

@blink1073 | @wpk-nist-gov

4.3.2 0.9.1

(Full Changelog)

Maintenance and upkeep improvements

- Update Release Scripts #309 (@blink1073)
- Pin to Pytest 7 #308 (@blink1073)

Other merged PRs

- chore: update pre-commit hooks #305 (@pre-commit-ci)
- chore: update pre-commit hooks #304 (@pre-commit-ci)
- chore: update pre-commit hooks #303 (@pre-commit-ci)
- Bump actions/checkout from 3 to 4 #302 (@dependabot)
- chore: update pre-commit hooks #300 (@pre-commit-ci)

Contributors to this release

(GitHub contributors page for this release)

@blink1073 | @dependabot | @pre-commit-ci

4.3.3 0.9.0

(Full Changelog)

Maintenance and upkeep improvements

- Use jupyter releaser #301 (@blink1073)
- Clean up lint and move tests out of source #299 (@blink1073)
- Adopt ruff format #298 (@blink1073)
- Update typings for mypy 1.6 #297 (@blink1073)
- Adopt sp-repo-review #295 (@blink1073)
- Fix lint error #289 (@blink1073)

Other merged PRs

- Bump actions/checkout from 3 to 4 #293 (@dependabot)

Contributors to this release

(GitHub contributors page for this release)

@blink1073 | @dependabot | @pre-commit-ci

4.3.4 0.8.0

(Full Changelog)

Maintenance and upkeep improvements

- Bump min version support #287 (@blink1073)

Other merged PRs

- Bump actions/checkout from 2 to 3 #275 (@dependabot)

Contributors to this release

(GitHub contributors page for this release)

@blink1073 | @dependabot | @pre-commit-ci

4.3.5 0.7.4

(Full Changelog)

Enhancements made

- include stream output in CellExecutionError #282 (@minrk)

Bugs fixed

- avoid duplicate 'Exception: message' in CellExecutionError #283 (@minrk)

Maintenance and upkeep improvements

- Use local coverage #281 (@blink1073)

Other merged PRs

- Send KeyboardInterrupt a little later in test_run_all_notebooks[Interrupt.ipynb-opts6] #285 (@kxxt)

Contributors to this release

(GitHub contributors page for this release)

@blink1073 | @davidbrochart | @kxxt | @minrk | @pre-commit-ci

4.3.6 0.7.3

(Full Changelog)

Maintenance and upkeep improvements

- Fix test stability #276 (@blink1073)
- Clean up license #274 (@dcsaba89)
- Update codecov link #271 (@blink1073)
- Add spelling and docstring enforcement #269 (@blink1073)
- Adopt ruff and address lint #267 (@blink1073)

Other merged PRs

- Add coalesce_streams #279 (@davidbrochart)

Contributors to this release

(GitHub contributors page for this release)

@blink1073 | @davidbrochart | @dcsaba89 | @pre-commit-ci

4.3.7 0.7.2

(Full Changelog)

Merged PRs

- Allow space after In #264 (@davidbrochart)
- Fix jupyter_core pinning #263 (@davidbrochart)
- Update README, add Python 3.11 #260 (@davidbrochart)

Contributors to this release

(GitHub contributors page for this release)

@davidbrochart

4.3.8 0.7.1

(Full Changelog)

Maintenance and upkeep improvements

- CI Refactor #257 (@blink1073)

Other merged PRs

- Remove nest-asyncio #259 (@davidbrochart)
- Add upper bound to dependencies #258 (@davidbrochart)

Contributors to this release

(GitHub contributors page for this release)

@blink1073 | @davidbrochart | @pre-commit-ci

4.3.9 0.7.0

(Full Changelog)

Maintenance and upkeep improvements

- Cleanup CI #254 (@blink1073)
- Handle client 8 support #253 (@blink1073)

Other merged PRs

Contributors to this release

(GitHub contributors page for this release)

@blink1073 | @pre-commit-ci

4.3.10 0.6.8

(Full Changelog)

Merged PRs

- Fix tests compatibility with IPython 8.5.0 #251 (@frenzymadness)

Contributors to this release

(GitHub contributors page for this release)

@davidbrochart | @frenzymadness

4.3.11 0.6.7

(Full Changelog)

Merged PRs

- Fix tests for ipywidgets 8 #246 (@frenzymadness)
- [pre-commit.ci] pre-commit autoupdate #236 (@pre-commit-ci)

Contributors to this release

(GitHub contributors page for this release)

@frenzymadness | @pre-commit-ci

4.3.12 0.6.6

(Full Changelog)

Merged PRs

- Start new client if needed in blocking setup_kernel #241 (@davidbrochart)

Contributors to this release

(GitHub contributors page for this release)

@davidbrochart

4.3.13 0.6.5

(Full Changelog)

Merged PRs

- Start new client if needed #239 (@davidbrochart)

Contributors to this release

(GitHub contributors page for this release)

@davidbrochart

4.3.14 0.6.4

(Full Changelog)

Merged PRs

- Make sure kernel is cleaned up in case an error occurred while starting kernel client #234 (@CiprianAnton)
- Suppress most warnings in tests #232 (@davidbrochart)

Contributors to this release

(GitHub contributors page for this release)

@CiprianAnton | @davidbrochart

4.3.15 0.6.3

(Full Changelog)

Bugs fixed

- Clean up docs and typings #230 (@blink1073)

Documentation improvements

- Clean up docs and typings #230 (@blink1073)

Contributors to this release

(GitHub contributors page for this release)

@blink1073 | @chrisjsewell | @davidbrochart | @meeseeksmachine

4.3.16 0.6.2

(Full Changelog)

Merged PRs

- Fix documentation generation #228 (@davidbrochart)

Contributors to this release

(GitHub contributors page for this release)

@davidbrochart

4.3.17 0.6.1

(Full Changelog)

Merged PRs

- [pre-commit.ci] pre-commit autoupdate #225 (@pre-commit-ci)
- Add error_on_interrupt trait #224 (@davidbrochart)
- Fix typo #223 (@davidbrochart)
- Add on_cell_executed hook #222 (@davidbrochart)

Contributors to this release

(GitHub contributors page for this release)

@brichet | @davidbrochart | @pre-commit-ci

4.3.18 0.6.0

(Full Changelog)

Maintenance and upkeep improvements

- Fix typings and update mypy settings #220 (@blink1073)
- Add missing dep on testpath #219 (@blink1073)
- Add more pre-commit hooks and update flake8 #218 (@blink1073)

Documentation improvements

- Clean up docs handling #216 (@blink1073)

Contributors to this release

(GitHub contributors page for this release)

@blink1073

4.3.19 0.5.13

(Full Changelog)

Merged PRs

- Drop ipython_genutils #209 (@davidbrochart)

Contributors to this release

(GitHub contributors page for this release)

@davidbrochart

4.3.20 0.5.12

(Full Changelog)

Merged PRs

- Require traitlets>=5.0.0 #204 (@davidbrochart)
- Extend the ignored part of IPython outputs #202 (@frenzymadness)

Contributors to this release

(GitHub contributors page for this release)

@davidbrochart | @frenzymadness

4.3.21 0.5.11

(Full Changelog)

Merged PRs

- Pin ipython<8 in tests #198 (@davidbrochart)
- Clear execution metadata, prefer msg header date when recording times #195 (@kevin-bates)
- Client hooks #188 (@devintang3)

Contributors to this release

(GitHub contributors page for this release)

@davidbrochart | @devintang3 | @kevin-bates

4.3.22 0.5.10

(Full Changelog)

Merged PRs

- Fix ipywidgets version in tests #192 (@martinRenou)
- Compatibility with IPython 8 where tracebacks are different #190 (@frenzymadness)
- Drop tox #187 (@davidbrochart)
- Update README #185 (@davidbrochart)
- Drop python3.6, test python3.10 #184 (@davidbrochart)
- Fix typos #182 (@kianmeng)
- Use codecov Github action v2 #168 (@takluyver)

Contributors to this release

(GitHub contributors page for this release)

@davidbrochart | @frenzymadness | @kianmeng | @martinRenou | @takluyver

4.3.23 0.5.9

(Full Changelog)

Merged PRs

- Remove jupyter-run, keep jupyter-execute #180 (@davidbrochart)

Contributors to this release

(GitHub contributors page for this release)

@davidbrochart

4.3.24 0.5.8

No merged PRs

4.3.25 0.5.7

(Full Changelog)

Merged PRs

- Prepare for use with Jupyter Releaser #175 (@davidbrochart)

Contributors to this release

(GitHub contributors page for this release)

@davidbrochart

4.3.26 0.5.6

- Changed `jupyter execute` to `jupyter run` #173 (@palewire)
- Move IPYKERNEL_CELL_NAME from tox to pytest #172 (@frenzymadness)

4.3.27 0.5.5

- Added CLI to README #170 (@palewire)
- Add "jupyter execute" command-line interface #165 (@palewire)
- Fix: updating buffers overwrote previous buffers #169 (@maartenbreddels)
- Fix tests for ipykernel without debugpy #166 (@frenzymadness)
- gitignore Pipfile #164 (@palewire)
- Fixed CONTRIBUTING.md link #163 (@palewire)
- Fix typo #162 (@The-Compiler)
- Move format & lint to pre-commit #161 (@chrisjsewell)
- Add `skip-execution` cell tag functionality #151 (@chrisjsewell)

4.3.28 0.5.4

- Replace `km.cleanup` with `km.cleanup_resources` #152 (@davidbrochart)
- Use async generator backport only on old python #154 (@mkoeppe)
- Support parsing of IPython dev version #150 (@cphyc)
- Set `IPYKERNEL_CELL_NAME = <IPY-INPUT>` #147 (@davidbrochart)
- Print useful error message on exception #142 (@certik)

4.3.29 0.5.3

- Fix ipykernel's `stop_on_error` value to take into account `raises-exception` tag and `force_raise_errors` #137

4.3.30 0.5.2

- Set minimum python version supported to 3.6.1 to avoid 3.6.0 issues
- CellExecutionError is now unpickleable
- Added testing for python 3.9
- Changed travis tests to github actions
- Documentation referencing an old model instead of NotebookClient was fixed
- `allow_error_names` option was added for a more specific scope of `allow_errors` to be applied

4.3.31 0.5.1

- Update kernel client class JIT if it's the synchronous version
- Several documentation fixes / improvements

4.3.32 0.5.0

- Move `language_info` retrieval before cell execution #102
- HistoryManager setting for ipython kernels no longer applies twice (fix for 5.0 trailets release)
- Improved error handling around language_info missing
- `(async_)start_new_kernel_client` is now split into `(async_)start_new_kernel` and `(async_)start_new_kernel_client`

4.3.33 0.4.2 - 0.4.3

These patch releases were removed due to backwards incompatible changes that should have been a minor release. If you were using these versions for the couple days they were up, move to 0.5.0 and you shouldn't have any issues.

4.3.34 0.4.1

- Python type hinting added to most interfaces! #83
- Several documentation fixes and improvements were made #86
- An asynchronous heart beat check was added to correctly raise a DeadKernelError when kernels die unexpectantly #90

4.3.35 0.4.0

Major Changes

- Use KernelManager's graceful shutdown rather than KILLing kernels #64
- Mimic an Output widget at the frontend so that the Output widget behaves correctly #68
- Nested asyncio is automatic, and works with Tornado #71
- `async_execute` now has a `reset_kc` argument to control if the client is reset upon execution request #53

Fixes

- Fix `OSError: [WinError 6] The handle is invalid` for windows/python<3.7 #77
- Async wrapper Exceptions no longer loose their caused exception information #65
- `extra_arguments` are now configurable by config settings #66

Operational

- Cross-OS testing now run on PRs via Github Actions #63

4.3.36 0.3.1

Fixes

- Check that a kernel manager exists before cleaning up the kernel #61
- Force client class to be async when kernel manager is MultiKernelManager #55
- Replace pip install with conda install in Binder #54

4.3.37 0.3.0

Major Changes

- The `(async_)start_new_kernel_client` method now supports starting a new client when its kernel manager (`self.km`) is a `MultiKernelManager`. The method now returns the kernel id in addition to the kernel client. If the kernel manager was a `KernelManager`, the returned kernel id is `None`. #51
- Added sphinx-book-theme for documentation. Added a CircleCI job to let us preview the built documentation in a PR. #50
- Added `reset_kc` option to `reset_execution_trackers`, so that the kernel client can be reset and a new one created in calls to `(async_)execute` #44

Docs

- Fixed documentation #46 #47
- Added documentation status badge to the README
- Removed conda from documentation build

4.3.38 0.2.0

Major Changes

- Async support is now available on the client. Methods that support async have an `async_` prefix and can be awaited #10 #35 #37 #38
- Dropped support for Python 3.5 due to async compatibility issues #34
- Notebook documents now include the new kernel timing fields #32

Fixes

- Memory and process leaks from nbclient should now be fixed #34
- Notebook execution exceptions now include error information in addition to the message #41

Docs

- Added binder examples / tests #7
- Added changelog to docs #22
- Doc typo fixes #27 #30

4.3.39 0.1.0

- Initial release – moved out of nbconvert 6.0.0-a0

CHAPTER

FIVE

API REFERENCE

If you are looking for information about a specific function, class, or method, this documentation section will help you.

5.1 Reference

This part of the documentation lists the full API reference of all public classes and functions.

5.1.1 nbclient package

Subpackages

Submodules

nbclient.client module

nbclient implementation.

class nbclient.client.**NotebookClient**(***kwargs: Any*)

Bases: LoggingConfigurable

Encompasses a Client for executing cells in a notebook

allow_error_names

List of error names which won't stop the execution. Use this if the allow_errors option it too general and you want to allow only specific kinds of errors.

allow_errors

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

async async_execute(*reset_kc: bool = False*, ***kwargs: Any*) → NotebookNode

Executes each code cell.

Parameters

kwargs – Any option for self.kernel_manager_class.start_kernel(). Because that defaults to AsyncKernelManager, this will likely include options accepted by jupyter_client.AsyncKernelManager.start_kernel(), which includes cwd.

reset_kc if True, the kernel client will be reset and a new one will be created (default: False).

Returns
nb – The executed notebook.

Return type
NotebookNode

async async_execute_cell(*cell: NotebookNode*, *cell_index: int*, *execution_count: int | None = None*, *store_history: bool = True*) → NotebookNode

Executes a single code cell.

To execute all cells see *execute()*.

Parameters

- **cell** (*nbformat.NotebookNode*) – The cell which is currently being processed.
- **cell_index** (*int*) – The position of the cell within the notebook object.
- **execution_count** (*int*) – The execution count to be assigned to the cell (default: Use kernel response)
- **store_history** (*bool*) – Determines if history should be stored in the kernel (default: False). Specific to ipython kernels, which can store command histories.

Returns
output – The execution output payload (or None for no output).

Return type
dict

Raises
CellExecutionError – If execution failed and should raise an exception, this will be raised with defaults about the failure.

Returns
cell – The cell which was just processed.

Return type
NotebookNode

async_setup_kernel(***kwargs: Any*) → AsyncGenerator[None, None]

Context manager for setting up the kernel to execute a notebook.

This assigns the Kernel Manager (`self.km`) if missing and Kernel Client(`self.kc`).

When control returns from the yield it stops the client's zmq channels, and shuts down the kernel.

Handlers for SIGINT and SIGTERM are also added to cleanup in case of unexpected shutdown.

async async_start_new_kernel(***kwargs: Any*) → None

Creates a new kernel.

Parameters
kwargs – Any options for `self.kernel_manager_class.start_kernel()`. Because that defaults to AsyncKernelManager, this will likely include options accepted by `AsyncKernelManager.start_kernel()`, which includes `cwd`.

async async_start_new_kernel_client() → KernelClient

Creates a new kernel client.

Returns
kc – Kernel client as created by the kernel manager `km`.

Return type
KernelClient

async async_wait_for_reply(*msg_id: str*, *cell: NotebookNode | None = None*) → dict[str, Any] | None

Wait for a message reply.

clear_display_id_mapping(*cell_index: int*) → None

Clear a display id mapping for a cell.

clear_output(*outs: list[NotebookNode]*, *msg: dict[str, Any]*, *cell_index: int*) → None

Clear output.

coalesce_streams

Merge all stream outputs with shared names into single streams.

create_kernel_manager() → KernelManager

Creates a new kernel manager.

Returns
km – Kernel manager whose client class is asynchronous.

Return type
KernelManager

display_data_priority

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

error_on_timeout

If a cell execution was interrupted after a timeout, don't wait for the execute_reply from the kernel (e.g. KeyboardInterrupt error). Instead, return an execute_reply with the given error, which should be of the following form:

```
{
    'ename': str,  # Exception name, as a string
    'evalue': str,  # Exception value, as a string
    'traceback': list(str),  # traceback frames, as strings
}
```

execute(***kwargs: Any*) → Any

Executes each code cell.

Parameters
kwargs – Any option for `self.kernel_manager_class.start_kernel()`. Because that defaults to AsyncKernelManager, this will likely include options accepted by `jupyter_client.AsyncKernelManager.start_kernel()`, which includes `cwd`.

`reset_kc` if True, the kernel client will be reset and a new one will be created (default: False).

Returns
nb – The executed notebook.

Return type
NotebookNode

execute_cell(***kwargs: Any*) → Any

Executes a single code cell.

To execute all cells see *execute()*.

Parameters

- **cell** (*nbformat.NotebookNode*) – The cell which is currently being processed.
- **cell_index** (*int*) – The position of the cell within the notebook object.
- **execution_count** (*int*) – The execution count to be assigned to the cell (default: Use kernel response)
- **store_history** (*bool*) – Determines if history should be stored in the kernel (default: False). Specific to ipython kernels, which can store command histories.

Returns
output – The execution output payload (or None for no output).

Return type
dict

Raises
CellExecutionError – If execution failed and should raise an exception, this will be raised with defaults about the failure.

Returns
cell – The cell which was just processed.

Return type
NotebookNode

extra_arguments

An instance of a Python list.

force_raise_errors

If False (default), errors from executing the notebook can be allowed with a `raises-exception` tag on a single cell, or the `allow_errors` or `allow_error_names` configurable options for all cells. An allowed error will be recorded in notebook output, and execution will continue. If an error occurs when it is not explicitly allowed, a `CellExecutionError` will be raised. If True, `CellExecutionError` will be raised for any error that occurs while executing the notebook. This overrides the `allow_errors` and `allow_error_names` options and the `raises-exception` cell tag.

handle_comm_msg(*outs: list[NotebookNode]*, *msg: dict[str, Any]*, *cell_index: int*) → None

Handle a comm message.

interrupt_on_timeout

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

iopub_timeout

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

ipython_hist_file

Path to file to use for SQLite history database for an IPython kernel.

The specific value `:memory:` (including the colon at both end but not the back ticks), avoids creating a history file. Otherwise, IPython will create a history file for each kernel.

When running kernels simultaneously (e.g. via multiprocessing) saving history a single SQLite file can result in database errors, so using `:memory:` is recommended in non-interactive contexts.

kernel_manager_class

The kernel manager class to use.

kernel_name

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

on_cell_complete

A callable which executes after a cell execution is complete. It is called even when a cell results in a failure. Called with kwargs `cell` and `cell_index`.

on_cell_error

A callable which executes when a cell execution results in an error. This is executed even if errors are suppressed with `cell_allows_errors`. Called with kwargs `cell`, ``cell_index` and `execute_reply`.

on_cell_execute

A callable which executes just before a code cell is executed. Called with kwargs `cell` and `cell_index`.

on_cell_executed

A callable which executes just after a code cell is executed, whether or not it results in an error. Called with kwargs `cell`, `cell_index` and `execute_reply`.

on_cell_start

A callable which executes before a cell is executed and before non-executing cells are skipped. Called with kwargs `cell` and `cell_index`.

on_comm_open_jupyter_widget(*msg: dict[str, Any]*) → Any | None

Handle a jupyter widget comm open.

on_notebook_complete

A callable which executes after the kernel is cleaned up. Called with kwargs `notebook`.

on_notebook_error

A callable which executes when the notebook encounters an error. Called with kwargs `notebook`.

on_notebook_start

A callable which executes after the kernel manager and kernel client are setup, and cells are about to execute. Called with kwargs `notebook`.

output(*outs: list[NotebookNode]*, *msg: dict[str, Any]*, *display_id: str*, *cell_index: int*) → NotebookNode | None

Handle output.

process_message(*msg: dict[str, Any]*, *cell: NotebookNode*, *cell_index: int*) → NotebookNode | None

Processes a kernel message, updates cell state, and returns the resulting output object that was appended to cell.outputs.

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

Parameters

- **msg** (*dict*) – The kernel message being processed.
- **cell** (*nbformat.NotebookNode*) – The cell which is currently being processed.
- **cell_index** (*int*) – The position of the cell within the notebook object.

Returns

output – The execution output payload (or None for no output).

Return type

NotebookNode

Raises

CellExecutionComplete – Once a message arrives which indicates computation completeness.

raise_on_iopub_timeout

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

record_timing

If `True` (default), then the execution timings of each cell will be stored in the metadata of the notebook.

register_output_hook(*msg_id: str*, *hook: OutputWidget*) → None

Registers an override object that handles output/clear_output instead.

Multiple hooks can be registered, where the last one will be used (stack based)

remove_output_hook(*msg_id: str*, *hook: OutputWidget*) → None

Unregisters an override object that handles output/clear_output instead

reset_execution_trackers() → None

Resets any per-execution trackers.

resources: dict[str, Any]

Additional resources used in the conversion process. For example, passing `{'metadata': {'path': run_path}}` sets the execution path to `run_path`.

set_widgets_metadata() → None

Set with widget metadata.

setup_kernel(***kwargs: Any*) → Generator[None, None, None]

Context manager for setting up the kernel to execute a notebook.

The assigns the Kernel Manager (`self.km`) if missing and Kernel Client(`self.kc`).

When control returns from the yield it stops the client's zmq channels, and shuts down the kernel.

shell_timeout_interval

The time to wait (in seconds) for Shell output before retrying. This generally doesn't need to be set, but if one needs to check for dead kernels at a faster rate this can help.

shutdown_kernel

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

skip_cells_with_tag

Name of the cell tag to use to denote a cell that should be skipped.

start_new_kernel(***kwargs: Any*) → Any

Creates a new kernel.

Parameters

kwargs – Any options for `self.kernel_manager_class.start_kernel()`. Because that defaults to AsyncKernelManager, this will likely include options accepted by `AsyncKernelManager.start_kernel()`, which includes `cwd`.

start_new_kernel_client(***kwargs: Any*) → Any

Creates a new kernel client.

Returns
kc – Kernel client as created by the kernel manager `km`.

Return type
KernelClient

startup_timeout

The time to wait (in seconds) for the kernel to start. If kernel startup takes longer, a RuntimeError is raised.

store_widget_state

If `True` (default), then the state of the Jupyter widgets created at the kernel will be stored in the metadata of the notebook.

timeout

The time to wait (in seconds) for output from executions. If a cell execution takes longer, a TimeoutError is raised.

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

timeout_func: `Callable[[...], int | None] | None`

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

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

wait_for_reply(***kwargs: Any*) → Any

Wait for a message reply.

nbclient.client.**execute**(*nb: NotebookNode*, *cwd: str | None = None*, *km: KernelManager | None = None*, ***kwargs: Any*) → NotebookNode

Execute a notebook's code, updating outputs within the notebook object.

This is a convenient wrapper around NotebookClient. It returns the modified notebook object.

Parameters

- **nb** (*NotebookNode*) – The notebook object to be executed
- **cwd** (*str, optional*) – If supplied, the kernel will run in this directory
- **km** (*AsyncKernelManager, optional*) – If supplied, the specified kernel manager will be used for code execution.
- **kwargs** – Any other options for NotebookClient, e.g. timeout, kernel_name

nbclient.client.**timestamp**(*msg: dict[str, Any] | None = None*) → str

Get the timestamp for a message.

nbclient.exceptions module

Exceptions for nbclient.

exception nbclient.exceptions.**CellControlSignal**

Bases: Exception

A custom exception used to indicate that the exception is used for cell control actions (not the best model, but it's needed to cover existing behavior without major refactors).

exception nbclient.exceptions.**CellExecutionComplete**

Bases: *CellControlSignal*

Used as a control signal for cell execution across execute_cell and process_message function calls. Raised when all execution requests are completed and no further messages are expected from the kernel over zeromq channels.

exception nbclient.exceptions.**CellExecutionError**(*traceback: str*, *ename: str*, *evalue: str*)

Bases: *CellControlSignal*

Custom exception to propagate exceptions that are raised during notebook execution to the caller. This is mostly useful when using nbconvert as a library, since it allows to deal with failures gracefully.

classmethod from_cell_and_msg(*cell: NotebookNode*, *msg: dict[str, Any]*) → *CellExecutionError*

Instantiate from a code cell object and a message contents (message is either execute_reply or error)

exception nbclient.exceptions.**CellTimeoutError**

Bases: TimeoutError, *CellControlSignal*

A custom exception to capture when a cell has timed out during execution.

classmethod error_from_timeout_and_cell(*msg: str*, *timeout: int*, *cell: NotebookNode*) → *CellTimeoutError*

Create an error from a timeout on a cell.

exception nbclient.exceptions.**DeadKernelError**

Bases: RuntimeError

A dead kernel error.

Module contents

class nbclient.**NotebookClient**(***kwargs: Any*)

Bases: LoggingConfigurable

Encompasses a Client for executing cells in a notebook

allow_error_names

List of error names which won't stop the execution. Use this if the `allow_errors` option it too general and you want to allow only specific kinds of errors.

allow_errors

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

async async_execute(*reset_kc: bool = False*, ***kwargs: Any*) → NotebookNode

Executes each code cell.

Parameters

kwargs – Any option for `self.kernel_manager_class.start_kernel()`. Because that defaults to AsyncKernelManager, this will likely include options accepted by `jupyter_client.AsyncKernelManager.start_kernel()`, which includes `cwd`.

`reset_kc` if True, the kernel client will be reset and a new one will be created (default: False).

Returns

nb – The executed notebook.

Return type

NotebookNode

async async_execute_cell(*cell: NotebookNode*, *cell_index: int*, *execution_count: int | None = None*, *store_history: bool = True*) → NotebookNode

Executes a single code cell.

To execute all cells see *execute()*.

Parameters

- **cell** (*nbformat.NotebookNode*) – The cell which is currently being processed.
- **cell_index** (*int*) – The position of the cell within the notebook object.
- **execution_count** (*int*) – The execution count to be assigned to the cell (default: Use kernel response)
- **store_history** (*bool*) – Determines if history should be stored in the kernel (default: False). Specific to ipython kernels, which can store command histories.

Returns

output – The execution output payload (or None for no output).

Return type

dict

Raises

CellExecutionError – If execution failed and should raise an exception, this will be raised with defaults about the failure.

Returns

cell – The cell which was just processed.

Return type

NotebookNode

async_setup_kernel(***kwargs: Any*) → AsyncGenerator[None, None]

Context manager for setting up the kernel to execute a notebook.

This assigns the Kernel Manager (`self.km`) if missing and Kernel Client(`self.kc`).

When control returns from the yield it stops the client's zmq channels, and shuts down the kernel.

Handlers for SIGINT and SIGTERM are also added to cleanup in case of unexpected shutdown.

async async_start_new_kernel(***kwargs: Any*) → None

Creates a new kernel.

Parameters
kwargs – Any options for `self.kernel_manager_class.start_kernel()`. Because that defaults to AsyncKernelManager, this will likely include options accepted by `AsyncKernelManager.start_kernel()`, which includes `cwd`.

async async_start_new_kernel_client() → KernelClient

Creates a new kernel client.

Returns
kc – Kernel client as created by the kernel manager `km`.

Return type
KernelClient

async async_wait_for_reply(*msg_id: str*, *cell: NotebookNode | None = None*) → dict[str, Any] | None

Wait for a message reply.

clear_display_id_mapping(*cell_index: int*) → None

Clear a display id mapping for a cell.

clear_output(*outs: list[NotebookNode]*, *msg: dict[str, Any]*, *cell_index: int*) → None

Clear output.

coalesce_streams

Merge all stream outputs with shared names into single streams.

comm_open_handlers: dict[str, Any]

create_kernel_manager() → KernelManager

Creates a new kernel manager.

Returns
km – Kernel manager whose client class is asynchronous.

Return type
KernelManager

display_data_priority

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

error_on_timeout

If a cell execution was interrupted after a timeout, don't wait for the execute_reply from the kernel (e.g. KeyboardInterrupt error). Instead, return an execute_reply with the given error, which should be of the following form:

```
{
    'ename': str,  # Exception name, as a string
    'evalue': str,  # Exception value, as a string
    'traceback': list(str),  # traceback frames, as strings
}
```

execute(***kwargs: Any*) → Any

Executes each code cell.

Parameters
kwargs – Any option for `self.kernel_manager_class.start_kernel()`. Because that defaults to AsyncKernelManager, this will likely include options accepted by `jupyter_client.AsyncKernelManager.start_kernel()`, which includes `cwd`.

reset_kc if True, the kernel client will be reset and a new one will be created (default: False).

Returns
nb – The executed notebook.

Return type
NotebookNode

execute_cell(***kwargs: Any*) → Any

Executes a single code cell.

To execute all cells see *execute()*.

Parameters

- **cell** (*nbformat.NotebookNode*) – The cell which is currently being processed.
- **cell_index** (*int*) – The position of the cell within the notebook object.
- **execution_count** (*int*) – The execution count to be assigned to the cell (default: Use kernel response)
- **store_history** (*bool*) – Determines if history should be stored in the kernel (default: False). Specific to ipython kernels, which can store command histories.

Returns
output – The execution output payload (or None for no output).

Return type
dict

Raises
CellExecutionError – If execution failed and should raise an exception, this will be raised with defaults about the failure.

Returns
cell – The cell which was just processed.

Return type
NotebookNode

extra_arguments

An instance of a Python list.

force_raise_errors

If False (default), errors from executing the notebook can be allowed with a `raises-exception` tag on a single cell, or the `allow_errors` or `allow_error_names` configurable options for all cells. An allowed error will be recorded in notebook output, and execution will continue. If an error occurs when it is not explicitly allowed, a `CellExecutionError` will be raised. If True, `CellExecutionError` will be raised for any error that occurs while executing the notebook. This overrides the `allow_errors` and `allow_error_names` options and the `raises-exception` cell tag.

handle_comm_msg(*outs: list[NotebookNode]*, *msg: dict[str, Any]*, *cell_index: int*) → None

Handle a comm message.

interrupt_on_timeout

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

iopub_timeout

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

ipython_hist_file

Path to file to use for SQLite history database for an IPython kernel.

The specific value `:memory:` (including the colon at both end but not the back ticks), avoids creating a history file. Otherwise, IPython will create a history file for each kernel.

When running kernels simultaneously (e.g. via multiprocessing) saving history a single SQLite file can result in database errors, so using `:memory:` is recommended in non-interactive contexts.

kc: KernelClient | None

kernel_manager_class

The kernel manager class to use.

kernel_name

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

km: KernelManager | None

nb: NotebookNode

on_cell_complete

A callable which executes after a cell execution is complete. It is called even when a cell results in a failure. Called with kwargs `cell` and `cell_index`.

on_cell_error

A callable which executes when a cell execution results in an error. This is executed even if errors are suppressed with `cell_allows_errors`. Called with kwargs `cell`, ``cell_index`` and `execute_reply`.

on_cell_execute

A callable which executes just before a code cell is executed. Called with kwargs `cell` and `cell_index`.

on_cell_executed

A callable which executes just after a code cell is executed, whether or not it results in an error. Called with kwargs `cell`, `cell_index` and `execute_reply`.

on_cell_start

A callable which executes before a cell is executed and before non-executing cells are skipped. Called with kwargs `cell` and `cell_index`.

on_comm_open_jupyter_widget(*msg: dict[str, Any]*) → Any | None

Handle a jupyter widget comm open.

on_notebook_complete

A callable which executes after the kernel is cleaned up. Called with kwargs `notebook`.

on_notebook_error

A callable which executes when the notebook encounters an error. Called with kwargs `notebook`.

on_notebook_start

A callable which executes after the kernel manager and kernel client are setup, and cells are about to execute. Called with kwargs `notebook`.

output(*outs: list[NotebookNode]*, *msg: dict[str, Any]*, *display_id: str*, *cell_index: int*) → NotebookNode | None

Handle output.

owns_km: bool

process_message(*msg: dict[str, Any]*, *cell: NotebookNode*, *cell_index: int*) → NotebookNode | None

Processes a kernel message, updates cell state, and returns the resulting output object that was appended to cell.outputs.

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

Parameters

- **msg** (*dict*) – The kernel message being processed.
- **cell** (*nbformat.NotebookNode*) – The cell which is currently being processed.
- **cell_index** (*int*) – The position of the cell within the notebook object.

Returns

output – The execution output payload (or None for no output).

Return type

NotebookNode

Raises

CellExecutionComplete – Once a message arrives which indicates computation completeness.

raise_on_iopub_timeout

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

record_timing

If `True` (default), then the execution timings of each cell will be stored in the metadata of the notebook.

register_output_hook(*msg_id: str*, *hook: OutputWidget*) → None

Registers an override object that handles output/clear_output instead.

Multiple hooks can be registered, where the last one will be used (stack based)

remove_output_hook(*msg_id: str*, *hook: OutputWidget*) → None

Unregisters an override object that handles output/clear_output instead

reset_execution_trackers() → None

Resets any per-execution trackers.

resources: dict[str, Any]

Additional resources used in the conversion process. For example, passing `{'metadata': {'path': run_path}}` sets the execution path to `run_path`.

set_widgets_metadata() → None

Set with widget metadata.

setup_kernel(***kwargs: Any*) → Generator[None, None, None]

Context manager for setting up the kernel to execute a notebook.

The assigns the Kernel Manager (`self.km`) if missing and Kernel Client(`self.kc`).

When control returns from the yield it stops the client's zmq channels, and shuts down the kernel.

shell_timeout_interval

The time to wait (in seconds) for Shell output before retrying. This generally doesn't need to be set, but if one needs to check for dead kernels at a faster rate this can help.

shutdown_kernel

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

skip_cells_with_tag

Name of the cell tag to use to denote a cell that should be skipped.

start_new_kernel(***kwargs: Any*) → Any

Creates a new kernel.

Parameters
kwargs – Any options for `self.kernel_manager_class.start_kernel()`. Because that defaults to AsyncKernelManager, this will likely include options accepted by `AsyncKernelManager.start_kernel()`, which includes `cwd`.

start_new_kernel_client(***kwargs: Any*) → Any

Creates a new kernel client.

Returns
kc – Kernel client as created by the kernel manager `km`.

Return type
KernelClient

startup_timeout

The time to wait (in seconds) for the kernel to start. If kernel startup takes longer, a RuntimeError is raised.

store_widget_state

If `True` (default), then the state of the Jupyter widgets created at the kernel will be stored in the metadata of the notebook.

timeout

The time to wait (in seconds) for output from executions. If a cell execution takes longer, a TimeoutError is raised.

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

timeout_func: `Callable[[...], int | None] | None`

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

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

wait_for_reply(***kwargs: Any*) → Any

Wait for a message reply.

widget_registry: `dict[str, dict[str, Any]]`

nbclient.**execute**(*nb: NotebookNode*, *cwd: str | None = None*, *km: KernelManager | None = None*, ***kwargs: Any*) → NotebookNode

Execute a notebook's code, updating outputs within the notebook object.

This is a convenient wrapper around NotebookClient. It returns the modified notebook object.

Parameters

- **nb** (*NotebookNode*) – The notebook object to be executed
- **cwd** (*str, optional*) – If supplied, the kernel will run in this directory
- **km** (*AsyncKernelManager, optional*) – If supplied, the specified kernel manager will be used for code execution.
- **kwargs** – Any other options for NotebookClient, e.g. timeout, kernel_name

5.1.2 Config file and command line options

Jupyter `nbclient` can be run with a variety of command line arguments. A list of available options can be found below in the *options section*.

Options

This list of options can be generated by running the following and hitting enter:

```
$ jupyter execute --help-all
```

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

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

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

The Logging format template

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

Set the log level by value or name.

Application.logging_config
[Dict] Default: `{}`

Configure additional log handlers.

The default stderr logs handler is configured by the log_level, log_datefmt and log_format settings.

This configuration can be used to configure additional handlers (e.g. to output the log to a file) or for finer control over the default handlers.

If provided this should be a logging configuration dictionary, for more information see: https://docs.python.org/3/library/logging.config.html#logging-config-dictschema

This dictionary is merged with the base logging configuration which defines the following:

- A logging formatter intended for interactive use called `console`.
- A logging handler that writes to stderr called `console` which uses the formatter `console`.
- A logger with the name of this application set to `DEBUG` level.

This example adds a new handler that writes to a file:

```
c.Application.logging_config = {
    'handlers': {
        'file': {
            'class': 'logging.FileHandler',
            'level': 'DEBUG',
            'filename': '<path/to/file>',
        }
    },
    'loggers': {
        '<application-name>': {
            'level': 'DEBUG',
            # NOTE: if you don't list the default "console"
            # handler here then it will be disabled
            'handlers': ['console', 'file'],
        },
    }
}
```

Application.show_config
[Bool] Default: `False`

Instead of starting the Application, dump configuration to stdout

Application.show_config_json
[Bool] Default: `False`

Instead of starting the Application, dump configuration to stdout (as JSON)

JupyterApp.answer_yes
[Bool] Default: `False`

Answer yes to any prompts.

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

Full path of a config file.

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

Specify a config file to load.

JupyterApp.generate_config
[Bool] Default: `False`

Generate default config file.

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

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

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

The Logging format template

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

Set the log level by value or name.

JupyterApp.logging_config
[Dict] Default: {}

Configure additional log handlers.

The default stderr logs handler is configured by the log_level, log_datefmt and log_format settings.

This configuration can be used to configure additional handlers (e.g. to output the log to a file) or for finer control over the default handlers.

If provided this should be a logging configuration dictionary, for more information see: https://docs.python.org/3/library/logging.config.html#logging-config-dictschema

This dictionary is merged with the base logging configuration which defines the following:

- A logging formatter intended for interactive use called `console`.
- A logging handler that writes to stderr called `console` which uses the formatter `console`.
- A logger with the name of this application set to `DEBUG` level.

This example adds a new handler that writes to a file:

```
c.Application.logging_config = {
    'handlers': {
        'file': {
            'class': 'logging.FileHandler',
            'level': 'DEBUG',
            'filename': '<path/to/file>',
        }
    },
    'loggers': {
        '<application-name>': {
            'level': 'DEBUG',
            # NOTE: if you don't list the default "console"
            # handler here then it will be disabled
            'handlers': ['console', 'file'],
        },
    }
}
```

JupyterApp.show_config
[Bool] Default: `False`

Instead of starting the Application, dump configuration to stdout

JupyterApp.show_config_json
[Bool] Default: `False`

Instead of starting the Application, dump configuration to stdout (as JSON)

NbClientApp.allow_errors
[Bool] Default: `False`

When a cell raises an error the default behavior is that execution is stopped and a *nbclient.exceptions.CellExecutionError* is raised. If this flag is provided, errors are ignored and execution is continued until the end of the notebook.

NbClientApp.answer_yes
[Bool] Default: `False`

Answer yes to any prompts.

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

Full path of a config file.

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

Specify a config file to load.

NbClientApp.generate_config
[Bool] Default: `False`

Generate default config file.

NbClientApp.inplace
[Bool] Default: `False`

Default is execute notebook without writing the newly executed notebook. If this flag is provided, the newly generated notebook will overwrite the input notebook.

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

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

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

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

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

The Logging format template

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

Set the log level by value or name.

NbClientApp.logging_config
[Dict] Default: `{}`

Configure additional log handlers.

The default stderr logs handler is configured by the log_level, log_datefmt and log_format settings.

This configuration can be used to configure additional handlers (e.g. to output the log to a file) or for finer control over the default handlers.

If provided this should be a logging configuration dictionary, for more information see: https://docs.python.org/3/library/logging.config.html#logging-config-dictschema

This dictionary is merged with the base logging configuration which defines the following:

- A logging formatter intended for interactive use called `console`.
- A logging handler that writes to stderr called `console` which uses the formatter `console`.
- A logger with the name of this application set to `DEBUG` level.

This example adds a new handler that writes to a file:

```
c.Application.logging_config = {
    'handlers': {
        'file': {
            'class': 'logging.FileHandler',
            'level': 'DEBUG',
            'filename': '<path/to/file>',
        }
    },
    'loggers': {
        '<application-name>': {
            'level': 'DEBUG',
            # NOTE: if you don't list the default "console"
            # handler here then it will be disabled
            'handlers': ['console', 'file'],
        },
    }
}
```

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

Path of notebooks to convert

NbClientApp.output_base
[Unicode] Default: `None`

Write executed notebook to this file base name. Supports pattern replacements `'{notebook_name}'`, the name of the input notebook file without extension. Note that output is always relative to the parent directory of the input notebook.

NbClientApp.show_config
[Bool] Default: `False`

Instead of starting the Application, dump configuration to stdout

NbClientApp.show_config_json
[Bool] Default: `False`

Instead of starting the Application, dump configuration to stdout (as JSON)

NbClientApp.skip_cells_with_tag
[Unicode] Default: `'skip-execution'`

Name of the cell tag to use to denote a cell that should be skipped.

NbClientApp.startup_timeout
[Int] Default: `60`

The time to wait (in seconds) for the kernel to start. If kernel startup takes longer, a RuntimeError is raised.

NbClientApp.timeout
[Int] Default: `None`

The time to wait (in seconds) for output from executions. If a cell execution takes longer, a TimeoutError is raised. `-1` will disable the timeout.

5.1.3 nbclient

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

n

INDEX

P

R

S

T

W