papermill Documentation

Release 2.3.0

nteract team

Jan 20, 2021

Contents

Papermill is a tool for parameterizing and executing Jupyter Notebooks.

Papermill lets you:

- **parameterize** notebooks
- **execute** notebooks

This opens up new opportunities for how notebooks can be used. For example:

- Perhaps you have a financial report that you wish to run with different values on the first or last day of a month or at the beginning or end of the year, **using parameters** makes this task easier.
- Do you want to run a notebook and depending on its results, choose a particular notebook to run next? You can now programmatically **execute a workflow** without having to copy and paste from notebook to notebook manually.

CHAPTER 1

Python Version Support

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

CHAPTER 2

Documentation

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

2.1 Installation

2.1.1 Installing papermill

From the command line:

```
python3 -m pip install papermill
```

2.1.2 Installing In-Notebook language bindings

In-Notebook language bindings provide helpers and utilities for using Papermill with a programming language.

Python bindings

No additional installation steps are required since python bindings are built into **papermill**.

2.2 Change Log

2.2.1 2.3.0

- Notebooks that are loaded with papermill now upgrade the document to the latest spec version (to support cell-id assignments).
- Empty yaml files are now accepted as parameter files

- Binder typo fix for example notebook
- Entry point documentation improvements
- Code of Conduct documentation link cleanup
- Tox change for local doc builds

2.2.2 2.2.1

- Allow `pathlib.Paths` in `execute_notebook` and `inspect_notebook`

2.2.3 2.2.0

- Provide help for Python notebooks by inspecting the `parameters` cell, via `--help-notebook`
- Support added for parameterizing Powershell kernels

2.2.4 2.1.3

- Removed jupyter_client dependency in requirements to avoid confusing pip on the actual version requirements.
- Parameterized commenting so that once can pass a `comment` argument to assign the comment string in injected cells.

2.2.5 2.1.2

- Expand Usage Docs for JupyterLab
- Support `nan` and `inf` in Python translator
- Added fix for required async loop registration in python 38 on windows

2.2.6 2.1.1

- DeadKernelExceptions, usually from OOM, now exit with a status code of 138 from the CLI.
- Error cell at the top of failed notebook has been made better. It now also has a link to an injected cell where the error occurred.
- Updated a deprecated function to the new function name for nbclient dependency.
- Some development and documentation updates / fixes have also been made by a few different contributions (thank you!).

2.2.7 2.1.0

- Support for python 3.5 has been dropped. Upstream library changes for async were causing process deadlocks with await commands. End-of-life is later this year for 3.5 anyway so we decided to also drop support here.
- Error cells injected at the top of failed notebooks look nicer now as markdown.

2.2.8 2.0.0

Papermill 2.0 has a number of awesome features from many different contributors. We used the major version change mostly to signify the change to Python 3 only, but we also allowed for PRs which has small interaction changes to also be made. No major functionality should change with this release, but many minor improvements might impact specific execution patterns. We'll keep an eye on issues and post bug fixes ASAP if any of these cause larger unexpected issues.

Features

- Papermill is now Python 3.5+ only!
- nbconvert is no longer a dependency of papermill, instead the smaller and newly released nbclient is now the execution dependency.
- Support added for parameterizing C# kernels
- Support added for parameterizing F# kernels
- `sys.exit(0)` now respected by papermill
- Python parameters are now black formatted (in python versions >= 3.6)
- Notebook documents are saved periodically now rather than solely on cell completion.
- A cell `--execute-timeout` option was added.
- HDFS io support added with `hdfs://` scheme (with `papermill[hdfs]` install).

Fixes

- Fixed metadata writing on markdown and raw cells to follow v4.4 schema correctly
- Azure Blob Storage support fixed for newer blob storage. `azure-storage-blob >= 12.1.0` is now supported, older version support was dropped.
- IOPub timeouts now raise an exception instead of a warning.

Interaction Changes (more details)

- nbconvert dependency has been replaced with nbclient. This means the default engine is now `nbclient` rather than `nbconvert` and the NBConvertEngine class no longer exists. This may mean extensions that extended this class will need to be updated slightly to the new class.
- `sys.exit(0)` in python kernels now transfers exit code to papermill, meaning papermill will gracefully stop the notebook execution and not raise an exception to the user. `sys.exit(1)` or other exceptions still raise as expected and change the status code from 0 for the papermill process.
- When generating parameters for python (when running on 3.6+) the parameters will be printed more cleanly with a pass of black before injecting into the notebook.
- Older Azure Blob Storage support was dropped: `azure-storage-blob < 12.1.0`
- The `--autosave-cell-every` option now controls the minimum time between notebook saves during cell execution. This time will exponentially backoff if it takes more than 25% of the autosave-cell-every value. Setting `--autosave-cell-every` to `0` disabled this feature.
- The `--execute-timeout` option can be set to enable a per-cell execution timeout limit.

- IOPub timeouts used to only warn and attempt to continue execution. This can be triggered by printing '0' in a wide for-loop without any sleeps. The side-effect of best-effort execution was that outputs and failures could be lost in the IOPub timeout event and notebooks would "succeed" when they were actually failing. We chose to change this pattern from a warning to an error for papermill. To fix the issue when it occurs you need to delay the number of print or display messages per second being produced in your notebooks.

2.2.9 1.2.1

- Importing papermill no longer manipulates `yaml.SafeLoader` globally
- Parameters with leading _ now have prefix _ stripped before passing to R kernels
- A few documentation typos were fixed

2.2.10 1.2.0

- Parameters lists passing feature from 1.1.0 was removed due to cli api issues it caused.
- Piping papermill into nbconvert no longer triggers an encoding error on Python 2
- Added `BOTO3_ENDPOINT_URL` environment variable to override boto session url
- stdout / stderr can now be streamed to a file via `--stdout-file /dev/stdout` and `--stderr-file /dev/stderr`.
- The CLI option `--not-report-mode` is now `--no-report-mode`
- GCFS connectors should now retry under all conditions that the upstream library defines as retryable. Papermill now uses the is_retryable method from the upstream dependency.

2.2.11 1.1.0 (This version should be avoided for several known issues fixed in 1.2.0)

- Read content from stdin/to stdout when the path is - or a pipe. This allows for `<generate input>... | papermill | ...<process output>`, with `papermill - -` being implied by the pipes.
- The built-in `ADLHandler` for Azure Pipelines should now work properly again.
- Many documentation improvements
- IPython is now lazily imported only when progress bars are needed.
- A MATLAB translator is now available for parameters being passed to MATLAB notebooks.
- Parameters lists can more easily be passed to the command line via: `-p name value1 value2 3 ...` which results in adding to notebooks a parameter list assignment `name = ["value1", "value2", 3]`.

2.2.12 1.0.1

- Cleaned up some dependency and build issues around pip 19 and pandas
- `execute_notebook` can now take notebook as strings instead of only as a path
- `kwargs` are now passed through the default engine to nbconvert's wrapper class
- Passing dates through yaml as parameters will no longer raise an exception (i.e. `-y "a_date: 2019-01-01"` without having to quote ala `-y "a_date: '2019-01-01'"`)

2.2.13 1.0.0

We made it to our 1.0 milestone goals! The largest change here is removal of `record`, `Notebook`, and `NotebookCollection` abstractions which are now living in scrapbook and requirement of nbconvert 5.5 as a dependency.

- Input and output paths can now reference input parameters. `my_nb_{nb_type}.ipynb out_{nb_type}.ipynb -p nb_type test` will substitute values into the paths passed in with python format application patterns.
- `read_notebook`, `read_notebooks`, `record`, and `display` api functions are now removed.
- [upstream] ipywidgets are now supported. See nbconvert docs for details.
- [upstream] notebook executions which run out of memory no longer hang indefinitely when the kernel dies.

2.2.14 0.19.1

- Added a warning when no `parameter` tag is present but parameters are being passed
- Replaced `retry` with `tenacity` to help with conda builds and to use a non-abandoned library

2.2.15 0.19.0

DEPRECATION CHANGE The record, read_notebook, and read_notebooks functions are now officially deprecated and will be removed in papermill 1.0.

- scrapbook functionality is now deprecated
- gcsfs support is expanded to cover recent releases

2.2.16 0.18.2

Fixes

- Addressed an issue with reading encoded notebook .ipynb files in python 3.5

2.2.17 0.18.1

Fixes

- azure missing environment variable now has a better error message and only fails lazily
- gcs connector now has a backoff to respect service rate limits

2.2.18 0.18.0

INSTALL CHANGE The iorw extensions now use optional dependencies. This means that installation for s3, azure, and gcs connectors are added via:

```
pip install papermill[s3,azure,gcs]
```

or for all dependencies

```
pip install papermill[all]
```

Features

- Optional IO extensions are now separated into different dependencies.
- Added gs:// optional dependency for google cloud storage support.
- null json fields in parmaeters now translate correctly to equivilent fields in each supported language

Fixes

- tqdm dependencies are pinned to fetch a minimum version for auto tqdm

Dev Improvements

- Releases and versioning patterns were made easier
- Tox is now used to capture all test and build requirements

2.2.19 0.17.0

Features

- Log level can now be set with `--log-level`
- The working directory of papermill can be set with the `--cwd` option. This will set the executing context of the kernel but not impact input/output paths. `papermill --cwd foo bar/input_nb.ipynb bar/output_nb.ipynb` would make the notebook able to reference files in the `foo` directoy without `../foo` but still save the output notebook in the `bar` directory.
- Tox has been added for testing papermill. This makes it easier to catch linting and manifest issues without waiting for a failed Travis build.

Fixes

- Fixed warnings for reading non-ipynb files
- Fixed `--report-mode` with parameters (and made it more compatible with JupyterLab)
- Papermill execution progress bars now render within a notebook correctly after importing seaborn
- The `--prepare-only` option no longer requires that kernels be installed locally (you can parameterize a notebook without knowing how to execute it)
- Azure IO adapter now correctly prefixes paths with the `adl://` scheme
- Tests on OSX should pass again

Docs

- Install doc improvements
- Guide links are updated in the README
- Test docs updated for tox usage

2.2.20 0.16.2

- Injected parameter cells now respect `--report-mode`
- Logging level is only set for logger through CLI commands
- Output and input paths can be automatically passed to notebooks with the `--inject-paths` option
- Entrypoints have been added for registration of new `papermill.io` and `papermill.engine` plugins via setup files

2.2.21 0.16.1

- Fixed issue with azure blob io operations

2.2.22 0.16.0

- Added engines abstraction and command line argument
- Moved some nbconvert wrappers out of papermill
- Added Azure blob storage support
- Fixed botocore upgrade comptability issue (all version of boto now supported again)
- Removed whitelisted environment variable assignment

2.2.23 0.15.1

- Added support for Julia kernels
- Many improvements to README.md and documentation
- nbconvert dependency pinned to >= 5.3
- Improved error handling for missing directories
- Warnings added when an unexpected file extension is used
- Papermill version is visible to the CLI
- More messages us logging module now (and can be filtered accordingly)
- Binder link from README was greatly improved to demostrate papermill features

2.2.24 0.15.0

- Moved translator functions into registry
- Added development guide to help new contributors
- Travis, coverage, linting, and doc improvements
- Added support for Azure data lake sources
- Added python 3.7 testing

2.2.25 0.14.2

- Added output flushing for log-output option

2.2.26 0.14.1

- Upgraded executor to stream outputs during execution
- Fixed UTF-8 encoding issues for windows machines
- Added black code formatter rules (experimental)
- Contributors document added
- Added report-mode option for hiding inputs

2.2.27 0.13.4 (no code changes)

- Release manifest fix

2.2.28 0.13.3

- Fixed scala int vs long assignment

2.2.29 0.13.2

- Pip 10 fixes

2.2.30 0.13.1

- iPython pin to circumvent upstream issue

2.2.31 0.13.0

- Added prepare-only flag for parameterizing without processing a notebook
- Fixed cell number display on failed output notebooks
- Added scala language support

2.2.32 0.12.6

- Changed CLI outputs from papermill messaging to stderr
- Changed IOResolvers to perseve ordering of definition in resolving paths

2.2.33 0.12.5

- Set click disable_unicode_literals_warning=True to disable unicode literals

2.2.34 0.12.4

- Added universal wheel support
- Test coverage for s3 improved

2.2.35 0.12.3

- Added start timeout option for slow booting kernels

2.2.36 0.12.2

- Added options around tqdm
- Fixed an S3 decoding issue

2.2.37 0.12.1

- ip_display improvements
- Docstring improvements

2.2.38 0.12.0

- Added type preservation for r and python parameters
- Massive test coverage improvements
- Codebase style pass

2.3 Usage

For an interactive example that demonstrates the usage of papermill, click the Binder link below:

2.3.1 Using papermill

The general workflow when using papermill is **parameterizing** a notebook, **executing** it, as well as **storing** the results. In addition to operating on a single notebook, papermill also works on a collection of notebooks.

Parameterize

See also:

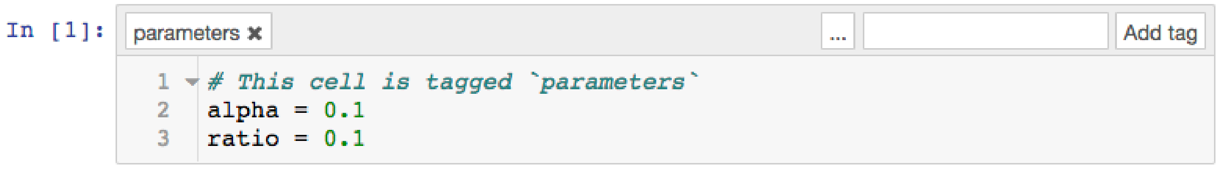
Workflow reference

Generally, the first workflow step when using papermill is to parameterize the notebook.

To do this, tag notebook cells with `parameters`. These `parameters` are later used when the notebook is executed or run.

Designate parameters for a cell

To parameterize a notebook, designate a cell with the tag `parameters`.



```python
# This cell is tagged `parameters`
alpha = 0.1
ratio = 0.1
```

Notebook

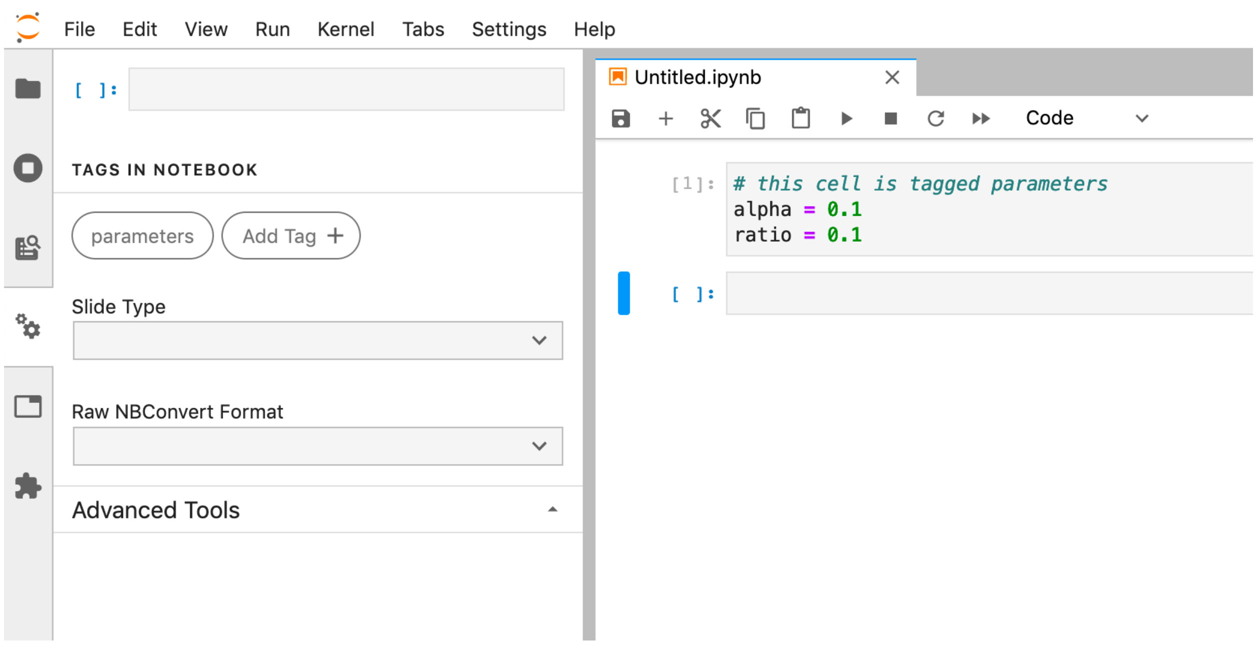
If using the Jupyter Notebook interface

1. Activate the tagging toolbar by navigating to `View`, `Cell Toolbar`, and then `Tags`
2. Enter `parameters` into a textbox at the top right of a cell
3. Click `Add tag`

JupyterLab 2.0+

If using the JupyterLab interface

1. Select the cell to parameterize
2. Click the property inspector on the left side (double gear icon)
3. Type "parameters" in the "Add Tag" box and hit "Enter".



```
[1]: # this cell is tagged parameters
alpha = 0.1
ratio = 0.1
```

JupyterLab < 2.0

If using JupyterLab < 2.0, consider using the jupyterlab-celltags extension.

1. Select the cell to parameterize
2. Click the cell inspector (wrench icon)
3. Type "parameters" in the "Add Tag" box and hit "Enter".

If the extension is not installed, you can manually add the tag by editing the Cell Metadata field in the cell inspector by adding "tags":["parameters"].

Learn more about the jupyter notebook format and metadata fields here.

How parameters work

The `parameters` cell is assumed to specify default values which may be overridden by values specified at execution time.

- papermill inserts a new cell tagged `injected-parameters` immediately after the `parameters` cell
- `injected-parameters` contains only the overridden parameters
- subsequent cells are treated as normal cells, even if also tagged `parameters`
- if no cell is tagged `parameters`, the `injected-parameters` cell is inserted at the top of the notebook

One caveat is that a `parameters` cell may not behave intuitively with inter-dependent parameters. Consider a notebook `note.ipynb` with two cells:

```
#parameters
a = 1
twice = a * 2
```

```
print("a =", a, "and twice =", twice)
```

when executed with `papermill note.ipynb -p a 9`, the output will be `a = 9 and twice = 2` (not `twice = 18`).

Inspect

The two ways to inspect the notebook to discover its parameters are: (1) through the Python API and (2) through the command line interface.

Execute via the Python API

The *inspect_notebook* function can be called to inspect a notebook:

```
inspect_notebook(<notebook path>)
```

```
import papermill as pm

pm.inspect_notebook('path/to/input.ipynb')
```

Note: If your path is parametrized, you can pass those parameters in a dictionary as second parameter:

`inspect_notebook('path/to/input_{month}.ipynb', parameters={month='Feb'})`

Inspect via CLI

To inspect a notebook using the CLI, enter the `papermill --help-notebook` command in the terminal with the notebook and optionally path parameters.

See also:

CLI reference

Inspect a notebook

Here's an example of a local notebook being inspected and an output example:

```
papermill --help-notebook ./papermill/tests/notebooks/complex_parameters.ipynb

Usage: papermill [OPTIONS] NOTEBOOK_PATH [OUTPUT_PATH]

Parameters inferred for notebook './papermill/tests/notebooks/complex_parameters.ipynb
↪':
msg: Unknown type (default None)
a: float (default 2.25)         Variable a
b: List[str] (default ['Hello','World'])
                                Nice list
c: NoneType (default None)
```

Execute

The two ways to execute the notebook with parameters are: (1) through the Python API and (2) through the command line interface.

Execute via the Python API

The *execute_notebook* function can be called to execute an input notebook when passed a dictionary of parameters:

```
execute_notebook(<input notebook>, <output notebook>, <dictionary of parameters>)
```

```
import papermill as pm

pm.execute_notebook(
   'path/to/input.ipynb',
   'path/to/output.ipynb',
   parameters=dict(alpha=0.6, ratio=0.1)
)
```

Execute via CLI

To execute a notebook using the CLI, enter the `papermill` command in the terminal with the input notebook, location for output notebook, and options.

See also:

CLI reference

Execute a notebook with parameters

Here's an example of a local notebook being executed and output to an Amazon S3 account:

```
$ papermill local/input.ipynb s3://bkt/output.ipynb -p alpha 0.6 -p l1_ratio 0.1
```

In the above example, two parameters are set: `alpha` and `l1_ratio` using `-p` (`--parameters` also works). Parameter values that look like booleans or numbers will be interpreted as such.

Here are the different ways users may set parameters:

Using raw strings as parameters

Using `-r` or `--parameters_raw`, users can set parameters one by one. However, unlike `-p`, the parameter will remain a string, even if it may be interpreted as a number or boolean.

```
$ papermill local/input.ipynb s3://bkt/output.ipynb -r version 1.0
```

Using a parameters file

Using `-f` or `--parameters_file`, users can provide a YAML file from which parameter values should be read.

```
$ papermill local/input.ipynb s3://bkt/output.ipynb -f parameters.yaml
```

Using a YAML string for parameters

Using `-y` or `--parameters_yaml`, users can directly provide a YAML string containing parameter values.

```
$ papermill local/input.ipynb s3://bkt/output.ipynb -y "
x:
    - 0.0
    - 1.0
    - 2.0
    - 3.0
linear_function:
    slope: 3.0
    intercept: 1.0"
```

Using `-b` or `--parameters_base64`, users can provide a YAML string, base64-encoded, containing parameter values.

```
$ papermill local/input.ipynb s3://bkt/output.ipynb -b␣
↪YWxwaGE6IDAuNgpsMV9yYXRpbzogMC4xCg==
```

Note about using YAML

When using YAML to pass arguments, through `-y`, `-b` or `-f`, parameter values can be arrays or dictionaries:

```
$ papermill local/input.ipynb s3://bkt/output.ipynb -y "
x:
    - 0.0
    - 1.0
    - 2.0
    - 3.0
linear_function:
    slope: 3.0
    intercept: 1.0"
```

Note about using with multiple account credentials

If you use multiple AWS accounts and are accessing S3 files, you can [configure your AWS credentials](https://boto3.amazonaws.com/v1/documentation/api/latest/guide/configuration.html), to specify which account to use by setting the *AWS_PROFILE* environment variable at the command-line. For example:

```
$ AWS_PROFILE=dev_account papermill local/input.ipynb s3://bkt/output.ipynb -p alpha␣
↪0.6 -p l1_ratio 0.1
```

A similar pattern may be needed for other types of remote storage accounts.

Store

See also:

Reference - Storage

Papermill can store notebooks in a number of locations including AWS S3, Azure data blobs, and Azure data lakes.

The modular architecture of papermill allows new data stores to be added over time.

See also:

Command Line Interface

papermill may be executed from the terminal. The following are the command options:

```
Usage: papermill [OPTIONS] NOTEBOOK_PATH [OUTPUT_PATH]

  This utility executes a single notebook in a subprocess.

  Papermill takes a source notebook, applies parameters to the source
  notebook, executes the notebook with the specified kernel, and saves the
  output in the destination notebook.

  The NOTEBOOK_PATH and OUTPUT_PATH can now be replaced by `-` representing
  stdout and stderr, or by the presence of pipe inputs / outputs. Meaning
  that

  `<generate input>... | papermill | ...<process output>`

  with `papermill - -` being implied by the pipes will read a notebook from
  stdin and write it out to stdout.

Options:
  --help-notebook                 Display parameters information for the given
                                  notebook path.

  -p, --parameters TEXT...        Parameters to pass to the parameters cell.
  -r, --parameters_raw TEXT...    Parameters to be read as raw string.
  -f, --parameters_file TEXT      Path to YAML file containing parameters.
  -y, --parameters_yaml TEXT      YAML string to be used as parameters.
  -b, --parameters_base64 TEXT    Base64 encoded YAML string as parameters.
  --inject-input-path             Insert the path of the input notebook as
                                  PAPERMILL_INPUT_PATH as a notebook
                                  parameter.

  --inject-output-path            Insert the path of the output notebook as
                                  PAPERMILL_OUTPUT_PATH as a notebook
                                  parameter.

  --inject-paths                  Insert the paths of input/output notebooks
                                  as
                                  PAPERMILL_INPUT_PATH/PAPERMILL_OUTPUT_PATH
                                  as notebook parameters.

  --engine TEXT                   The execution engine name to use in
                                  evaluating the notebook.

  --request-save-on-cell-execute / --no-request-save-on-cell-execute
                                  Request save notebook after each cell
                                  execution
```

(continues on next page)

(continued from previous page)

```
  --autosave-cell-every INTEGER   How often in seconds to autosave the
                                  notebook during long cell executions (0 to
                                  disable)

  --prepare-only / --prepare-execute
                                  Flag for outputting the notebook without
                                  execution, but with parameters applied.

  -k, --kernel TEXT               Name of kernel to run.
  --cwd TEXT                      Working directory to run notebook in.
  --progress-bar / --no-progress-bar
                                  Flag for turning on the progress bar.
  --log-output / --no-log-output  Flag for writing notebook output to the
                                  configured logger.

  --stdout-file FILENAME          File to write notebook stdout output to.
  --stderr-file FILENAME          File to write notebook stderr output to.
  --log-level [NOTSET|DEBUG|INFO|WARNING|ERROR|CRITICAL]
                                  Set log level
  --start-timeout, --start_timeout INTEGER
                                  Time in seconds to wait for kernel to start.
  --execution-timeout INTEGER     Time in seconds to wait for each cell before
                                  failing execution (default: forever)

  --report-mode / --no-report-mode
                                  Flag for hiding input.
  --version                       Flag for displaying the version.
  -h, --help                      Show this message and exit.
```

Extending papermill

Papermill provides some interfaces with external services out of the box. However, you may find that you would like papermill to do more than it currently does. You could contribute to the papermill project yourself (see *Extending papermill by contributing to it*). However, an easier method might be to extend papermill using entry points.

In general, when you run a notebook with papermill, the following happens:

1. The notebook file is read in
2. The file content is converted to a notebook python object
3. The notebook is executed
4. The notebook is written to a file

Through entry points, you can write your own tools to handle steps 1, 3, and 4. If you find that there's more you want to contribute to papermill, consider developing papermill itself.

Extending papermill through entry points

What are entry points?

The python packaging documentation describes entry points as:

> Entry points are a mechanism for an installed distribution to advertise components it provides to be discovered and used by other code. For example:

> Distributions can specify console_scripts entry points, each referring to a function. When pip (or another console_scripts aware installer) installs the distribution, it will create a command-line wrapper for each entry point.
>
> Applications can use entry points to load plugins; e.g. Pygments (a syntax highlighting tool) can use additional lexers and styles from separately installed packages. For more about this, see Creating and discovering plugins.

When running, papermill looks for entry points that implement input / output (I/O) handlers, and execution handlers.

Developing new I/O handlers

Virtually the first thing that happens when papermill is used is that the input notebook is read in. This is managed by I/O handlers, which allow papermill to access not just the local filesystem, but also remote services such as Amazon S3. The same goes for writing the executed notebook to a file system: I/O handlers allow papermill to write files to S3 or otherwise.

Creating a new handler

Writing your own I/O handler requires writing a class that has four methods. All I/O handlers should implement the following class methods:

- `CustomIO.read(file_path)`, returning the file content
- `CustomIO.write(file_content, file_path)`, returning nothing
- `CustomIO.pretty_path(path)`, returning a prettified path
- `CustomIO.listdir(path)`, returning a list of paths.

Note: If you don't want to support things such as `read` because your I/O handler is only intended for writing (such as a publish-only platform), then you should implement the method but raise an exception when it is used.

Ensuring your handler is found by papermill

Once you have developed a new handler, you need to declare papermill entry points in your `setup.py` file.

This is done by including the `entry_points` key-word argument to `setup` in your setup.py file:

```python
from setuptools import setup, find_packages
setup(
    # all the normal setup.py arguments...
    entry_points={"papermill.io": ["sftp://=papermill_sftp:SFTPHandler"]},
)
```

This indicates to papermill that when a file path begins with `sftp://`, it should use the class `papermill_sftp.SFTPHandler` to handle reading or writing to that path. Anything before the equal sign is the path prefix, and everything after it is the class to be used, including where it is imported from.

Traditionally, entry points for papermill I/O handlers look like URL prefixes. For example, the Amazon Web Services S3 handler is registered under `s3://`, and so is used whenever a path begins with `s3://`.

Example: sftp I/O handler

As an example, let's go through how we would create an I/O handler that reads from an sftp server and writes back to it, so we could do the following:

```
papermill sftp://my_ftp_server.co.uk/input.ipynb sftp://my_ftp_server.co.uk/output.
↪ipynb
```

Our project structure will look like this:

```
papermill_sftp
    |- setup.py
    |- src
        |- papermill_sftp
            |- __init__.py
```

We can define the I/O handler in `src/papermill_sftp/__init__.py`. To do so, we have to create a class that does the relevant actions.

For reading, we will download the file to a temporary path and read it in from there. For writing, we will write to a temporary path and upload it from there. Prettifying the path doesn't need to change the path, and we are not going to implement a listdir option for now.

```
import os
import pysftp

sftp_username = os.getenv('SFTP_USERNAME')
sftp_password = os.getenv('SFTP_PASSWORD')

class SFTPHandler:

    @classmethod
    def read(cls, path):
        """
        Read a notebook from an SFTP server.
        """
        parsed_url = urllib.parse.urlparse(path)
        with tempfile.TemporaryDirectory() as tmpdir:
            tmp_file = pathlib.Path(tmpdir) / pathlib.Path(parsed_url.path).name
            with pysftp.Connection(
                parsed_url.hostname,
                username=sftp_username,
                password=sftp_password,
                port=(parsed_url.port or 22),
                cnopts=cnopts,
            ) as sftp:
                sftp.get(parsed_url.path, str(tmp_file))
            return tmp_file.read_text()

    @classmethod
    def write(cls, file_content, path):
        """
        Write a notebook to an SFTP server.
        """
        parsed_url = urllib.parse.urlparse(path)
        with tempfile.TemporaryDirectory() as tmpdir:
            tmp_file = pathlib.Path(tmpdir) / "output.ipynb"
```

(continues on next page)

(continued from previous page)

```
            tmp_file.write_text(file_content)
            with pysftp.Connection(
                parsed_url.hostname,
                username=sftp_username,
                password=sftp_password,
                port=(parsed_url.port or 22),
                cnopts=cnopts,
            ) as sftp:
                sftp.put(str(tmp_file), parsed_url.path)

    @classmethod
    def pretty_path(cls, path):
        return path

    @classmethod
    def listdir(cls, path):
        raise NotImplementedError
```

The `setup.py` file contains the following code:

```
from setuptools import setup, find_packages

setup(
    name="papermill_sftp",
    version="0.1",
    url="https://github.com/my_username/papermill_sftp.git",
    author="My Name",
    author_email="my.email@gmail.com",
    description="An SFTP I/O handler for papermill.",
    packages=find_packages("./src"),
    package_dir={"": "src"},
    install_requires=["pysftp"],
    entry_points={"papermill.io": ["sftp://=papermill_sftp:SFTPHandler"]},
)
```

When executing, papermill will check if the input or output path begin with `sftp://`, and if so, use the SFTPHandler from the papermill_sftp project.

Developing a new engine

A papermill engine is a python object that can run, or execute, a notebook. The default implementation in papermill for example takes in a notebook object, and runs it locally on your machine.

By writing a custom engine, you could allow execution to be handled remotely, or you could apply post-processing to the executed notebook. In the next section, you will see a demonstration.

Creating a new engine

Papermill engines need to inherit from the `papermill.engines.Engine` class.

In order to be used, the new class needs to implement the class method `execute_managed_notebook`. The call signature should match that of the parent class:

```
class CustomEngine(papermill.engines.Engine):

    @classmethod
    execute_managed_notebook(cls, nb_man, kernel_name, **kwargs):
        pass
```

nb_man is a nbformat.NotebookNode object, and kernel_name is a string. Your custom class then needs to implement the execution of the notebook. For example, you could insert code that executes the notebook remotely on a server, or executes the notebook many times to simulate different conditions.

As an example, the following project implements a custom engine that adds the time it took to execute each cell as additional output after every code cell.

The project structure is:

```
papermill_timing
    |- setup.py
    |- src
        |- papermill_timing
            |- __init__.py
```

The file src/papermill_timing/__init__.py will implement the engine. Since papermill already stores information about execution timing in the metadata, we can leverage the default engine. We will also need to use the nbformat library to create a notebook node object.

```
from datetime import datetime
from papermill.engines import NBClientEngine
from nbformat.v4 import new_output

class CustomEngine(NBClientEngine):

    @classmethod
    def execute_managed_notebook(cls, nb_man, kernel_name, **kwargs):

        # call the papermill execution engine:
        super().execute_managed_notebook(nb_man, kernel_name, **kwargs)

        for cell in nb_man.nb.cells:

            if cell.cell_type == "code" and cell.execution_count is not None:
                start = datetime.fromisoformat(cell.metadata.papermill.start_time)
                end = datetime.fromisoformat(cell.metadata.papermill.end_time)
                output_message = f"Execution took {(end - start).total_seconds():.3f}␣
↪seconds"
                output_node = new_output("display_data", data={"text/plain": [output_
↪message]})
                cell.outputs = [output_node] + cell.outputs
```

Once this is in place, we need to add our engine as an entry point to our setup.py script - for this, see the following section.

Ensuring your engine is found by papermill

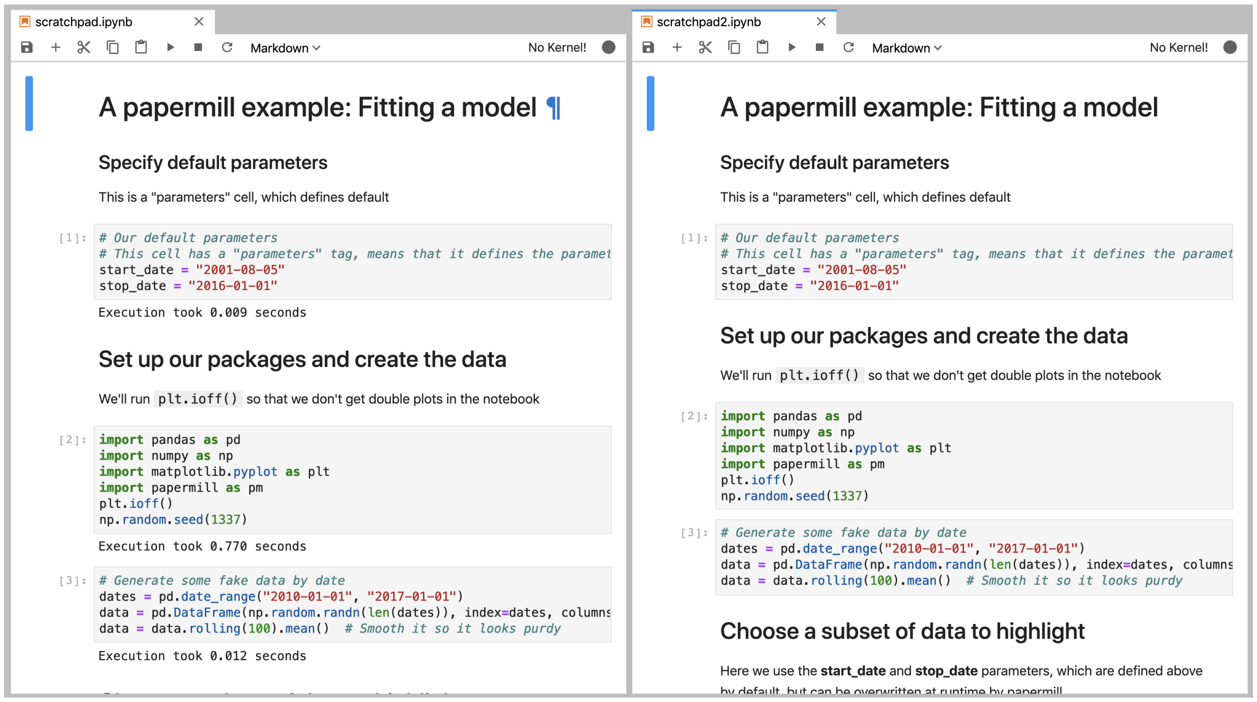
Custom engines can be specified as entry points, under the papermill.engine prefix. The entry point needs to reference the class that we have just implemented. For example, if you write an engine called TimingEngine in a package called papermill_timing, then in the setup.py file, you should specify:

```
from setuptools import setup, find_packages

setup(
    name="papermill_timing",
    version="0.1",
    url="https://github.com/my_username/papermill_timing.git",
    author="My Name",
    author_email="my.email@gmail.com",
    description="A papermill engine that logs additional timing information about␣
↪code.",
    packages=find_packages("./src"),
    package_dir={"": "src"},
    install_requires=["papermill", "nbformat"],
    entry_points={"papermill.engine": ["timer_engine=papermill_timing:CustomEngine"]},
)
```

This allows users to specify the engine from `papermill_timing` by passing the command line argument `--engine timer_engine`.

In the image below, the notebook on the left was executed with the new custom engine, while the one on the left was executed with the standard papermill engine. As you can see, this adds our "injected" output to each code cell



A papermill example: Fitting a model ¶

Specify default parameters

This is a "parameters" cell, which defines default

```
[1]: # Our default parameters
     # This cell has a "parameters" tag, means that it defines the paramet
     start_date = "2001-08-05"
     stop_date = "2016-01-01"

     Execution took 0.009 seconds
```

Set up our packages and create the data

We'll run `plt.ioff()` so that we don't get double plots in the notebook

```
[2]: import pandas as pd
     import numpy as np
     import matplotlib.pyplot as plt
     import papermill as pm
     plt.ioff()
     np.random.seed(1337)

     Execution took 0.770 seconds

[3]: # Generate some fake data by date
     dates = pd.date_range("2010-01-01", "2017-01-01")
     data = pd.DataFrame(np.random.randn(len(dates)), index=dates, columns
     data = data.rolling(100).mean()  # Smooth it so it looks purdy

     Execution took 0.012 seconds
```

A papermill example: Fitting a model

Specify default parameters

This is a "parameters" cell, which defines default

```
[1]: # Our default parameters
     # This cell has a "parameters" tag, means that it defines the paramet
     start_date = "2001-08-05"
     stop_date = "2016-01-01"
```

Set up our packages and create the data

We'll run `plt.ioff()` so that we don't get double plots in the notebook

```
[2]: import pandas as pd
     import numpy as np
     import matplotlib.pyplot as plt
     import papermill as pm
     plt.ioff()
     np.random.seed(1337)

[3]: # Generate some fake data by date
     dates = pd.date_range("2010-01-01", "2017-01-01")
     data = pd.DataFrame(np.random.randn(len(dates)), index=dates, columns
     data = data.rolling(100).mean()  # Smooth it so it looks purdy
```

Choose a subset of data to highlight

Here we use the **start_date** and **stop_date** parameters, which are defined above

Extending papermill by contributing to it

If you find that you'd like to not only add I/O and execution handlers, but think a fundamental aspect of the project could use some improvement, then you may want to contribute to it.

Development of papermill happens on github, and a detailed guide to contributing to it can be found there. There is also a code of conduct there. Please read both documents before beginning!

Troubleshooting

NoSuchKernel Errors (using Conda)

`NoSuchKernel` Errors can appear when running papermill on jupyter notebooks whose kernel has been specified via conda (nb_conda). nb_conda is used to easily set conda environment per notebook from within jupyterlab.

To illustrate, the following example demonstrates the creation of a new environment with all the dependencies necessary for an analysis.

```
conda create -n analysis_1 python=2 ipykernel
```

Once nb_conda is used within the jupyter server to set the kernel for a notebook to *analysis_1*, the notebook gets metadata similar to the following:

```
{
 "kernelspec": {
  "display_name": "Python [conda env:analysis_1]",
  "language": "python",
  "name": "conda-env-analysis_1-py"
 }
}
```

Papermill cannot use this metadata to determine that it should use *analysis_1* to execute this notebook. Running papermill (from *analysis_1* or another environment) will raise the following error:

```
jupyter_client.kernelspec.NoSuchKernel: No such kernel named conda-env-analysis_1-py
```

This can be fixed by:

- Installing jupyter (or at least ipykernel) in *analysis_1*

```
conda install -n analysis_1 jupyter
```

- Expose the *analysis_1* environment as a jupyter kernel (this is no longer automatic).

```
conda activate analysis_1
jupyter kernelspec install --user --name analysis_1
```

- Run papermill (from any environment) specifying the correct kernel using the `-k` option

```
papermill my_notebook.ipynb output_notebook.ipynb -k analysis_1
```

CHAPTER 3

API Reference

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

3.1 Reference

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

3.1.1 CLI

papermill.cli

Main *papermill* interface.

papermill.cli.**print_papermill_version**(*ctx*, *param*, *value*)

Command Line options

```
Usage: papermill [OPTIONS] NOTEBOOK_PATH OUTPUT_PATH

  This utility executes a single notebook in a subprocess.

  Papermill takes a source notebook, applies parameters to the source
  notebook, executes the notebook with the specified kernel, and saves the
  output in the destination notebook.

  The NOTEBOOK_PATH and OUTPUT_PATH can now be replaced by `-` representing
  stdout and stderr, or by the presence of pipe inputs / outputs. Meaning
  that
```

(continues on next page)

(continued from previous page)

```
  `<generate input>... | papermill | ...<process output>`

  with `papermill - -` being implied by the pipes will read a notebook from
  stdin and write it out to stdout.

Options:
  -p, --parameters TEXT...        Parameters to pass to the parameters cell.
  -r, --parameters_raw TEXT...    Parameters to be read as raw string.
  -f, --parameters_file TEXT      Path to YAML file containing parameters.
  -y, --parameters_yaml TEXT      YAML string to be used as parameters.
  -b, --parameters_base64 TEXT    Base64 encoded YAML string as parameters.
  --inject-input-path             Insert the path of the input notebook as
                                  PAPERMILL_INPUT_PATH as a notebook
                                  parameter.
  --inject-output-path            Insert the path of the output notebook as
                                  PAPERMILL_OUTPUT_PATH as a notebook
                                  parameter.
  --inject-paths                  Insert the paths of input/output notebooks
                                  as
                                  PAPERMILL_INPUT_PATH/PAPERMILL_OUTPUT_PATH
                                  as notebook parameters.
  --engine TEXT                   The execution engine name to use in
                                  evaluating the notebook.
  --request-save-on-cell-execute / --no-request-save-on-cell-execute
                                  Request save notebook after each cell
                                  execution
  --prepare-only / --prepare-execute
                                  Flag for outputting the notebook without
                                  execution, but with parameters applied.
  -k, --kernel TEXT               Name of kernel to run.
  --cwd TEXT                      Working directory to run notebook in.
  --progress-bar / --no-progress-bar
                                  Flag for turning on the progress bar.
  --log-output / --no-log-output  Flag for writing notebook output to the
                                  configured logger.
  --stdout-file FILENAME          File to write notebook stdout output to.
  --stderr-file FILENAME          File to write notebook stderr output to.
  --log-level [NOTSET|DEBUG|INFO|WARNING|ERROR|CRITICAL]
                                  Set log level
  --start_timeout INTEGER         Time in seconds to wait for kernel to start.
  --execution_timeout INTEGER     Time in seconds to wait for each cell before
                                  failing execution (default: forever)
  --report-mode / --no-report-mode
                                  Flag for hiding input.
  --version                       Flag for displaying the version.
  -h, --help                      Show this message and exit.
```

3.1.2 Workflow

papermill.engines

Engines to perform different roles

class papermill.engines.**Engine**

Bases: object

Base class for engines.

Other specific engine classes should inherit and implement the *execute_managed_notebook* method.

Defines *execute_notebook* method which is used to correctly setup the *NotebookExecutionManager* object for engines to interact against.

classmethod execute_managed_notebook(*nb_man*, *kernel_name*, ***kwargs*)
An abstract method where implementation will be defined in a subclass.

classmethod execute_notebook(*nb*, *kernel_name*, *output_path=None*, *progress_bar=True*, *log_output=False*, *autosave_cell_every=30*, ***kwargs*)
A wrapper to handle notebook execution tasks.

Wraps the notebook object in a *NotebookExecutionManager* in order to track execution state in a uniform manner. This is meant to help simplify engine implementations. This allows a developer to just focus on iterating and executing the cell contents.

class `papermill.engines.`**NBClientEngine**
Bases: `papermill.engines.Engine`

A notebook engine representing an nbclient process.

This can execute a notebook document and update the *nb_man.nb* object with the results.

classmethod execute_managed_notebook(*nb_man*, *kernel_name*, *log_output=False*, *stdout_file=None*, *stderr_file=None*, *start_timeout=60*, *execution_timeout=None*, ***kwargs*)
Performs the actual execution of the parameterized notebook locally.

Parameters

- **nb** (*NotebookNode*) – Executable notebook object.
- **kernel_name** (*str*) – Name of kernel to execute the notebook against.
- **log_output** (*bool*) – Flag for whether or not to write notebook output to the configured logger.
- **start_timeout** (*int*) – Duration to wait for kernel start-up.
- **execution_timeout** (*int*) – Duration to wait before failing execution (default: never).

class `papermill.engines.`**NotebookExecutionManager**(*nb*, *output_path=None*, *log_output=False*, *progress_bar=True*, *autosave_cell_every=30*)
Bases: `object`

Wrapper for execution state of a notebook.

This class is a wrapper for notebook objects to house execution state related to the notebook being run through an engine.

In particular the NotebookExecutionManager provides common update callbacks for use within engines to facilitate metadata and persistence actions in a shared manner.

COMPLETED = 'completed'

FAILED = 'failed'

PENDING = 'pending'

RUNNING = 'running'

autosave_cell()
Saves the notebook if it's been more than self.autosave_cell_every seconds since it was last saved.

cell_complete(*cell*, *cell_index=None*, ***kwargs*)
Finalize metadata for a cell and save notebook.

Optionally called by engines during execution to finalize the metadata for a cell and save the notebook to the output path.

cell_exception(*cell*, *cell_index=None*, ***kwargs*)
Set metadata when an exception is raised.

Called by engines when an exception is raised within a notebook to set the metadata on the notebook indicating the location of the failure.

cell_start(*cell*, *cell_index=None*, ***kwargs*)
Set and save a cell's start state.

Optionally called by engines during execution to initialize the metadata for a cell and save the notebook to the output path.

cleanup_pbar()
Clean up a progress bar

complete_pbar()
Refresh progress bar

notebook_complete(***kwargs*)
Finalize the metadata for a notebook and save the notebook to the output path.

Called by Engine when execution concludes, regardless of exceptions.

notebook_start(***kwargs*)
Initialize a notebook, clearing its metadata, and save it.

When starting a notebook, this initializes and clears the metadata for the notebook and its cells, and saves the notebook to the given output path.

Called by Engine when execution begins.

now()
Helper to return current UTC time

save(***kwargs*)
Saves the wrapped notebook state.

If an output path is known, this triggers a save of the wrapped notebook state to the provided path.

Can be used outside of cell state changes if execution is taking a long time to conclude but the notebook object should be synced.

For example, you may want to save the notebook every 10 minutes when running a 5 hour cell execution to capture output messages in the notebook.

set_timer()
Initializes the execution timer for the notebook.

This is called automatically when a NotebookExecutionManager is constructed.

class papermill.engines.**PapermillEngines**
Bases: `object`

The holder which houses any engine registered with the system.

This object is used in a singleton manner to save and load particular named Engine objects so they may be referenced externally.

execute_notebook_with_engine(*engine_name*, *nb*, *kernel_name*, ***kwargs*)
Fetch a named engine and execute the nb object against it.

get_engine(*name=None*)
Retrieves an engine by name.

register(*name*, *engine*)
Register a named engine

register_entry_points()
Register entrypoints for an engine

Load handlers provided by other packages

papermill.engines.**catch_nb_assignment**(*func*)
Wrapper to catch *nb* keyword arguments

This helps catch *nb* keyword arguments and assign onto self when passed to the wrapped function.

Used for callback methods when the caller may optionally have a new copy of the originally wrapped *nb* object.

papermill.execute

papermill.execute.**execute_notebook**(*input_path*, *output_path*, *parameters=None*, *engine_name=None*, *request_save_on_cell_execute=True*, *prepare_only=False*, *kernel_name=None*, *progress_bar=True*, *log_output=False*, *stdout_file=None*, *stderr_file=None*, *start_timeout=60*, *report_mode=False*, *cwd=None*, ***engine_kwargs*)
Executes a single notebook locally.

Parameters

- **input_path** (*str or Path*) – Path to input notebook
- **output_path** (*str or Path*) – Path to save executed notebook
- **parameters** (*dict, optional*) – Arbitrary keyword arguments to pass to the notebook parameters
- **engine_name** (*str, optional*) – Name of execution engine to use
- **request_save_on_cell_execute** (*bool, optional*) – Request save notebook after each cell execution
- **autosave_cell_every** (*int, optional*) – How often in seconds to save in the middle of long cell executions
- **prepare_only** (*bool, optional*) – Flag to determine if execution should occur or not
- **kernel_name** (*str, optional*) – Name of kernel to execute the notebook against
- **progress_bar** (*bool, optional*) – Flag for whether or not to show the progress bar.
- **log_output** (*bool, optional*) – Flag for whether or not to write notebook output to the configured logger
- **start_timeout** (*int, optional*) – Duration in seconds to wait for kernel start-up
- **report_mode** (*bool, optional*) – Flag for whether or not to hide input.

- **cwd** (*str or Path, optional*) – Working directory to use when executing the notebook
- ****kwargs** – Arbitrary keyword arguments to pass to the notebook engine

Returns **nb** – Executed notebook object

Return type NotebookNode

`papermill.execute.`**`prepare_notebook_metadata`**(*nb*, *input_path*, *output_path*, *report_mode=False*)

Prepare metadata associated with a notebook and its cells

Parameters

- **nb** (*NotebookNode*) – Executable notebook object
- **input_path** (*str*) – Path to input notebook
- **output_path** (*str*) – Path to write executed notebook
- **report_mode** (*bool, optional*) – Flag to set report mode

`papermill.execute.`**`raise_for_execution_errors`**(*nb*, *output_path*)

Assigned parameters into the appropriate place in the input notebook

Parameters

- **nb** (*NotebookNode*) – Executable notebook object
- **output_path** (*str*) – Path to write executed notebook

`papermill.execute.`**`remove_error_markers`**(*nb*)

papermill.clientwrap

class `papermill.clientwrap.`**`PapermillNotebookClient`**(*nb_man*, *km=None*, *raise_on_iopub_timeout=True*, ***kw*)

Bases: `nbclient.client.NotebookClient`

Module containing a that executes the code cells and updates outputs

`execute`(***kwargs*)

Wraps the parent class process call slightly

`log_output`

A boolean (True, False) trait.

`log_output_message`(*output*)

Process a given output. May log it in the configured logger and/or write it into the configured stdout/stderr files.

Parameters **output** – nbformat.notebooknode.NotebookNode

Returns

`papermill_execute_cells`()

This function replaces cell execution with it's own wrapper.

We are doing this for the following reasons:

1. Notebooks will stop executing when they encounter a failure but not raise a *CellException*. This allows us to save the notebook with the traceback even though a *CellExecutionError* was encountered.
2. We want to write the notebook as cells are executed. We inject our logic for that here.

3. We want to include timing and execution status information with the metadata of each cell.

`process_message`(**arg*, ***kwargs*)
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 dict

Raises `CellExecutionComplete` – Once a message arrives which indicates computation completeness.

`stderr_file`
A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

Subclasses can declare default classes by overriding the klass attribute

`stdout_file`
A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

Subclasses can declare default classes by overriding the klass attribute

3.1.3 Language Translators

Translators

Translator

class `papermill.translators.`**`Translator`**

classmethod **`assign`**(*name*, *str_val*)

classmethod **`codify`**(*parameters*, *comment='Parameters'*)

classmethod **`comment`**(*cmt_str*)

classmethod **`inspect`**(*parameters_cell*)
Inspect the parameters cell to get a Parameter list

It must return an empty list if no parameters are found and it should ignore inspection errors.

Note: `inferred_type_name` should be "None" if unknown (set it to "NoneType" for null value)

Parameters **`parameters_cell`** (*`NotebookNode`*) – Cell tagged _parameters_

Returns A list of all parameters

Return type List[Parameter]

classmethod translate(*val*)
Translate each of the standard json/yaml types to appropiate objects.

classmethod translate_bool(*val*)
Default behavior for translation

classmethod translate_dict(*val*)

classmethod translate_escaped_str(*str_val*)
Reusable by most interpreters

classmethod translate_float(*val*)
Default behavior for translation

classmethod translate_int(*val*)
Default behavior for translation

classmethod translate_list(*val*)

classmethod translate_none(*val*)
Default behavior for translation

classmethod translate_raw_str(*val*)
Reusable by most interpreters

classmethod translate_str(*val*)
Default behavior for translation

PapermillTranslators

class papermill.translators.**PapermillTranslators**
The holder which houses any translator registered with the system. This object is used in a singleton manner to save and load particular named Translator objects for reference externally.

find_translator(*kernel_name*, *language*)

register(*language*, *translator*)

Python

class papermill.translators.**PythonTranslator**

PARAMETER_PATTERN = re.compile('^(?P<target>\\w[\\w_]*)\\s*(:\\s*[\\"\']?(?P<annotatio

classmethod codify(*parameters*, *comment='Parameters'*)

classmethod comment(*cmt_str*)

classmethod inspect(*parameters_cell*)
Inspect the parameters cell to get a Parameter list

It must return an empty list if no parameters are found and it should ignore inspection errors.

Parameters **parameters_cell** (*NotebookNode*) – Cell tagged _parameters_

Returns A list of all parameters

Return type List[Parameter]

classmethod translate_bool(*val*)
Default behavior for translation

classmethod translate_dict(*val*)

classmethod translate_float(*val*)
Default behavior for translation

classmethod translate_list(*val*)

R

class papermill.translators.**RTranslator**

classmethod assign(*name*, *str_val*)

classmethod comment(*cmt_str*)

classmethod translate_bool(*val*)
Default behavior for translation

classmethod translate_dict(*val*)

classmethod translate_list(*val*)

classmethod translate_none(*val*)
Default behavior for translation

Julia

class papermill.translators.**JuliaTranslator**

classmethod comment(*cmt_str*)

classmethod translate_dict(*val*)

classmethod translate_list(*val*)

classmethod translate_none(*val*)
Default behavior for translation

Scala

class papermill.translators.**ScalaTranslator**

classmethod assign(*name*, *str_val*)

classmethod comment(*cmt_str*)

classmethod translate_dict(*val*)
Translate dicts to scala Maps

classmethod translate_int(*val*)
Default behavior for translation

classmethod translate_list(*val*)
Translate list to scala Seq

Functions

`papermill.translators.`**`translate_parameters`**(*kernel_name*, *language*, *parameters*, *comment='Parameters'*)

3.1.4 Input / Output

papermill.iorw

class `papermill.iorw.`**`ABSHandler`**

Bases: `object`

`listdir`(*path*)

`pretty_path`(*path*)

`read`(*path*)

`write`(*buf*, *path*)

class `papermill.iorw.`**`ADLHandler`**

Bases: `object`

`listdir`(*path*)

`pretty_path`(*path*)

`read`(*path*)

`write`(*buf*, *path*)

class `papermill.iorw.`**`GCSHandler`**

Bases: `object`

`RATE_LIMIT_RETRIES = 3`

`RETRY_DELAY = 1`

`RETRY_MAX_DELAY = 4`

`RETRY_MULTIPLIER = 1`

`listdir`(*path*)

`pretty_path`(*path*)

`read`(*path*)

`write`(*buf*, *path*)

class `papermill.iorw.`**`HDFSHandler`**

Bases: `object`

`listdir`(*path*)

`pretty_path`(*path*)

`read`(*path*)

`write`(*buf*, *path*)

class `papermill.iorw.`**`HttpHandler`**

Bases: `object`

classmethod **`listdir`**(*path*)

classmethod pretty_path(*path*)

classmethod read(*path*)

classmethod write(*buf*, *path*)

class papermill.iorw.**LocalHandler**

Bases: object

cwd(*new_path*)

Sets the cwd during reads and writes

listdir(*path*)

pretty_path(*path*)

read(*path*)

write(*buf*, *path*)

class papermill.iorw.**NoDatesSafeLoader**(*stream*)

Bases: yaml.loader.SafeLoader

yaml_implicit_resolvers = {'': [('tag:yaml.org,2002:null', re.compile('^(?: ~\n |nul

class papermill.iorw.**PapermillIO**

Bases: object

The holder which houses any io system registered with the system. This object is used in a singleton manner to save and load particular named Handler objects for reference externally.

get_handler(*path*)

listdir(*path*)

pretty_path(*path*)

read(*path, extensions=['.ipynb', '.json']*)

register(*scheme*, *handler*)

register_entry_points()

reset()

write(*buf, path, extensions=['.ipynb', '.json']*)

class papermill.iorw.**S3Handler**

Bases: object

classmethod listdir(*path*)

classmethod pretty_path(*path*)

classmethod read(*path*)

classmethod write(*buf*, *path*)

papermill.iorw.**fallback_gs_is_retriable**(*e*)

papermill.iorw.**get_pretty_path**(*path*)

papermill.iorw.**gs_is_retriable**(*e*)

papermill.iorw.**list_notebook_files**(*path*)

Returns a list of all the notebook files in a directory.

papermill.iorw.**load_notebook_node**(*notebook_path*)

Returns a notebook object with papermill metadata loaded from the specified path.

Parameters **notebook_path** (*str*) – Path to the notebook file.

Returns nbformat.NotebookNode

`papermill.iorw.`**`local_file_io_cwd`**(*path=None*)

`papermill.iorw.`**`read_yaml_file`**(*path*)
Reads a YAML file from the location specified at 'path'.

`papermill.iorw.`**`write_ipynb`**(*nb, path*)
Saves a notebook object to the specified path. :param nb_node: Notebook object to save. :type nb_node: nbformat.NotebookNode :param notebook_path: Path to save the notebook object to. :type notebook_path: str

3.1.5 Storage

Azure

These modules outline how to interact with Azure data stores, specifically Azure Blob Storage and Azure Data Lakes.

papermill.abs module

papermill.adl module

AWS

This module shows how to interact with AWS S3 data stores.

papermill.s3 module

3.1.6 Utilities

Utils

`papermill.utils.`**`any_tagged_cell`**(*nb, tag*)
Whether the notebook contains at least one cell tagged `tag`?

Parameters

- **nb** (*nbformat.NotebookNode*) – The notebook to introspect
- **tag** (*str*) – The tag to look for

Returns Whether the notebook contains a cell tagged `tag`?

Return type bool

`papermill.utils.`**`chdir`**(*path*)
Change working directory to *path* and restore old path on exit.

path can be *None* in which case this is a no-op.

`papermill.utils.`**`find_first_tagged_cell_index`**(*nb, tag*)
Find the first tagged cell `tag` in the notebook.

Parameters

- **nb** (*nbformat.NotebookNode*) – The notebook to introspect

- **tag** (*str*) – The tag to look for

Returns Whether the notebook contains a cell tagged `tag`?

Return type nbformat.NotebookNode

`papermill.utils.`**`merge_kwargs`**(*caller_args*, ***callee_args*)

Merge named argument.

Function takes a dictionary of caller arguments and callee arguments as keyword arguments Returns a dictionary with merged arguments. If same argument is in both caller and callee arguments the last one will be taken and warning will be raised.

Parameters

- **caller_args** (*dict*) – Caller arguments
- ****callee_args** – Keyword callee arguments

Returns **args** – Merged arguments

Return type dict

`papermill.utils.`**`remove_args`**(*args=None*, ***kwargs*)

Remove arguments from kwargs.

Parameters

- **args** (*list*) – Argument names to remove from kwargs
- ****kwargs** – Arbitrary keyword arguments

Returns **kwargs** – New dictionary of arguments

Return type dict

`papermill.utils.`**`retry`**(*num*)

Exceptions

exception `papermill.exceptions.`**`AwsError`**

Raised when an AWS Exception is encountered.

exception `papermill.exceptions.`**`FileExistsError`**

Raised when a File already exists on S3.

exception `papermill.exceptions.`**`PapermillException`**

Raised when an exception is encountered when operating on a notebook.

exception `papermill.exceptions.`**`PapermillExecutionError`**(*cell_index*, *exec_count*, *source*, *ename*, *evalue*, *traceback*)

Raised when an exception is encountered in a notebook.

exception `papermill.exceptions.`**`PapermillMissingParameterException`**

Raised when a parameter without a value is required to operate on a notebook.

exception `papermill.exceptions.`**`PapermillOptionalDependencyException`**

Raised when an exception is encountered when an optional plugin is missing.

exception `papermill.exceptions.`**`PapermillParameterOverwriteWarning`**

Callee overwrites caller argument to pass down the stream.

exception `papermill.exceptions.`**`PapermillRateLimitException`**

Raised when an io request has been rate limited

exception papermill.exceptions.**PapermillWarning**
Base warning for papermill.

papermill.exceptions.**missing_dependency_generator**(*package*, *dep*)

papermill.exceptions.**missing_environment_variable_generator**(*package*, *env_key*)

Log

Sets up a logger

3.2 papermill.tests package

3.2.1 Submodules

3.2.2 papermill.tests.test_abs module

3.2.3 papermill.tests.test_adl module

3.2.4 papermill.tests.test_autosave module

class papermill.tests.test_autosave.**TestMidCellAutosave**(*methodName='runTest'*)
Bases: unittest.case.TestCase

setUp()
Hook method for setting up the test fixture before exercising it.

test_autosave_disable()

test_autosave_not_too_fast()

test_end2end_autosave_slow_notebook()

3.2.5 papermill.tests.test_cli module

3.2.6 papermill.tests.test_clientwrap module

class papermill.tests.test_clientwrap.**TestPapermillClientWrapper**(*methodName='runTest'*)
Bases: unittest.case.TestCase

setUp()
Hook method for setting up the test fixture before exercising it.

test_logging_data_msg()

test_logging_stderr_msg()

test_logging_stdout_msg()

3.2.7 papermill.tests.test_conf module

3.2.8 papermill.tests.test_engines module

papermill.tests.test_engines.**AnyMock**(*cls*)
    Mocks a matcher for any instance of class cls. e.g. my_mock.called_once_with(Any(int), "bar")

class papermill.tests.test_engines.**TestEngineBase**(*methodName='runTest'*)
    Bases: `unittest.case.TestCase`

setUp()
    Hook method for setting up the test fixture before exercising it.

test_cell_callback_execute()

test_no_cell_callback_execute()

test_wrap_and_execute_notebook()
    Mocks each wrapped call and proves the correct inputs get applied to the correct underlying calls for execute_notebook.

class papermill.tests.test_engines.**TestEngineRegistration**(*methodName='runTest'*)
    Bases: `unittest.case.TestCase`

setUp()
    Hook method for setting up the test fixture before exercising it.

test_getting()

test_registering_entry_points()

test_registration()

class papermill.tests.test_engines.**TestNBClientEngine**(*methodName='runTest'*)
    Bases: `unittest.case.TestCase`

setUp()
    Hook method for setting up the test fixture before exercising it.

test_nb_convert_engine()

test_nb_convert_engine_execute()

test_nb_convert_log_outputs()

test_nb_convert_no_log_outputs()

class papermill.tests.test_engines.**TestNotebookExecutionManager**(*methodName='runTest'*)
    Bases: `unittest.case.TestCase`

setUp()
    Hook method for setting up the test fixture before exercising it.

test_basic_pbar()

test_cell_complete_after_cell_exception()

test_cell_complete_after_cell_start()

test_cell_complete_new_nb()

test_cell_complete_without_cell_start()

test_cell_exception()

test_cell_exception_new_nb()

test_cell_start()

test_cell_start_new_nb()

test_nb_isolation()
    Tests that the engine notebook is isolated from source notebook

test_no_pbar()

test_notebook_complete()

test_notebook_complete_cell_status_completed()

test_notebook_complete_cell_status_with_failed()

test_notebook_complete_new_nb()

test_notebook_start()

test_notebook_start_markdown_code()

test_notebook_start_new_nb()

test_save()

test_save_new_nb()

test_save_no_output()

test_set_timer()

3.2.9 papermill.tests.test_exceptions module

3.2.10 papermill.tests.test_execute module

class papermill.tests.test_execute.**TestBrokenNotebook1**(*methodName='runTest'*)
    Bases: unittest.case.TestCase

setUp()
    Hook method for setting up the test fixture before exercising it.

tearDown()
    Hook method for deconstructing the test fixture after testing it.

test()

class papermill.tests.test_execute.**TestBrokenNotebook2**(*methodName='runTest'*)
    Bases: unittest.case.TestCase

setUp()
    Hook method for setting up the test fixture before exercising it.

tearDown()
    Hook method for deconstructing the test fixture after testing it.

test()

class papermill.tests.test_execute.**TestCWD**(*methodName='runTest'*)
    Bases: unittest.case.TestCase

setUp()
    Hook method for setting up the test fixture before exercising it.

tearDown()
    Hook method for deconstructing the test fixture after testing it.

test_execution_respects_cwd_assignment()

test_local_save_ignores_cwd_assignment()

test_pathlib_paths()

class `papermill.tests.test_execute.`**TestNotebookHelpers**(*methodName='runTest'*)
Bases: `unittest.case.TestCase`

setUp()
Hook method for setting up the test fixture before exercising it.

tearDown()
Hook method for deconstructing the test fixture after testing it.

test_backslash_params()

test_backslash_quote_params()

test_cell_insertion()

test_default_start_timeout(*preproc_mock*)

test_double_backslash_quote_params()

test_no_tags()

test_prepare_only()

test_quoted_params()

test_start_timeout(*preproc_mock*)

class `papermill.tests.test_execute.`**TestNotebookValidation**(*methodName='runTest'*)
Bases: `unittest.case.TestCase`

setUp()
Hook method for setting up the test fixture before exercising it.

tearDown()
Hook method for deconstructing the test fixture after testing it.

test_from_version_4_4_upgrades()

class `papermill.tests.test_execute.`**TestReportMode**(*methodName='runTest'*)
Bases: `unittest.case.TestCase`

setUp()
Hook method for setting up the test fixture before exercising it.

tearDown()
Hook method for deconstructing the test fixture after testing it.

test_report_mode()

class `papermill.tests.test_execute.`**TestSysExit**(*methodName='runTest'*)
Bases: `unittest.case.TestCase`

setUp()
Hook method for setting up the test fixture before exercising it.

tearDown()
Hook method for deconstructing the test fixture after testing it.

test_sys_exit()

test_sys_exit0()

test_sys_exit1()

test_system_exit()

papermill.tests.test_execute.**execute_notebook**(*input_path*, *output_path*, *parameters=None*, *engine_name=None*, *request_save_on_cell_execute=True*, *prepare_only=False*, ***, *kernel_name='python3'*, *progress_bar=True*, *log_output=False*, *stdout_file=None*, *stderr_file=None*, *start_timeout=60*, *report_mode=False*, *cwd=None*, ***engine_kwargs*)

Executes a single notebook locally.

Parameters

- **input_path** (*str or Path*) – Path to input notebook
- **output_path** (*str or Path*) – Path to save executed notebook
- **parameters** (*dict, optional*) – Arbitrary keyword arguments to pass to the notebook parameters
- **engine_name** (*str, optional*) – Name of execution engine to use
- **request_save_on_cell_execute** (*bool, optional*) – Request save notebook after each cell execution
- **autosave_cell_every** (*int, optional*) – How often in seconds to save in the middle of long cell executions
- **prepare_only** (*bool, optional*) – Flag to determine if execution should occur or not
- **kernel_name** (*str, optional*) – Name of kernel to execute the notebook against
- **progress_bar** (*bool, optional*) – Flag for whether or not to show the progress bar.
- **log_output** (*bool, optional*) – Flag for whether or not to write notebook output to the configured logger
- **start_timeout** (*int, optional*) – Duration in seconds to wait for kernel start-up
- **report_mode** (*bool, optional*) – Flag for whether or not to hide input.
- **cwd** (*str or Path, optional*) – Working directory to use when executing the notebook
- ****kwargs** – Arbitrary keyword arguments to pass to the notebook engine

Returns **nb** – Executed notebook object

Return type NotebookNode

3.2.11 papermill.tests.test_gcs module

class papermill.tests.test_gcs.**GCSTest**(*methodName='runTest'*)

Bases: unittest.case.TestCase

Tests for *GCS*.

setUp()

Hook method for setting up the test fixture before exercising it.

test_fallback_gcs_invalid_code(*mock_gcs_filesystem*, *mock_gcs_retriable*)

test_gcs_fallback_retry_unknown_failure_code(*mock_gcs_filesystem*, *mock_gcs_retriable*)

test_gcs_handle_exception(*mock_gcs_filesystem*)

test_gcs_invalid_code(*mock_gcs_filesystem*, *mock_gcs_retriable*)

test_gcs_listdir(*mock_gcs_filesystem*)

test_gcs_read(*mock_gcs_filesystem*)

test_gcs_retry(*mock_gcs_filesystem*)

test_gcs_retry_older_exception(*mock_gcs_filesystem*)

test_gcs_unretryable(*mock_gcs_filesystem*)

test_gcs_write(*mock_gcs_filesystem*)

class papermill.tests.test_gcs.**MockGCSFile**(*exception=None*, *max_raises=1*)
Bases: `object`

read()

write(*buf*)

papermill.tests.test_gcs.**mock_gcs_fs_wrapper**(*exception=None*, *max_raises=1*)

3.2.12 papermill.tests.test_hdfs module

class papermill.tests.test_hdfs.**HDFSTest**(*methodName='runTest'*)
Bases: `unittest.case.TestCase`

setUp()
Hook method for setting up the test fixture before exercising it.

test_hdfs_listdir(*mock_hdfs_filesystem*)

test_hdfs_read(*mock_hdfs_filesystem*)

test_hdfs_write(*mock_hdfs_filesystem*)

class papermill.tests.test_hdfs.**MockHadoopFile**
Bases: `object`

read()

write(*new_content*)

class papermill.tests.test_hdfs.**MockHadoopFileSystem**(**args*, ***kw*)
Bases: `unittest.mock.MagicMock`

ls(*path*)

open(*path*, **args*)

3.2.13 papermill.tests.test_iorw module

3.2.14 papermill.tests.test_parameterize module

class papermill.tests.test_parameterize.**TestBuiltinParameters**(*methodName='runTest'*)
Bases: `unittest.case.TestCase`

test_add_builtin_parameters_adds_dict_of_builtins()

test_add_builtin_parameters_allows_to_override_builtin()

test_add_builtin_parameters_keeps_provided_parameters()

test_builtin_parameters_include_current_datetime_local()

test_builtin_parameters_include_current_datetime_utc()

test_builtin_parameters_include_run_uuid()

class papermill.tests.test_parameterize.**TestNotebookParametrizing**(*methodName='runTest'*)
Bases: unittest.case.TestCase

count_nb_injected_parameter_cells(*nb*)

test_custom_comment()

test_injected_parameters_tag()

test_no_parameter_tag()

test_no_tag_copying()

test_repeated_run_injected_parameters_tag()

test_repeated_run_no_parameters_tag()

class papermill.tests.test_parameterize.**TestPathParameterizing**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_parameterized_path_with_none_parameters()

test_parameterized_path_with_undefined_parameter()

test_path_with_boolean_parameter()

test_path_with_dict_parameter()

test_path_with_float_format_string()

test_path_with_list_parameter()

test_path_with_multiple_parameter()

test_path_with_none_parameter()

test_path_with_numeric_format_string()

test_path_with_numeric_parameter()

test_path_with_single_parameter()

test_plain_text_path_with_empty_parameters_object()

test_plain_text_path_with_none_parameters()

test_plain_text_path_with_unused_parameters()

3.2.15 papermill.tests.test_s3 module

3.2.16 papermill.tests.test_translators module

3.2.17 papermill.tests.test_utils module

3.2.18 Module contents

papermill.tests.**get_notebook_dir**(**args*)

papermill.tests.**get_notebook_path**(**args*)

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index