Contents

Books with Jupyter

DOI 10.5281/zenodo.2561065

Jupyter Book is an open source project for building beautiful, publication-quality books and documents from computational material.

Here are some of the features of Jupyter Book:

✓ **Write publication-quality content in Markdown**
You can write in either Jupyter Markdown, or an extended flavor of Markdown with publishing features. This includes support for rich syntax such as citations and cross-references, math and equations, and figures.

✓ **Write content in Jupyter Notebook**
This allows you to include your code and outputs in your book. You can also write notebooks entirely in Markdown that get executed when you build your book.

✓ **Execute and cache your book's content**
For `.ipynb` and Markdown notebooks, execute code and insert the latest outputs into your book. In addition, cache and re-use outputs to be used later.

✓ **Insert notebook outputs into your content**
Generate outputs as you build your documentation, and insert them in-line with your content across pages.

✓ **Add interactivity to your book**
You can toggle cell visibility, include interactive outputs from Jupyter, and connect with online services like Binder.

✓ **Generate a variety of outputs**
This includes single- and multi-page websites, as well as PDF outputs.

✓ **Build books with a simple command-line interface**
You can quickly generate your books with one command, like so: `jupyter-book build mybook/`

This website is built with Jupyter Book! You can browse its contents to the left to see what is possible.

Learn more and get involved

Jupyter Book is an open community that welcomes your feedback, input, and contributions!

💡 **Open an issue**
to provide feedback and new ideas, and to help others.

📚 **See the Jupyter Book Gallery**
to be inspired by Jupyter Books that others have created.

👍 **Vote for new features**
by adding a +1 to issues you'd like to see completed.

🙌 **Contribute to Jupyter Book**
by following our contributing guidelines and finding an issue to work on. See the feature voting leaderboard for inspiration.

Install Jupyter Book

You can install Jupyter Book via `pip`:

```
pip install -U jupyter-book
```

This will install everything you need to build a Jupyter Book locally.

Get started

To get started with Jupyter Book, you can either

- check out the getting started guide,
- browse the contents of the navigation menu of this book (to the left, if you're on a laptop), or
- review the example project shown immediately below (if you like learning from examples).

⚠ **Warning**

Jupyter Book `0.8` is a total re-write from previous versions, and some things have changed. See the legacy upgrade guide for how to upgrade, and legacy.jupyterbook.org for legacy documentation.

In addition, note that Jupyter Book is pre-1.0. Its API may change!

To install `jupyter-book` from pip, run the following command:

```
pip install -U jupyter-book
```

Create books from a template to get started right away, or see the getting started guide for more information.

A small example project

Here's a short example of a web-based book created by Jupyter Book.

Some of the features on display include

- Jupyter Notebook-style inputs and outputs
- citations
- numbered equations
- numbered figures with captions and cross-referencing

The source files can be found on GitHub in the docs directory. These files are written in MyST Markdown, an extension of the Jupyter Notebook Markdown, that allows for additional scientific markup. They could alternatively have been written directly as Jupyter notebooks.

Build the demo book

You can build this book locally on the command line via the following steps:

1. Ensure you have a recent version of Anaconda Python installed.
2. Clone the repository containing the demo book source files

   ```
   git clone https://github.com/executablebooks/quantecon-mini-example
   cd quantecon-mini-example
   ```

3. Install the Python libraries needed to run the code in this particular example from the `environment.yml` file. This includes the latest version of Jupyter Book:

   ```
   conda env create -f environment.yml
   conda activate qe-mini-example
   ```

4. Run Jupyter Book over the source files

   ```
   jupyter-book build ./mini_book
   ```

5. View the result through a browser — try (with, say, firefox)

   ```
   firefox mini_book/_build/html/index.html
   ```

   (or simply double-click on the `html` file)

Now you might like to try editing the files in `mini_book/docs` and then rebuilding.

If you'd like to install Jupyter Book with `pip`, you can do so with:

```
pip install -U jupyter-book
```

See the getting started page for more information.

Further reading

See the full QuantEcon example for a longer Jupyter Book use case, drawn from the same source material.

For more information on how to use Jupyter Book, see Overview.

Under the hood - the components of Jupyter Book

Jupyter Book is a wrapper around a collection of tools in the Python ecosystem that make it easier to publish computational documents. Here are a few key pieces:

- It uses the MyST Markdown language in Markdown and notebook documents. This allows users to write rich, publication-quality markup in their documents.
- It uses the MyST-NB package to parse and read-in notebooks so they are built into your book.
- It uses the Sphinx documentation engine to build outputs from your book's content.
- It uses a slightly modified version of the PyData Sphinx theme for beautiful HTML output.
- It uses a collection of Sphinx plugins and tools to add new functionality.

For more information about the project behind many of these tools, see The Executable Book Project documentation.

Contribute to Jupyter Book

Jupyter Book is an open project and we welcome your feedback and contributions! To contribute to Jupyter Book, see Contribute to Jupyter Book.

Acknowledgements

Jupyter Book is supported by an open community of contributors, many of whom come from the Jupyter community. Jupyter Book and many of the tools it uses are stewarded by the Executable Book Project, which is supported in part by the Alfred P. Sloan foundation.

Create your first book

In this tutorial, we'll cover the basics of the Jupyter Book ecosystem, and step you through creating, building, and publishing your first book.

What you should already know

In order to complete this tutorial, you should be relatively familiar with using the command line, as well as using a text editor.

A note for Windows users

Jupyter Book is now also tested against Windows OS 😀

However, there is a known incompatibility for notebook execution when using Python 3.8.

See Working on Windows

See the sections of this tutorial to the left to get started!

Overview

This is a short overview of the major components and steps in building a Jupyter Book. See the other pages in this guide for more in-depth information.

The Jupyter Book command-line interface

Jupyter Book uses a command-line interface to perform a variety of actions. For example, building and cleaning books. You can run the following command to see what options are at your control:

```
jupyter-book --help
```

You can also use the short-hand `jb` for `jupyter-book`. E.g.,: `jb create mybookname`. We'll use `jupyter-book` fo the rest of this guide.

```
Usage: jupyter-book [OPTIONS] COMMAND [ARGS]...

  Build and manage books with Jupyter.

Options:
  --version   Show the version and exit.
  -h, --help  Show this message and exit.

Commands:
  build   Convert your book's or page's content to HTML or a PDF.
  clean   Empty the _build directory except jupyter_cache.
  config  Inspect your _config.yml file.
  create  Create a Jupyter Book template that you can customize.
  myst    Manipulate MyST markdown files.
  toc     Generate a _toc.yml file for your content folder.
```

For more complete information about the CLI, see The command-line interface.

The book building process

Building a Jupyter Book broadly consists of these steps:

1. **Create your book's content**. You structure your book with a collection of folders, files, and configuration. See Anatomy of a Jupyter Book.
2. **Build your book**. Using Jupyter Book's command-line interface you can convert your pages into either an HTML or a PDF book. See Build your book.
3. **Publish your book online**. Once your book is built, you can share it with others. Most common is to build HTML, and host it as a public website. See Publish your book online.

Anatomy of a Jupyter Book

There are three things that you need in order to build a Jupyter Book:

- A configuration file (`_config.yml`)
- A table of contents file (`_toc.yml`)
- Your book's content

For example, consider the following folder structure, which makes up a simple Jupyter Book.

```
mybookname/
├── _config.yml
├── _toc.yml
├── landing-page.md
└── page1.ipynb
```

We'll cover each briefly below, and you can find more information about them elsewhere in this documentation.

Book configuration (`_config.yml`)

All of the configuration for your book is in a YAML file called `_config.yml`.

You can define metadata for your book (such as its title), add a book logo, turn on different "interactive" buttons (such as a Binder button for pages built from a Jupyter Notebook), and more.

Here's an example of a simple `_config.yml` file:

For more information about your book' configuration file, see Configure book settings.

```
# in _config.yml
title: "My book title"
logo: images/logo.png
execute:
  execute_notebooks: "off"
```

- `title:` defines a title for the book. It will show up in the left sidebar.
- `logo:` defines a path to an image file for your book's logo (it will also show up in the sidebar).
- `execute:` contains a collection of configuration options to control execution and cacheing.
  - `execute_notebooks: "off"` tells Jupyter Book **not to execute** any computational content that it finds when building the book. By default, Jupyter Book executes and caches all book content.

More about `_config.yml`

There is much more that you can do with the `_config.yml` file. For example, you can Add source repository buttons or add Interactive data visualizations. For a complete list of fields for `_config.yml`, see Configure book settings.

Table of Contents (`_toc.yml`)

Jupyter Book uses your Table of Contents to define the structure of your book. For example, your chapters, sub-chapters, etc.

This is a YAML file with a collection of pages, each one linking to a file in your book. Here's an example of the two content files shown above.

```
# In _toc.yml
- file: landing-page
- file: page1
```

If you would like to quickly **generate a basic Table of Contents** YAML file, ru the following command:

```
jupyter-book toc mybookname/
```

And it will generate a TOC for you. Not that there must be at least one conten file in each folder in order for any sub-folders to be parsed.

Each item in the `_toc.yml` file points to a single file. The links should be **relative to your book's folder and with no extension.** Think of the top-most level of your TOC file as **book chapters** (excluding the landing page). The title of each chapter will be inferred from the title in your files.

The first file specifies the **landing page** of your book (in this case, it is a **markdown file**). The landing page is the highest page in your book's content hierarchy. The second file specifies a **content page** of your book (in this case, it is a **Jupyter Notebook**).

For more information about how sectic structure maps onto book structure, see How headers and sections map onto to book structure.

More about `_toc.yml`

You can specify more complex book configurations with your `_toc.yml` file. For example, you can specify **parts**, **sections**, and control **custom titles**. For more information about your book's table of contents file, see Structure your book with the Table of Contents.

Book content

A collection of text files make up your book's content. These can be one of several types of files, such as markdown (`.md`), Jupyter Notebooks (`.ipynb`) or reStructuredText (`.rst`) files (see Types of content source files for a full list).

In the above example, there were two files listed: a **markdown** file and a **Jupyter Notebook**. We'll cover each in the next section.

Create your book's source files

Now that we understand our book's structure, let's create a sample book to learn from.

Quickly generate a sample book

Jupyter Book comes bundled with a lightweight sample book to help you understand a book's structure. Create a sample book by running the following command:

```
jupyter-book create mynewbook/
```

This will generate a mini Jupyter Book that you can both build and explore locally. It will have a few decisions made for you, and you can explore the configuration of the book in `_config.yml` and its structure in `_toc.yml`. Use this book as inspiration, or as a starting point to work from.

Investigate your book's content files

First, note that there are at least two different kinds of content files: **markdown** files (ending in `.md`) and **Jupyter Notebooks** (ending in `.ipynb`).

We'll discuss each below.

Markdown files (`.md`)

Markdown is an example of a markup language - a way to structure text with extra characters and syntax that give it extra meaning (e.g., using `**bold**` to denote **bold**). It is very popular and used across many different technology platforms.

Markdown files come in slight variations, often called *flavors of markdown*. There are two flavors of markdown that Jupyter Book supports:

- CommonMark markdown - a markdown standard that is very common.
- MyST Markdown - an extension of CommonMark with extra functionality for enriched documents.

Let's take a look at one of the markdown files in the template book, `intro.md`:

```
# Welcome to your Jupyter Book

This is a small sample book to give you a feel for how book content is
structured.

:::{note}
Here is a note!
:::

And here is a code block:

```
e = mc^2
```

Check out the content pages bundled with this sample book to see more.
```

Above you see several different kinds of structure:

- `#` symbols denote **section headers** in CommonMark markdown. They define the section headers on this page, for example.
- `:::{note}` is a **directive** in MyST Markdown. It is rendered like this:

  > **Note**
  >
  > I'm a note!

- ```` ``` ```` denotes a **code block** in CommonMark markdown. It is rendered like this:

  ```
  e=mc^2
  ```

All content files must have a page title (specified as the first header). All subsequent headers must increase linearly (so no jumps from H1 to H3). See Rules for all content types for more rules that all content must adhere to.

For more information about MyST markdown and all the things you can do with it, see MyST Markdown overview.

Jupyter Notebooks (`.ipynb`)

The other type of content we'll note is a **Jupyter Notebook**, ending in `.ipynb`. Jupyter Notebooks have a combination of computational content and narrative conent. Each notebook is associated with a **kernel** (aka, Python, R, Julia, etc) that defines the language used to execute the notebook's computational content.

By default, when Jupyter Book builds your book, **notebooks will be executed and their outputs cached**. On subsequent builds, notebook pages will be re-executed only if their code has changed.

Any outputs generated by the notebook will be inserted into your built book (though they may not be in your input notebook). This way you do not need to store the notebook's outputs with your repository. See Execute and cache your pages for more information.

There are many other interesting things that you can do with notebook content as a part of your book. We recommend checking out Formatting code outputs as well as Interactive data visualizations to get started with Jupyter notebooks.

✨**Notebooks with text files**✨

You can also store Jupyter Notebooks as markdown files or other text files. See Notebooks written entirely in Markdown and Custom notebook formats and Jupytext.

Inspect your configuration file

In addition to these content files, your book also has a **configuration file** (`_config.yml`). This controls the behavior of your book in many ways.

A few decisions have been made for you. Let's take a look at a few examples:

This snippet configures some metadata about your book, and determins the logo displayed in the top left:

```yaml
# Book settings
# Learn more at https://jupyterbook.org/customize/config.html

title: My sample book
author: The Jupyter Book Community
logo: logo.png
```

This configuration activates **citations** for your book (see Get started with references for getting started with citations and references).

```yaml
# Add a bibtex file so that we can create citations
bibtex_bibfiles:
  - references.bib
```

This configuration tells Jupyter Book to execute all of your Jupyter Notebook files **every time** the book is re-built, instead of cache-ing them.

```yaml
# Force re-execution of notebooks on each build.
# See https://jupyterbook.org/content/execute.html
execute:
  execute_notebooks: force
```

Check out the other content in your configuration file, and reference it against the pages in this documentation to see what it does.

Your book's table of contents

Finally, your book also has a **Table of Contents** that tells Jupyter Book where each of your content source files should go.

```yaml
- file: intro
- file: markdown
- file: notebooks
```

This is a simple table of contents, consisting of three pages. The first item of your ToC is always your book's **landing page** - the first page people will see when they look at your book.

Subsequent pages are defined in sequential order. You can do a lot more than just define a single list of pages - see Structure your book with the Table of Contents for more information about structuring your book.

Create your own content file

Now that you've seen a few sample content files, try creating your own!

In the folder with all of your sample book contents, create a new file called `mymarkdownfile.md`. Put the following content in it:

```md
# Here's my sample title

This is some sample text.

(section-label)=
## Here's my first section

Here is a [reference to the intro](intro.md). Here is a reference to [](section-label).
```

We've added two new pieces of markdown syntax, both of them are related to **cross-references**.

- `(section-label)=` is a label that's attached to a section header. It refers to whatever header follows, and allows you to refer to this label later on in your text.
- `[link text](link-target)` syntax is how you specify a link in markdown. Here we've linked to another page, as well as to the label we created above.

When you build your book, you'll see how these links resolve in the output.

Next step: build your book

Now that you've got a Jupyter Book folder structure, you can create the HTML (or PDF) for each of your book's pages. That's covered in the next section.

Build your book

Once you've added content and configured your book, it's time to build outputs for your book. We'll use the `jupyter-book build` command line tool for this.

Currently, there are two kinds of supported outputs: an HTML website for your book, and a PDF that contains all of the pages of your book that is built from the book HTML. In this tutorial, we'll focus on building HTML outputs.

Prerequisites

In order to build the HTML for each page, you should have followed the steps in Overview and Create your book's source files. You should have a collection of notebook/Markdown files in your `mybookname/` folder, a `_toc.yml` file that defines the structure of your book, and any configuration you'd like in the `_config.yml` file.

Build your book's HTML

Now that your book's content is in your book folder and you've defined your book's structure in `_toc.yml`, you can build the HTML for your book.

Do so by running the following command:

```
jupyter-book build mybookname/
```

This will generate a fully-functioning HTML site using a **static site generator**. The site will be placed in the `_build/html` folder. You can then open the pages in the site by navigating to that folder and opening the `html` files with your web browser.

> **Note**
>
> You can also use the short-hand `jb` for `jupyter-book`. E.g.,: `jb build mybookname/`.

Source vs build files

At this point, you have created a combination of Jupyter notebooks, markdown files, and configuration files, including `_toc.yml` and `_config.yml`. These files are your **source** files. The **source** files comprise all of the content and configuration that Jupyter Book needs to build your book.

In addition, you have created a collection of *outputs* in the `_build` folder. The `_build` folder contains all of your static website **build** files. The **build** files contain all of the output from Jupyter Book's `build` command. These files are only used to view your content in a browser or to share with others.

The best practice for publishing your book is to use separate branches for your **source** and your **build** files. For example, you may tell git to ignore your `_build` folder on your `main` branch, and push the outputs in your `_build` folder to a branch called `gh-pages`. We'll cover some of this later on.

A note on page cacheing

By default, Jupyter Book will only build the HTML for pages that have been updated since the last time you built the book.

If you'd like to force Jupyter Book to re-build a particular page, you can either edit the corresponding file in your book's folder, or delete that page's HTML in the `_build/html` folder.

You can also signal a full re-build using the `--all` option:

```
jupyter-book build --all mybookname/
```

Preview your built HTML

To preview your book, you can open the generated HTML files in your browser. Either double-click the html file in your local folder, or enter the absolute path to the file in your browser navigation bar adding `file://` at the beginning (e.g. `file://Users/my_path_to_book/_build/index.html`).

Take a look at the web page that was generated from the markdown page that you created. Note how the links you inserted were automatically **resolved** to point to the right place. This is how you can keep consistent pointers from one section of your book to another.

Next step: publish your book

Now that you've created the HTML for your book, it's time to publish it online. That's covered in the next section.

Publish your book online

Once you've built the HTML for your book, you can host it online. The best way to do this is with a service that hosts **static websites** (because that's what you have just created with Jupyter Book). In this tutorial, we'll cover how to publish your book online with GitHub Pages, a popular and free online hosting platform.

Create an online repository for your book

In order to connect your hosted book with your book's source content, you should put your book's source content in a public repository. This section describes one approach to create your own GitHub repository and add your book's content to it.

1. First, log in to GitHub, then go to the "create a new repository" page: https://github.com/new
2. Next, give your online repository a name and a description. Make your repository public and do not initialize it with a README file, then click "Create repository".
3. Now, clone the (currently empty) online repository to a location on your local computer. You can do this via the command line with:

   ```
   git clone https://github.com/<my-org>/<my-repository-name>
   ```

4. Copy all of your book files and folders into this newly cloned repository. For example, if you created your book locally with `jupyter-book create mylocalbook` and your new repository is called `myonlinebook`, you could do this via the command line with:

   ```
   cp -r mylocalbook/* myonlinebook/
   ```

5. Now you need to sync your local and remote (i.e., online) repositories. You can do this with the following commands:

   ```
   cd myonlinebook
   git add ./*
   git commit -m "adding my first book!"
   git push
   ```

Publish your book online with GitHub Pages

We have just pushed the *source files* for our book into our GitHub repository. This makes it publicly accessible for you or others to see.

Next, we'll publish the *build artifact* of our book online, so that it is rendered as a website.

The easiest way to use GitHub Pages with your built HTML is to use the `ghp-import` package. `ghp-import` is a lightweight Python package that makes it easy to push HTML content to a GitHub repository.

`ghp-import` works by copying *all* of the contents of your built book (i.e., the `_build/html` folder) to a branch of your repository called `gh-pages`, and pushes it to GitHub. The `gh-pages` branch will be created and populated automatically for you by `ghp-import`. To use `ghp-import` to host your book online with GitHub Pages follow the steps below:

> **Note**
>
> Before performing the below steps, ensure that HTML has been built for each page of your book (see the previous section). There should be a collection of HTML files in your book's `_build/html` folder.

1. Install `ghp-import`

   ```
   pip install ghp-import
   ```

2. Update the settings for your GitHub pages site:
   a. Use the `gh-pages` branch to host your website.
   b. Choose root directory `/` if you're building the book in it's own repository. Choose `/docs` directory if you're building documentation with jupyter-book.
3. From the `main` branch of your book's root directory (which should contain the `_build/html` folder) call `ghp-import` and point it to your HTML files, like so:

   ```
   ghp-import -n -p -f _build/html
   ```

> **Warning**
>
> Make sure that you included the `-n` - this tells GitHub *not* to build your book with Jekyll, which we don't want because our HTML is already built!

Typically after a few minutes your site should be viewable online at a url such as: `https://<user>.github.io/<myonlinebook>/`. If not, check your repository settings under **Options** -> **GitHub Pages** to ensure that the `gh-pages` branch is configured as the build source for GitHub Pages and/or to find the url address GitHub is building for you.

To update your online book, make changes to your book's content on the `main` branch of your repository, re-build your book with `jupyter-book build mybookname/` and then use `ghp-import -n -p -f mylocalbook/_build/html` as before to push the newly built HTML to the `gh-pages` branch.

> **Warning**
>
> Note this warning from the `ghp-import` GitHub repository:
>
> *"...`ghp-import` will DESTROY your gh-pages branch... and assumes that the `gh-pages` branch is 100% derivative. You should never edit files in your `gh-pages` branch by hand if you're using this script..."*

Get started with references

References allow you to refer to other content in your book or to external content. They allow you to automatically generate links to that content, or to add extra information like *numbers* to the reference.

Citations and bibliographies allow you to cite scholarly work and provide bibliographies that allow readers to follow the references.

This tutorial covers the basics of setting up **references** as well as **citations and bibliographies** for your book.

See also

For more information about citation and reference syntax, see the `sphinxcontrib-bibtex` documentation. Note that this documentation is written with rST syntax in mind and you'll need to adapt the directive/role syntax for your Markdown content.

Prerequisites

This tutorial assumes that you've either created a demo Jupyter Book from Create your first book, or that you've got your own Jupyter Book to work from.

Basic structure of a reference

Cross-references in Jupyter Book usually involve two things:

1. Create a **label** for something. This is the thing you'll refer to later in your reference.
2. Create a reference with a **target**. This target is usually the label you created in `#1`.

Create a label

First, we'll create a label. Labels must come just before headers. You can then refer to them elsewhere in your text.

To start with, create a new markdown file in your book (or edit a pre-existing one). Add a markdown header with a label like so:

```
(my-label)=
## My header

Some text
```

This is how you specify a label called `my-label` that points to the header just below (`## My header`).

Refer to your label

Now that you've created a label, you can refer to it from elsewhere. Try adding the following markdown on the same page (or some other page).

```
Here's some text and [here's my label](my-label).
```

You can also reference labels in this way with the following syntax:

```
Some text and {ref}`my-label`
```

Now re-build your book's HTML:

```
jb build pathto/mybook
```

You should see that your reference has been replaced with a link to the right spot on the page.

Create a citation

Next, we'll add a *citation*.

Create a bibtex file

You'll need a bibtex file to store the information for your citations. In this case, we'll create an empty bibtex file and populate it with one reference.

```
touch references.bib
```

Next, configure your book to include this bibtex file, like so:

```
# In _config.yml
bibtex_bibfiles:
    - references.bib
```

This will activate the `sphinxcontrib.bibtex` extension

Add your references

Add some references to your BibTex file. Here's an example citation:

```
@article{perez2011python
,       title   = {Python: an ecosystem for scientific computing}
,       author  = {Perez, Fernando and Granger, Brian E and Hunter, John D}
,       journal = {Computing in Science \\& Engineering}
,       volume  = {13}
,       number  = {2}
,       pages   = {13--21}
,       year    = {2011}
,       publisher       = {AIP Publishing}
}
```

See also

See the BibTex documentation for information on the BibTex reference style.

Add a citation

In your content, add the following syntax to include a citation:

```
Here is my nifty citation {cite}`perez2011python`.
```

Re-build your book, and it should look like this:

Here is my nifty citation [Perez *et al.*, 2011].

Add multiple citations at once

Now try adding multiple citations at once by separating each one with a comma.

Add the following text to your page:

```
Here are multiple citations
{cite}`perez2011python,holdgraf_rapid_2016,RePEc:the:publsh:1367,caporaso2010qiime`!
```

when you build your book, it should look like this:

Here are multiple citations [Caporaso *et al.*, 2010, Holdgraf *et al.*, 2016, Perez *et al.*, 2011, Stachurski and Kamihigashi, 2014]!

Add a bibliography

Finally, we'll generate a bibliography for our citations. Links to this bibliography will automatically be created when you cite something.

We'll use the `{bibliography}` directive to add one to our book. Add the following to your page:

~~~
```{bibliography}
```
~~~

This will generate the bibliography of all citations in your book. See the bibliography below for an example.

See also

For more information about configuring and using citations and bibliographies, see References and cross-references.

Bibliography

An example bibliography, for reference:

Get help with Jupyter Book

What if I have an issue or question?

If you've got questions, concerns, or suggestions, please open an issue at at the Jupyter Book issues page

Structure your book with the Table of Contents

There are many ways in which you can control the table of contents for your book. Most of them involve adding syntax to your `_toc.yml` file.

This page covers a few common options.

Note

The `_toc.yml file for this site` has an entry for each of the features described below for reference.

General TOC structure

The table of contents is broadly organized like so:

- The first entry of your `_toc.yml` file is the *introduction* to your book. It is the landing page for the HTML of your book.
- Subsequent entries define either **parts** or **chapters** in your book. These make up the main structure of your book. See Defining chapters and parts in _toc.yml for more information.
- Each chapter can optionally have **sections** that are defined by separate files. These are nested underneath the top page of the chapter. See How headers and sections map onto to book structure for more information.
- Throughout the `_toc.yml` file, `- file:` entries point to text files that make up your book's content. Their paths are relative to the book's root.

Note

By default, the landing page of your book will appear in the navbar, but this can be disabled in your `_config.yml` file by setting the `home_page_in_navbar` option to `false` (under the html section).

Note

Currently, it is not possible to add nested sections to your landing page (see #844)

For reference, here is an example similar to this book's `_toc.yml` file:

```
- file: myintro
  numbered: true

- part: Get started
  chapters:
  - file: start/overview
  - file: start/build

- part: Reference and test pages
  chapters:
  - file: test_pages/test
    sections:
      - file: test_pages/layout_elements
      - file: test_pages/equations
```

The sections below cover this information in more depth.

Defining chapters and parts in `_toc.yml`

The top layer of entries in your table of contents allows you to define **chapters** and (optionally) **parts** of your book.

The first entry (`- file: myintro` above) defines the introductory page for your book. It is also where you can control some behavior for the entire book (in the example above, we set `numbered: true` to number *all* sections of the book).

Below the first entry you have two options for defining the structure of your book:

1. **A list of chapters.** You can specify each chapter with a `- file:` entry. Below is an example `_toc.yml` file with this structure:

   ```
   - file: myintro

   - file: firstchapter
   - file: secondchapter
   ```

2. **A list of parts with chapters.** If you'd like to separate chapters into groups, do so by using `- part:` entries in the top level of `_toc.yml`. Each part should have a `chapters:` section that contains a list of `- file:` entries, each one pointing to the file for a chapter. Below is an example `_toc.yml` file with this structure:

   ```
   - file: myintro

   - part: My first part
     chapters:
     - file: part1_firstchapter
     - file: part1_secondchapter
   - part: My second part
     chapters:
     - file: part2_firstchapter
   ```

   Note that **chapters do not continue between parts**. Think of each part as a self-contained collection of chapters (e.g., for the purposes of numbering).

⚠ **Don't mix these two structures!**

When designing the top-level sections of your `_toc.yml` file, you must pick *either* a list of chapters via `- file:` entries, or a list of parts via `- part:` entries with chapters inside of them. You cannot intermix them both.

Files

Files point to a single file of content in your book's folder. If these files are at the top level of your `_toc.yml` file, they will denote **chapters**. If they are nested within another file (via the `sections:` key) then they will denote **sections** within a chapter.

Here is an example file entry:

```
- file: path/to/myfile
```

Additionally, **files can have nested sections in other files**. These subsections allow you to define hierarchical structure in your book. For example, you may wish for the top-level file to serve as an "introduction" for a collection of files underneath, like so:

```
- file: my_intro
  sections:
    - file: my_first_page
    - file: my_second_page
      sections:
        - file: my_second_page_subsection
```

We recommend nesting your sections no more than 3 layers deep (as shown above).

Specifying alternate titles

If you'd like to specify an alternate title from the one defined within a file, you may do so with the `title:` key. For example:

```
- file: path/to/myfile
  title: My alternate page title
```

Note that this only applies to the sidebar in the table of contents, it does not change the actual chapter/section title.

Number your book's chapters and sections

You can automatically add numbers to each chapter of your book. To add numbers to **all chapters of your book**, add the `numbered: true` flag to your introduction page entry (the first entry in `_toc.yml`). For example:

```
- file: intro
  numbered: true

- file: chapter1
- file: chapter2
- file: chapter3
```

This will cause all chapters of the book to be numbered. They will follow a hierarchy according to the sub-sections structure defined in your `_toc.yml` file. You can also **limit the TOC numbering depth** by setting the `numbered` flag to an integer instead of `true`, e.g., `numbered: 3`.

If you'd like to number **subsets of chapters**, group them into parts and apply the `numbered: true` flag to the parts whose chapters you wish to be numbered. For example:

```
- file: home

# Chapters in this part will not be numbered
- part: Introduction
  chapters:
  - file: page2

# Chapters in this part will be numbered
- part: Part 1
  numbered: true
  chapters:
  - file: chapter1
  - file: chapter2
```

Numbering caveats and notes

Jupyter Book relies on Sphinx to apply section numbering, and this has a few quirks to it. Here are a few gotchas:

- **Numbering applies to *sections* of your page**. Note that when you add numbering to a section, it will add numbers to *each header in a file*. This means that if you have headers in a top-level section, then its headers will become numbered as sub-sections, and any other *files* underneath it will begin as third-level children. See How headers and sections map onto to book structure for more information.
- **Numbering resets across parts**. If you specify groups of sections via `- part:` entries, then numbering will restart between them. That means if you have two `- part:` entries with 2 pages each, you will have two sets of `1.` and `2.` sections, one for each part.

How headers and sections map onto to book structure

Jupyter Book uses the Sphinx documentation engine under the hood, which represents the structure of your book in a particular way. Different choices for the structure of `_toc.yml` and the header structures within your pages will result in different outcomes for your overall book structure. Here are some general tips and best-practices.

Note

This is particularly important when you number your book's sections or when you build a PDF of your book through Latex.

Chapters are at the top of your book hierarchy. The top level of your `_toc.yml` contains a list of chapters. The title of each file will be the chapter's title.

Headers map onto sections. Jupyter Book interprets your book as a collection of sections, and decides how those sections should be nested according to the hierarchy of `_toc.yml` and the hierarchy of headers in a page. Within a file, the first `##` header it discovers will define the top-most section in the file, and any subsequent `###` headers underneath will become sub-sections (until another `##` section is encountered). This behavior is a bit different if the page is *nested* under another (see below).

Nested files define sections *underneath* the last section of their parent. If you specify sections that are *nested* under a file (with the `sections:` key) then those sections will begin *underneath* the last headers of the parent page.

For example, if your `_toc.yml` file looks like this:

```
- file: myintro

- file: chapter1
  sections:
  - file: chapter1section
```

Then the sections of `chapter1section` will begin **under** the sections of `chapter1`. Any headers in `chapter1section` will be treated as a "next-header-deeper" section in `chapter1`.

In other words, if `chapter1` and `chapter1section` look like this:

`chapter1.md`

```
# Chapter 1 title

## Chapter 1 second header
```

`chapter1section.md`

```
# Chapter 1 section title

## Chapter 1 section second header
```

Then your book will treat them like so:

```
# Chapter 1 title

## Chapter 1 second header

### Chapter 1 section title

#### Chapter 1 section second header
```

However, if `chapter1.md` had an extra third-level header, like so:

`chapter1.md`

```
# Chapter 1 title

## Chapter 1 second header

### Chapter 1 third header
```

`chapter1section.md`

```
# Chapter 1 section title

## Chapter 1 section second header
```

Then your book would treat them like so:

```
# Chapter 1 title

## Chapter 1 second header

### Chapter 1 third header

#### Chapter 1 section title

##### Chapter 1 section second header
```

Keep this in mind when you design the structure of your files.

Tip

A good rule of thumb is to take one of these two approaches:

1. **don't put headers in your introduction pages**. This is true for both the book's introduction, as well as for any chapter introductions. Instead, leave the headers to pages that have more content in them, and use **bolded text** where you would otherwise use headers.
2. **Use a flat list of files instead of nested files**. This way the section hierarchy is defined only in a single file within each section. However, this means you will have longer files in general.

Exclude some pages from your book's build

By default, Jupyter Book will build all content files that are found in your book's folder, even if they are not specified in `_toc.yml` (and will raise a warning if it finds a file that isn't listed there).

If you'd like Jupyter Book to skip a file entirely, you can do so with the following configuration in `_config.yml`:

```
exclude_patterns: [pattern1/*, path/to/myfile.ipynb]
```

Any files that match the patterns described there will be excluded from the build. If you'd like to exclude files from being *executed* but still wish for them to be built by Jupyter Book, see Exclude files from execution.

Web-based navigation bar functionality

The following sections apply to controlling the left navigation bar in HTML books built with Jupyter Book.

Add external links

You can also add external links to websites that are outside of your book. To do so, use the following pattern:

```
- url: https://yoururl.com
```

The URL will be placed alongside the links to other pages above and below the entry.

Add a table of contents to a page's content

If you'd like to add a table of contents for the sub-sections of a page *within the page content* (in-line with the content on the page), you may do so by using the `{tableofcontents}` directive. You can use it like so:

~~~
```{tableofcontents}
```
~~~

See the source of the content types page for an example.

Automatically generate your `_toc.yml` file

You can use `jupyter-book` to *generate* a table of contents file from your book using the filenames of your book's content. To do so, run the following command

```
jupyter-book toc mybookpath/
```

Jupyter Book will search `mybookpath/` for any content files and create a `_toc.yml` file out of them. There are a few considerations to keep in mind:

- Each sub-folder must have at least one content file inside it
- The ordering of files in `_toc.yml` will depend on the alphanumeric order of the filenames (e.g., `folder_01` comes before `folder_02`, and `apage` comes before `b_page`)
- If there is a file called `index.md` in any folder, it will be listed first.

You may also **generate navigation bar *titles* from each file of your book**. If you do so, note that if the file name begins with `<integer>_filename.md`, then the `<integer>` part will be removed before it is inserted into `_toc.yml`.

Create books from a template

Jupyter Book lets you quickly generate a book structure from templates. This section covers the process of creating a template book and building it as an alternative to manually creating the files in your book.

To see your options for creating books from templates, run the following command:

```
jupyter-book create --help
```

```
Usage: jupyter-book create [OPTIONS] PATH_BOOK

  Create a Jupyter Book template that you can customize.

Options:
  --cookiecutter  Use cookiecutter to interactively create a Jupyter Book
                  template.

  -h, --help      Show this message and exit.
```

Quickly generate a sample book

This option is best if you are starting from scratch, or would like to see one example of a simple Jupyter Book on your own filesystem.

If you'd just like to quickly create a sample book, you may do so by running the following command:

```
jupyter-book create mynewbook/
```

This will generate a mini Jupyter Book that you can both build and explore locally. It will have a few decisions made for you, and you can explore the configuration of the book in `_config.yml` and its structure in `_toc.yml`. Use this book as inspiration, or as a starting point to work from.

Generate a more complete book from interactive prompts

This option is best if you'd like to answer a few questions from the command line in order to create a template book that is more complex and customized for your use-case.

Jupyter Book also provides a Jupyter Book cookiecutter that can be used to interactively create a book directory structure.

The cookiecutter is suitable for users that want to create a ready-to-go repository to host their book that includes pre-populated metafiles such as `README`, `LICENSE`, `CONDUCT`, `CONTRIBUTING`, etc., as well as GitHub Actions workflow files to Automatically host your book with GitHub Actions.

`cookiecutter` is a Python tool for quickly generating folders from a templatized repository. Jupyter Book uses `cookiecutter` under the hood.

To try the cookiecutter template, run the following command:

```
jupyter-book create mynewbook/ --cookiecutter
```

For more help, see the Jupyter Book cookiecutter GitHub repository, or run:

Book content and syntax

In this section we'll cover the many ways that you can write content for your book. You can write book content in a variety of markup languages and file formats, and create special kinds of content for your book with MyST Markdown.

See the sections below to get started.

MyST Markdown overview

In addition to Jupyter Notebook Markdown, Jupyter Book also supports a special flavour of Markdown called **MyST (or Markedly Structured Text)**. It was designed to make it easier to create publishable computational documents written with Markdown notation. It is a superset of CommonMark Markdown and draws heavy inspiration from the fantastic RMarkdown language from RStudio.

Whether you write your book's content in Jupyter notebooks (`.ipynb`) or in regular Markdown files (`.md`), you'll write in the same flavour of **MyST Markdown**. Jupyter Book will know how to parse both of them.

For those who are familiar with Sphinx, MyST Markdown is basically CommonMark + Markdown extensions + Sphinx roles and directives

This page contains a few pieces of information about MyST Markdown and how it relates to Jupyter Book. You can find much more information about this flavour of Markdown at the Myst Parser documentation.

Want to use RMarkdown directly?

See Custom notebook formats and Jupytext

Directives and roles

Roles and directives are two of the most powerful tools in Jupyter Book. They are kind of like *functions*, but written in a markup language. They both serve a similar purpose, but **roles are written in one line** whereas **directives span many lines**. They both accept different kinds of inputs, and what they do with those inputs depends on the specific role or directive being used.

Directives

Directives customize the look, feel, and behaviour of your book. They are kind of like *functions*, and come in a variety of names with different behaviour. This section covers how to structure and use them.

At its simplest, you can use directives in your book like so:

~~~
```{mydirectivename}
My directive content
```
~~~

This will only work if a directive with name `mydirectivename` already exists (which it doesn't). There are many pre-defined directives associated with Jupyter Book. For example, to insert a note box into your content, you can use the following directive:

~~~
```{note}
Here is a note
```
~~~

This results in:

Note

Here is a note

being inserted in your built book.

For more information on using directives, see the MyST documentation.

More arguments and metadata in directives

Many directives allow you to control their behaviour with extra pieces of information. In addition to the directive name and the directive content, directives allow two other configuration points:

directive arguments - a list of words that come just after the `{directivename}`.

Here's an example usage of directive arguments:

~~~
```{directivename} arg1 arg2
My directive content.
```
~~~

directive keywords - a collection of flags or key/value pairs that come just underneath `{directivename}`.

There are two ways to write directive keywords, either as `:key: val` pairs, or as `key: val` pairs enclosed by `---` lines. They both work the same way:

Here's an example of directive keywords using the `:key: val` syntax:

~~~
```{directivename}
:key1: metadata1
:key2: metadata2
My directive content.
```
~~~

and here's an example of directive keywords using the enclosing `---` syntax:

~~~
```{directivename}
---
metadata1: metadata2
metadata3: metadata4
---
My directive content.
```
~~~

Tip

Remember, specifying directive keywords with `:key:` or `---` will make no difference. We recommend using `---` if you have many keywords you wish to specify, or if some values will span multiple lines. Use the `:key: val` syntax as a shorthand for just one or two keywords.

For examples of how this is used, see the sections below.

Roles

Roles are very similar to directives, but they are less complex and written entirely in one line. You can use a role in your book with this syntax:

```
Some content {rolename}`and here is my role's content!`
```

Again, roles will only work if `rolename` is a valid role name. For example, the `doc` role can be used to refer to another page in your book. You can refer directly to another page by its relative path. For example, the syntax ``{doc}`../intro` `` will result in: Books with Jupyter.

Warning

It is currently a requirement for roles to be on the **same line** in your source file. It will not be parsed correctly if it spans more than one line. Progress towards supporting roles that span multiple lines can be tracked by this issue

For more information on using roles, see the MyST documentation.

What roles and directives are available?

There is currently no single list of roles / directives to use as a reference, but this section tries to give as much as information as possible. For those who are familiar with the Sphinx ecosystem, **you may use any directive / role that is available in Sphinx**. This is because Jupyter Book uses Sphinx to build your book, and MyST Markdown supports all syntax that Sphinx supports (think of it as a Markdown version of reStructuredText).

Caution

If you search the internet (and the links below) for information about roles and directives, the documentation will generally be written with reStructuredText in mind. MyST Markdown is different from reStructuredText, but all of the functionality should be the same. See the MyST Sphinx parser documentation for more information about the differences between MyST and rST.

For a list of directives that are available to you, there are three places to check:

1. The Sphinx directives page has a list of directives that are available by default in Sphinx.
2. The reStructuredText directives page has a list of directives in the Python "docutils" module.
3. This documentation has several additional directives that are specific to Jupyter Book.

What if it exists in rST but not MyST?

In some unusual cases, MyST may be incompatible with a certain role or directive. In this case, you can use the special `eval-rst` directive, to directly parse reStructuredText:

~~~
```{eval-rst}
.. note::

   A note written in reStructuredText.
```
~~~

which produces

Note

A note written in reStructuredText.

See also

The MyST-Parser documentation on how directives parse content, and its use for including rST files into a Markdown file, and using `sphinx.ext.autodoc` in Markdown files.

Nesting content blocks in Markdown

If you'd like to nest content blocks inside one another in Markdown (for example, to put a `{note}` inside of a `{margin}`), you may do so by adding extra backticks (`` ` ``) to the outer-most block. This works for literal code blocks as well.

For example, the following syntax:

~~~
````
```
```
````
~~~

yields

~~~
```
```
~~~

Thus, if you'd like to nest directives inside one another, you can take the same approach. For example, the following syntax:

~~~
````{margin}
```{note}
Here's my note!
```
````
~~~

produces:

Note

Here's my note!

Other MyST Markdown syntax

In addition to roles and directives, there are numerous other kinds of syntax that MyST Markdown supports. MyST supports all syntax of CommonMark Markdown (the kind of Markdown that Jupyter notebooks use), as well as an extended syntax that is used for scientific publishing.

The MyST-Parser is the tool that Jupyter Book uses to allow you to write your book content in MyST. It is also a good source of information about the MyST syntax. Here are some links you can use as a reference:

- CommonMark block syntax
- Extended MyST block syntax in MyST
- CommonMark in-line syntax
- Extended in-line syntax in MyST

See also

For information about enabling extended MyST syntax, see MyST syntax extensions. In addition, see other examples of this extended syntax (and how to enable each) throughout this documentation.

What can I create with MyST Markdown?

See Special content blocks for an introduction to what you can do with MyST Markdown in Jupyter Book. In addition, the other pages in this site cover many more use-cases for how to use directives with MyST.

Tools for writing MyST Markdown

There is some support for MyST Markdown in tools across the community. Here we include a few prominent ones.

Jupyter interfaces

While MyST Markdown does not (yet) render in traditional Jupyter interfaces, most of its syntax should "gracefully degrade", meaning that you can still work with MyST in Jupyter, and then build your book with Jupyter Book.

Jupytext and text sync

For working with Jupyter notebook and Markdown files, we recommend jupytext, an open source tool for two-way conversion between `.ipynb` and text files. Jupytext supports the MyST Markdown format.

Note

For full compatibility with `myst-parser`, it is necessary to use `jupytext>=1.6.0`.

See also Convert a Jupytext file into a MyST notebook.

VS Code

If editing the Markdown files using VS Code, the VS Code MyST Markdown extension provides syntax highlighting and other features.

Special content blocks

A common use of directives and roles is to designate "special blocks" of your content. This allows you to include more complex information such as warnings and notes, citations, and figures. This section covers a few common ones.

MyST syntax extensions

MyST Markdown has a base syntax that it supports, and additional syntax can be enabled to add extra functionality. By default, Jupyter Book enables a few extra syntax pieces for MyST in order to more closely resemble the Markdown experience in Jupyter Notebooks and interfaces. These extensions are:

`dollarmath`
: To support `$$` and `$` syntax for math blocks. See Math and equations.

`linkify`
: To auto-detect HTML-like links in your markdown and convert them to hyperlinks.

`substitution`
: To allow you to define markdown "variables" and substitute text in using them. See Substitutions and variables in markdown.

`colon_fence`
: To allow you to use `:::` fences for admonitions, in order to make them easier to render in interfaces that do not support MyST. See Markdown-friendly directives with :::.

To enable your own syntax extensions, use the following configuration pattern:

```
parse:
  myst_enable_extensions:
    - extension-1
    - extension-2
```

Note that this will **override** the default Jupyter Book extension list. You should include all of the extensions that you want to be enabled.

See also

For a list of syntax extensions in MyST, see the MyST documentation.

Notes, warnings, and other admonitions

Let's say you wish to highlight a particular block of text that exists slightly apart from the narrative of your page. You can use the `{note}` directive for this.

For example, the following text:

~~~
```{note}
Here is a note!
```
~~~

Results in the following output:

Note

Here is a note!

A note on nesting

You can nest admonitions (and other content blocks) inside one another. Fo example:

Note

Here's a note block inside a margin block

See Nesting content blocks in Markdown for instructions to do this.

There are a number of similarly-styled blocks of text. For example, here is a `{warning}` block:

Warning

Here's a warning! It was created with:

~~~
```{warning}
```
~~~

For a complete list of options, see the `sphinx-book-theme` documentation.

Blocks of text with custom titles

You can also choose the title of your message box by using the `{admonition}` directive. For example, the following text:

~~~
```{admonition} Here's your admonition
Here's the admonition content
```
~~~

Results in the following output:

Here's your admonition

Here's the admonition content

If you'd like to **style these blocks**, then use the `:class:` option. For example:

This admonition was styled...

Using the following pattern:

~~~
```{admonition} My title
:class: tip
My content
```
~~~

Markdown-friendly directives with `:::`

The admonition syntax above utilises the general directives syntax. However, if you're using an interface that does not support MyST Markdown, it will render as a raw literal block. Many directives contain markdown inside, and if you'd like this markdown to render "normally", you may also use `:::` fences rather than ```` ``` ```` fences to define the directive. As a result, the contents of the directive will be rendered as markdown.

For example:

```
:::{note}
This text is **standard** _Markdown_
:::
```

Note

This text is **standard** *Markdown*

Similar to normal directives, these admonitions can also be nested:

```
::::{important}
:::{note}
This text is **standard** _Markdown_
:::
::::
```

Important

Note

This text is **standard** *Markdown*

Note

You can use this syntax for any kind of directive, though it is generally recommended to use only with directives that contain pure markdown in their content.

Insert code cell outputs into admonitions

If you'd like to insert the outputs of running code *inside* admonition blocks, we recommend using `glue` functionality. For example, we'll insert one of the outputs that was glued into the book from the code outputs page.

The code below:

~~~
```{note}
Here's my figure:
{glue:figure}`sorted_means_fig`
```
~~~

generates:

Note

Here's my figure:

See Insert code outputs into page content for more information on how to use `glue` to insert your outputs directly into your content.

Tip

To hide code input and output that generated the variable you are inserting, use the `remove_cell` tag. See Hide or remove content for more information and other tag options.

HTML admonitions

A drawback of admonition syntax is that it will not render in interfaces that do not support this syntax (e.g., GitHub). If you'd like to use admonitions that are defined *purely with HTML*, MyST can parse them via the `html_admonitions` extension. To use it, first enable it with the following configuration:

```
parse:
  myst_enable_extensions:
    # don't forget to list any other extensions you want enabled,
    # including those that are enabled by default!
    - html_admonition
```

Then, you may define admonitions in your book like so:

Markdown Input Rendered Output

```
<div class="admonition note" name="html-admonition" style="background: lightgreen; padding: 10px">
<p class="title">This is the **title**</p>
This is the *content*
</div>
```

See HTML Admonitions for more information about HTML admonitions.

Panels

Panels provide an easy way for you to organize chunks of content into flexible containers on your page. They are useful for creating card-like layouts, flexible columns, and grids. Panels are based off of Bootstrap CSS, and utilize Bootstrap's classes to control the look and feel of panels.

Here is an example that creates two panels:

~~~
````{panels}
Panel header 1
^^^
Panel body 1
+++
Panel footer 1
---

Panel header 2
^^^
Panel body 2
+++
Panel footer 2
~~~
```

- `---` separates each panel
- `^^^` defines the panel header
- `+++` defines the panel footer

Note

Panel headers and footers are optional. If you don't include `^^^` or `+++` in your panel, they will not show up.

You can embed all kinds of content inside of panels. For example, the following panels:

Content of the left panel.

example-badge

Clickable right panel

were created with:

```
````{panels}
Content of the left panel.

{badge}`example-badge,badge-primary`

---

```{link-button} content/panels
:text: Clickable right panel
:type: ref
:classes: stretched-link
```

````
```

**See also**

See the Sphinx Panels card layout documentation for more information.

## Controlling the look and feel of panels

You can control the look and feel of panels by passing attaching bootstrap classes to panel headers/body/footers. You do this by passing configuration options to your `{panels}` directive.

For example:

**See also**

See the Panels card styling documentation for more information.

For example, you can control how many columns are in your panels by using Bootstrap column classes. These panels:

Header A

Body A

Header B

Body B

Header C

Body C
```

Were created by this code:

~~~
````{panels}
:column: col-4
:card: border-2
Header A
^^^
Body A
---
Header B
^^^
Body B
---
Header C
^^^
Body C
~~~
```

## Dropdowns

Dropdowns allows you to hide content behind a title and a button. There are two kinds of dropdowns in Jupyter Book:

### The `{dropdown}` directive

Use the `{dropdown}` directive to create a clickable dropdown with a title.

For example:

source

```
```{dropdown} Here's my dropdown
And here's my dropdown content
```
```

result

Here's my dropdown

### Dropdown admonitions

You can also hide the body of your admonition blocks so that users must click a button to reveal their content. This is helpful if you'd like to include some text that isn't immediately visible to the user.

To turn an admonition into a dropdown, add the `dropdown` class to them. For example:

source

```
```{note}
:class: dropdown
The note body will be hidden!
```
```

result

**Note**

You can use this in conjunction with `{admonition}` directives to include your own titles and stylings. For example:

source

```
:::{admonition} Click here!
:class: tip, dropdown
```

result

**Click here!**
```

```
Here's what's inside!
:::
```

**Important**

Admonition dropdowns require JavaScript to be enabled on the browser which they are viewed. By contrast, the dropdown directive below works purely *via* HTML+CSS.

## Definition lists

Definition lists are enabled by defining the following setting in your `_config.yml`:

```yaml
parse:
  myst_enable_extensions:
    # don't forget to list any other extensions you want enabled,
    # including those that are enabled by default!
    - deflist
```

Definition lists utilise the markdown-it-py deflist plugin, which itself is based on the Pandoc definition list specification.

Here's an example:

source

```
Term 1
: Definition

Term 2
: Definition
```

result

**Term 1**
  Definition
**Term 2**
  Definition

From the Pandoc documentation:

> Each term must fit on one line, which may optionally be followed by a blank line, and must be followed by one or more definitions. A definition begins with a colon or tilde, which may be indented one or two spaces.
>
> A term may have multiple definitions, and each definition may consist of one or more block elements (paragraphs, code blocks, lists, etc.)

Here is a more complex example, demonstrating some of these features:

**Term *with Markdown***
  Definition with reference

  A second paragraph
  A second definition
**Term 2**
  Definition 2a
  Definition 2b
**Term 3**

```
A code block
```

> A quote

A final definition, that can even include images:

This was created with the following Markdown:

```
Term *with Markdown*
: Definition [with reference](ontent/definition-lists)

  A second paragraph

Term 2
  ~ Definition 2a
  ~ Definition 2b

Term 3
:     A code block

: > A quote

: A final definition, that can even include images:

  <img src="../images/fun-fish.png" alt="fishy" width="200px">
```

## Quotations and epigraphs

Quotations and epigraphs provide ways to highlight information given by others.

### Quotations

**Regular quotations** are controlled with standard Markdown syntax, i.e., by inserting a caret (>) symbol in front of one or more lines of text. For example:

source

```
> Here is a cool quotation.
>
> From me, Jo the Jovyan
```

result

> Here is a cool quotation.
>
> From me, Jo the Jovyan

### Epigraphs

**Epigraphs** draw more attention to a quote and highlight its author. You should keep these relatively short so that they don't take up too much vertical space. Here's how an epigraph looks:

source

~~~
```{epigraph}
Here is a cool quotation.

From me, Jo the Jovyan
```
~~~

result

Here is a cool quotation.

From me, Jo the Jovyan

You can provide an **attribution** to an epigraph by adding `--` to the final line, followed by the quote author. For example:

source

~~~
```{epigraph}
Here is a cool quotation.

-- Jo the Jovyan
```
~~~

result

> Here is a cool quotation.
>
> —Jo the Jovyan

### Glossaries

Glossaries allow you to define terms in a glossary so you can then link back to it throughout your content. You can create a glossary with the following syntax:

~~~
```{glossary}
Term one
  An indented explanation of term 1

A second term
  An indented explanation of term2
```
~~~

which creates:

**Term one**
  An indented explanation of term 1

**A second term**
  An indented explanation of term2

To reference terms in your glossary, use the `{term}` role. For example, ``{term}`Term one` `` becomes Term one and ``{term}`A second term` `` becomes A second term.

### Tabbed content

You can also use `sphinx-panels` to produce **tabbed content**. This allows you to display a variety of tabbed content blocks that users can click on.

For example, here's a group of tabs showing off code in a few different languages:

c++ python java julia fortran

```
int main(const int argc, const char **argv) {
  return 0;
}
```

You can use this functionality with the `{tabbed}` directive. You can provide a sequence of `{tabbed}` directives, and each one will be used to generate a new tab (unless the `:new-group:` option is added to a `{tabbed}` directive.)

For example, the following code:

~~~
```{tabbed} Tab 1 title
My first tab
```

```{tabbed} Tab 2 title
My second tab with `some code`!
```
~~~

produces

**Tab 1 title** Tab 2 title

My first tab

**Insert code outputs in your tabs** with the `glue` functionality.

For example, the following tabs use this functionality to glue images and tables generated somewhere else in these docs:

**A histogram** A table Code to generate this

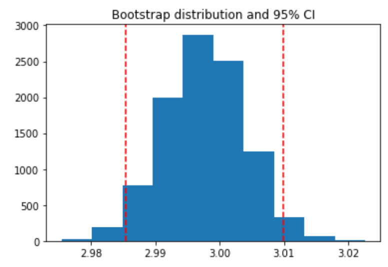


**Fig. 1** This is a **caption**, with an embedded `{glue:text}` element: **3.00**!

See the `sphinx-panels` tabbed documentation for more information on how to use this.

## Substitutions and variables in markdown

Substitutions allow you to define **variables** in the front-matter of your page, and then **insert** those variables into your content throughout.

To use a substitution, first add front-matter content to the top of a page like so:

~~~
---
substitutions:
  key1: "I'm a **substitution**"
  key2: |
    ```{note}
    {{ key1 }}
    ~~~
  fishy: |
    ```{image} img/fun-fish.png
    :alt: fishy
    :width: 200px
    ```
---
```

You can use these substitutions inline or as blocks, and you can even nest substitutions in other substitutions (but circular references are prohibited):

**Markdown Input** Rendered Output

```
Inline: {{ key1 }}

Block level:

{{ key2 }}
```

You can also insert substitutions inside of other markdown structures like tables:

**Markdown Input** Rendered Output
```

```
| col1     | col2      |
| -------- | --------- |
| {{key2}} | {{fishy}} |
```

**See also**

For more information about Substitutions, see Substitutions (with Jinja2).

### Define substitutions for your whole book

You can also define book-level substitution variables with the following configuration:

```yaml
parse:
  myst_substitutions:
    key: value
```

These substitutions will be available throughout your book. For example, the global substitution key `my-global-substitution` is defined in this book's `_config.yml` file, and it produces: My *global* value!.

### Formatting substitutions

MyST substitutions use Jinja templates in order to substite in key / values. This means that you can apply any standard Jinja formatting to your substitutions. For example, you can **replace text in your substitutions** like so:

Markdown Input    Rendered Output

```
The original key1: {{ key1 }}

{{ key1 | replace("a substitution", "the best substitution")}}
```

### Using substitutions in links

If you'd like to use substitutions to insert and modify **links** in your book, here are two options to explore:

1. **Define the entire markdown link as a variable**. For example:

   Markdown Input    Rendered Output

   ```yaml
   substitutions:
     repo_url: [my repo url](https://github.com/executablebooks/jupyter-book)
   ```

   ```
   Here's my link: {{ repo_url }}
   ```

2. Use Jinja features to insert the variable. Because substitutions use Jinja templates, you also have access to **Python formatting** operations in your substitution. For example:

   Markdown Input    Rendered Output

   ```yaml
   substitutions:
     repo_name: jupyter-book
   ```

   ```
   Here's my link: {{ '[my repo: `{repo}`]
   (https://github.com/executablebooks/{repo})'.format(repo=repo_name) }}
   ```

### Citations and cross-references

You can add citations and cross-references to your book. See Citations and bibliographies for more information on how to do this.

### Figures

You can thoroughly customise the look of figures in your book. See Images and figures for more information.

### Page layout and sidebar content

You can also use MyST to control various aspects of the page layout. For more information on this, see Control the page layout.

### Footnotes

You can include footnotes in your book using standard Markdown syntax. This will include a numbered reference to the footnote in-line, and append the footnote to a list of footnotes at the bottom of the page.

To create a footnote, first insert a reference in-line with this syntax: `[^mylabel]`. Then, define the text for that label like so:

```
[^mylabel]: My footnote text.
```

You can define `[^mylabel]` anywhere in the page, though its definition will always be placed at the bottom of your built page. For example, here's a footnote [1] and here's another one [2]. You can click either of them to see the footnotes at the bottom of this page.

### Custom `<div>` blocks

You can add custom `div` blocks along with whatever classes you'd like using the `{div}` directive. The `{div}` directive will wrap everything inside in a single `<div>` with the classes you provide. For example:

~~~
```{div} my-class
**Some content.**
```
~~~

Will result in the following HTML when your book is built:

```
<div class="my-class">
  <strong>Some content.</strong>
</div>
```

This can be useful if you'd like to style your book with custom CSS or JavaScript.

[1] Here's the text of my first note.

[2] And the text of my second note. Note that you can include Markdown footnote definitions.

## References and cross-references

Because `jupyter-book` is built on top of Sphinx, there are many ways to reference content within your book (or even across other books, or Sphinx websites).

Referencing is accomplished with **roles** or with **markdown link syntax**, depending on your use-case. There are a few ways to reference content from your book, depending on what kind of content you'd like to reference.

**See also**

If you're getting started, check out Get started with references for more information.

## Reference section labels

Labels are a way to add tags to parts of your content so that you can reference them later on. This is helpful if you want to quickly insert links to other parts of your book. Labels can be added before major elements of a page, such as titles or figures.

To add a label, use the following pattern **before** the element you wish to label:

```
(my-label)=
# The thing to label
```

For example, we've added the following label above the header for this section with:

```
(content:references:labels)=
## Cross-references and labels
```

You can insert cross-references to labels in your content with two kinds of syntax:

- `{ref}`label-text``
- `[](label-text)`

For example, the syntax `{ref}`content:references:labels`` or `[](content:references:labels)` results in a link to this section like so: Reference section labels.

## Reference figures

To reference a figure in your book, first add a figure and ensure that it has both a `name` as well as a caption associated with it:

source

~~~
```{figure} ../images/cool.jpg
:name: my-fig-ref

My figure title.
```
~~~

result

**Fig. 3** My figure title.

Then, reference the figure by its `:name:` value. For example:

| source | result |
|---|---|
| `Here is {ref}`my-fig-ref`` | Here is My figure title. |
| `Here is {ref}`My cool fig <my-fig-ref>`` | Here is My cool fig |
| `Here is [](my-fig-ref)` | Here is My figure title. |
| `Here is [My cool fig](my-fig-ref)` | Here is My cool fig |
| `Here is {numref}`my-fig-ref`` | Here is Fig. 3 |
| `Here is {numref}`Custom Figure %s text `` | Here is Custom Figure 3 text |

## Reference tables

To reference a table, first create a table and ensure that it has a `:name:` and a title:

| source | result |
|---|---|
| <pre>```{table} My table title<br>:name: my-table-ref<br><br>\| header 1 \| header 2 \|<br>\|---\|---\|<br>\| 3 \| 4 \|<br>```</pre> | **header 1 header 2**<br>3 4<br><br>Table 1 My table title |

Here are several ways to reference this content:

| source | result |
|---|---|
| `Here is {ref}`my-table-ref`` | Here is My table title |
| `Here is {ref}`My cool table <my-table-ref>`` | Here is My cool table |
| `Here is [](my-table-ref)` | Here is My table title |
| `Here is [My cool table](my-table-ref)` | Here is My cool table |
| `Here is {numref}`my-table-ref`` | Here is Table 1 |
| `Here is {numref}`Custom Table %s text `` | Here is Custom Table 1 text |

## Reference content files

To reference other files of book content, use the `{doc}` role, or link directly to another file with Markdown link syntax. For exmaple:

| source | result |
|---|---|
| `Here is {doc}`../file-types/myst-notebooks`` | Here is Notebooks written entirely in Markdown |
| `Here is {doc}`A different page <../file-types/myst-notebooks>`` | Here is A different page |
| `Here is [](../file-types/myst-notebooks.md)` | Here is Notebooks written entirely in Markdown |
| `Here is [A different page](../file-types/myst-notebooks.md)` | Here is A different page |

## Reference equations

To reference equations, first insert an equation with a label like so:

$$w_{t+1} = (1 + r_{t+1}) s(w_t) + y_{t+1} \tag{1}$$

To reference equations, use the `{eq}` role. It will automatically insert the number of the equation. Note that you cannot modify the text of equation links.

For example:

- `See Equation `{eq}`my-math-ref`` results in: See Equation (1)

- `See Equation [](my-math-ref)` results in: See Equation (1).

### Choose the reference text

If you'd like to choose the text of the rendered reference link, use the following pattern:

```
{someref}`your text here <reference-target>`
```

Above, `reference-target` is the reference to which you are referring, and `your text here` will be the displayed text on the page.

For example, see the following references:

| Raw text | Rendered text |
|---|---|
| `{ref}`Here's another references section <content:references:labels>`` | Here's another references section |
| `{doc}`Here's the code outputs section <code-outputs>`` | Here's the code outputs section |

### Number your references

You can add **numbered references** to either tables or figures. To add a numbered reference to a table or figure, use the `{numref}` role. If you wish to use custom text, add `%s` as a placeholder for the number.

See the examples in each section below for usage.

### Reference with markdown link syntax

If you wish to use Markdown style syntax, then MyST Markdown will try to find a reference from any of the above reference types (and more!).

This has an advantage, in that you can used nested markdown syntax in your text, for example:

| Raw text | Rendered text |
|---|---|
| `[A **bolded _reference_** to a page](./myst.md)` | A **bolded *reference*** to a page |
| `[A reference to a header](content:references:labels)` | A reference to a header |

Leaving the title empty will mean the reference uses the target as text, for example the syntax

```
[](./myst.md)
```

will link to a section and use its header text as the link text itself:

MyST Markdown overview

**Internal vs. External URLs**

You can control how MyST Markdown distinguishes between internal references and external URLs in your `_config.yml`. For example,

```
parse:
  myst_url_schemes: [mailto, http, https]
```

means that `[Jupyter Book](https://jupyterbook.org)` will be recognised as a URL, but `[Citations](content:citations)` will not:

- Jupyter Book
- Citations

### Check for missing references

You can check for missing references when building a Jupyter Book. To do so, use the following options:

```
jupyter-book build -W -n --keep-going docs/
```

This will check for missing references (`-n`) and turn them into errors (`-W`), but will still attempt to run the full build (`--keep-going`), so that you can see all errors in one run.

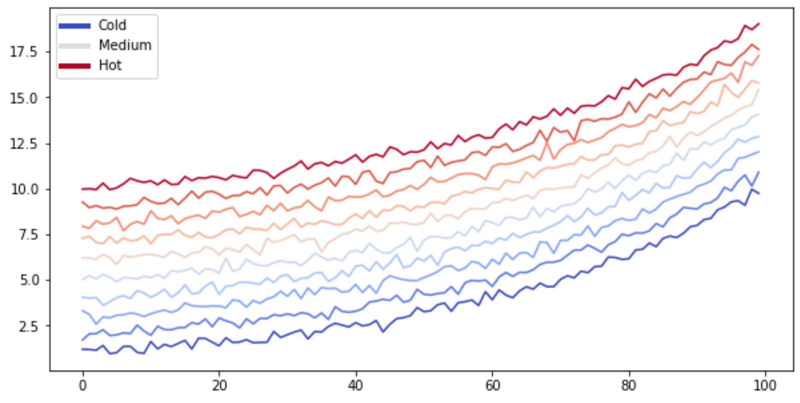
## Control the page layout

**Warning**

Many of the features on this page are experimental and may change at any time.

There are a few ways to control the layout of a page with Jupyter Book. Many of these ideas take inspiration from the Edward Tufte layout CSS guide.

Let's begin with a sample plot. You can click the toggle button to the right to see the code that generated it.



### Sidebar content

Adding sidebar elements allow you to provide contextual information that doesn't break up the flow of your main content. It is one of the main patterns recommended in the Tufte style guide.

There are two kinds of sidebars supported by Jupyter Book. We'll describe them below.

**Note**

Some sidebar content behaves differently depending on the screen size. If the screen is narrow enough, the sidebar content will exist in-line with your content. Make the screen wider and it'll pop out to the right.

### Sidebars within content

If you use a sidebar within your content, the sidebar will stay in-line with your page's content. However, it will be placed to the right, allowing your content to wrap around it. This prevents the sidebar from breaking up the flow of your content. This is particularly useful if you've got tall-and-long blocks of content or images that you would like to provide context to throughout your content.

To add a sidebar to your content, use the following syntax:

~~~
```{sidebar} My sidebar title
My sidebar content
```
~~~

**Here is some sidebar content**

It spans a bit of your main content, as well as the margin, as seen by the note block below:

**Note**

Here's a note block within the sidebar!

### Margin content

To add content to the margin with MyST Markdown, use this syntax:

~~~
```{margin} An optional title
My margin content
```
~~~

Controlling margin content with code cells uses a slightly different syntax, which we'll cover below.

**For example**

Here's some margin content! It was created by using the

~~~
```{margin}
```
~~~

directive in a Markdown cell. Jupyter Book automatically converts these cell into helpful margin content.

### Margins with code cells

You can make a code cell move to the right margin by adding `margin` to your cell's tags.

**Jupyter Notebook** MyST Text File

Here's what the cell metadata for a margin cell looks like:

```
{
    "tags": [
        "margin",
    ]
}
```

**See also**

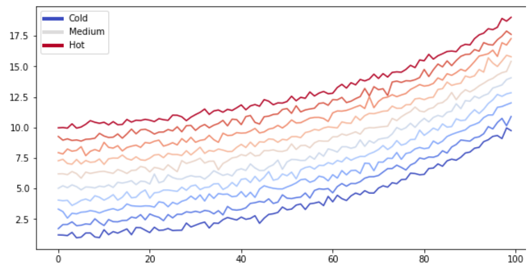
[Add metadata to notebooks]

For example, we'll re-display the figure above, and add a `margin` tag to the code cell.

This can be combined with other tags like `remove-input` to **only display the figure**.

The MyST cheat sheet provides a list of `code-cell` tags available

```
make_fig(figsize=(10, 5))
```



## Full-width content

Sometimes, you'd like to use **all** of the horizontal space available to you. This allows you to highlight particular ideas, visualizations, etc.

### Full-width code cells

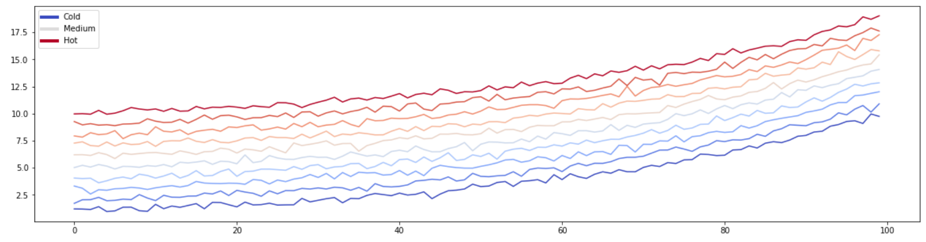
You can specify that a code cell's inputs and/or outputs should take up all of the horizontal space (including the margin to the right) using the following cell metadata tag:

```
{
    "tags": [
        "full-width",
    ]
}
```

**See also**

For tips on how to add cell metadata to your notebooks, see Add metadata to notebooks.

For example, let's take a look at the figure in the margin above in a cell with `full-width` set. We'll tell Matplotlib to make it a bit wider so we can take advantage of the extra space!



### Full-width markdown content

If you'd like to make your markdown content full-width, you cannot do so via cell tags. Instead, you have a few options:

1. **Use the `{div}` directive with a `full-width` class.**. Any content with a `full-width` class will take up the full width of the screen. For example, the following code:

   ~~~
   ````{div} full-width
   ```{note}
   Here's a note that will take the full width
   ~~~
   ~~~
   ```

   results in:

   **Note**

   Here's a note that will take the full width

   For more information on `<div>` blocks, see Custom <div> blocks.
2. **Add a `full-width` class to directives that support classes**. Many directives allow you to directly add a CSS class to them.
   For example, the `{note}` directive above allows for this:

   ~~~
   ```{note}
   :class: full-width
   Here's a note that will take the full width
   ```
   ````

   results in:

   Note

   Here's a note that will take the full width

   Check the documentation of the directive to see if it supports adding your own classes, or use the `{div}` directive as described above.
   ````

Mixing margins and full-width content

Be careful when mixing margins and full-width content. Sometimes these can conflict with one another in visual space. You should use them relatively sparingly in order for them to have their full effect of highlighting information.

Citations and bibliographies

You can add citations and bibliographies using references that are stored in a `bibtex` file that is in your book's folder. You can then add a citation in-line in your Markdown with the `{cite}` role, and include the bibliography from your bibtex file with the `{bibliography}` directive.

See also

This functionality uses the excellent sphinxcontrib-bibtex extension.

Basic citations

To get started with citations in Jupyter Book, check out Get started with references.

Change the in-line citation style

There are a few alternatives roles that you can use to change the in-line citation style. Below are a few examples:

- The citation `` {cite:p}`perez2011python` `` results in [Perez *et al.*, 2011]
- The citation `` {cite:t}`perez2011python` `` results in Perez *et al.* [2011]
- The citation `` {cite:ps}`perez2011python` `` results in [Perez, Granger, and Hunter, 2011]
- The citation `` {cite:ts}`perez2011python` `` results in Perez, Granger, and Hunter [2011]

See also

For a more complete list of in-line citation styles, check out the `sphinxcontrib-bibtex` docs.

Select your bibliography style

You can also optionally customize the style of your references. By default, references are displayed in the `alpha` style. Other currently supported styles include `plain`, `unsrt`, and `unsrtalpha`. These styles create the following bibliography formatting:

- `alpha`: Use alphanumeric reference labels, citations are sorted by author, year.
- `plain`: Use numeric reference labels, citations are sorted by author, year.
- `unsrt`: Use numeric reference labels, citations are sorted by order of appearance.
- `unsrtalpha`: Use alphanumeric reference labels, citations are sorted by order of appearance.

To set your reference style, use the style option:

~~~
```{bibliography}
:style: unsrt
```
~~~

Change the reference style

There are a few options for your in-line citation style. To select one, use the following configuration in `_config.yml`.

```
# In _config.yml
bibtex_reference_style: author_year  # or label, super, \supercite
```

See also

For a list of configuration options and more detail about this, see the `sphinxcontrib-bibtex` docs

Local bibliographies

You may wish to include a bibliography listing at the end of each document rather than having a single bibliography contained in a separate document. Having multiple bibliography directives, however, can cause `sphinx` to issue `duplicate citation warnings`.

A common fix is to add a filter to the bibliography directives:

~~~
```{bibliography}
:filter: docname in docnames
```
~~~

See `sphinxcontrib-bibtex` documentation on local bibliographies.

Example Bibliography

An example bibliography, for reference:

[CKS+10]
J Gregory Caporaso, Justin Kuczynski, Jesse Stombaugh, Kyle Bittinger, Frederic D Bushman, Elizabeth K Costello, Noah Fierer, Antonio Gonzalez Pena, Julia K Goodrich, Jeffrey I Gordon, and others. Qiime allows analysis of high-throughput community sequencing data. *Nature methods*, 7(5):335–336, 2010.

[HdHP+16]
Christopher Ramsay Holdgraf, Wendy de Heer, Brian N. Pasley, Jochem W. Rieger, Nathan Crone, Jack J. Lin, Robert T. Knight, and Frédéric E. Theunissen. Rapid tuning shifts in human auditory cortex enhance speech intelligibility. *Nature Communications*, 7(May):13654, 2016. URL: http://www.nature.com/doifinder/10.1038/ncomms13654, doi:10.1038/ncomms13654.

[PGH11](1,2,3,4)
Fernando Perez, Brian E Granger, and John D Hunter. Python: an ecosystem for scientific computing. *Computing in Science ||& Engineering*, 13(2):13–21, 2011.

[SK14]
John Stachurski and Takashi Kamihigashi. Stochastic stability in monotone economies. *Theoretical Economics*, 2014.

Math and equations

Jupyter Book uses MathJax for typesetting math in your HTML book build. This allows you to have LaTeX-style mathematics in your online content. This page shows you a few ways to control this.

See also

For more information about equation numbering, see the MathJax equation numbering documentation.

Tip

By default MathJax version 2 is currently used. If you are using a lot of math, you may want to try using version 3, which claims to improve load speeds by 60 - 80%:

```
sphinx:
  config:
    mathjax_path: https://cdn.jsdelivr.net/npm/mathjax@3/es5/tex-mml-chtml.js
```

See the Sphinx documentation for details.

In-line math

To insert in-line math use the `$` symbol within a Markdown cell. For example, the text `$this_{is}^{inline}$` will produce: $this_{is}^{inline}$.

Math blocks

You can also include math blocks for separate equations. This allows you to focus attention on more complex or longer equations, as well as link to them in your pages. To use a block equation, wrap the equation in either `$$` or `\begin` statements.

For example,

```
$$
  \int_0^\infty \frac{x^3}{e^x-1}\,dx = \frac{\pi^4}{15}
$$
```

results in:

$$\int_0^\infty \frac{x^3}{e^x-1}\,dx = \frac{\pi^4}{15}$$

Latex-style math

You can enable parsing LaTeX-style math blocks with the `amsmath` MyST extension. Enable it by adding the following to `_config.yml`

```
parse:
  myst_enable_extensions:
    # don't forget to list any other extensions you want enabled,
    # including those that are enabled by default!
    - amsmath
```

Once enabled, you can define math blocks like so:

```
\begin{gather*}
a_1=b_1+c_1\\
a_2=b_2+c_2-d_2+e_2
\end{gather*}

\begin{align}
a_{11}& =b_{11}&
  a_{12}& =b_{12}\\
a_{21}& =b_{21}&
  a_{22}& =b_{22}+c_{22}
\end{align}
```

which results in:

$$\begin{gather*}
a_1=b_1+c_1\\
a_2=b_2+c_2-d_2+e_2
\end{gather*}$$

$$\begin{align}
a_{11}& =b_{11}&
  a_{12}& =b_{12}\\
a_{21}& =b_{21}&
  a_{22}& =b_{22}+c_{22}
\end{align} \tag{2}$$

See also

The MyST guides to dollar math syntax, LaTeX math syntax, and how MyST-Parser works with MathJax.

For advanced use, also see how to define MathJax TeX Macros.

Numbering equations

If you'd like to number equations so that you can refer to them later, use the **math directive**. It looks like this:

~~~
```{math}
:label: my_label
my_math
```
~~~

For example, the following code:

~~~
```{math}
:label: my_label
w_{t+1} = (1 + r_{t+1}) s(w_t) + y_{t+1}
```
~~~

will generate

$$w_{t+1} = (1 + r_{t+1}) s(w_t) + y_{t+1} \tag{3}$$

Alternatively you can use the dollar math syntax with a prefixed label:

```
$$
  w_{t+1} = (1 + r_{t+1}) s(w_t) + y_{t+1}
$$ (my_other_label)
```

which generates

$$w_{t+1} = (1 + r_{t+1}) s(w_t) + y_{t+1} \tag{4}$$

Note

Labels cannot start with an integer, or they won't be able to be referenced and will throw a warning message if referenced. For example, `:label: 1` and `:label: 1eq` cannot be referenced.

Linking to equations

If you have created an equation with a label, you can link to it from within your text (and across pages!).

You can refer to the equation using the label that you've provided by using the `{eq}` role. For example:

```
- A link to an equation directive: {eq}`my_label`
- A link to a dollar math block: {eq}`my_other_label`
```

results in

- A link to an equation directive: (3)
- A link to a dollar math block: (4)

Note

`\labels` inside LaTeX environment are not currently identified, and so cannot be referenced. We hope to implement this in a future update (see executablebooks/MyST-Parser#202)!

Images and figures

Images

MyST Markdown provides a few different syntaxes for including images in your documents, as explained below.

The first is the standard Markdown syntax, by which

```
![fishy](../images/fun-fish.png)
```

results in

This will correctly copy the image to the build folder and will render it in all output formats (HTML, TeX, etc). However, it is limited in the configuration that can be applied. For example, the image width cannot be set with this syntax.

As discussed in this section, MyST allows for directives such as `image` and `figure` to be used (see the Sphinx documentation for available options).

As an example,

~~~
```{image} ../images/fun-fish.png
:alt: fishy
:class: bg-primary mb-1
:width: 200px
:align: center
```
~~~

will include the following *customized* figure:

These directives allow you to control aspects of the image with directive arguments.

Raw HTML images

The image syntax described above gives you more customizability, but note that this syntax will not show the image in common Markdown viewers (for example when the files are viewed on GitHub).

A workaround is to use HTML directly, and MyST can parse HTML images directly via the `html_image` extension.

> **Warning**
>
> Using raw HTML is usually a bad choice (see this explanation), so be careful before doing so!

To parse raw HTML image syntax, enable the `html_image` extension to your list of extensions in `_config.yml`:

```
parse:
  myst_enable_extensions:
    # don't forget to list any other extensions you want enabled,
    # including those that are enabled by default!
    - html_image
```

HTML images will be parsed like any other image. For example:

```
<img src="../images/fun-fish.png" alt="fishy" class="bg-primary" width="200px">
```

will correctly render

This will also be output in PDF LaTeX builds!

Allowed attributes are equivalent to the `image` directive: `src`, `alt`, `class`, `width` and `height`. Any other attributes will be ignored.

Supported image formats

Standard rasterized image formats, such as `.png` and `jpg`, are supported for both HTML and LaTeX/PDF output formats. By contrast, vector formats such as `.svg`, `.pdf` and `.eps` are normally builder specific. See the `supported_image_types` specification for each Sphinx builder here.

To support multiple builders, Jupyter Book allows you to use a `*` asterisk as the extension. For example, with the HTML

```
<img src="../images/fun-fish.*" alt="fishy" class="bg-primary mb-1" width="200px">
```

all images matching the provided pattern will then be searched for and each builder chooses the best image out of the available candidates.

The code above produced the following image:

You can use a tool such as imagemagick, to convert your images to multiple formats prior to building your book.

Alternatively, you may wish to check out these Sphinx extensions:

- sphinx.ext.imgconverter
- sphinxcontrib-svg2pdfconverter

Figures

MyST Markdown also lets you include **figures** in your page. Figures are like images, except that they are easier to reference elsewhere in your book, and they include things like captions. To include a figure, use this syntax:

~~~
```{figure} ../images/C-3PO_droid.png
---
height: 150px
name: directive-fig
---
Here is my figure caption!
```
~~~

which will produce the following:

Fig. 4 Here is my figure caption!

Note

You can also include figures that were generated by your code in notebooks. To do so, see Insert code outputs into page content.

Markdown figures

Markdown figures combine colon style admonitions and HTML image parsing, to produce a "Markdown friendly" syntax for figures, with equivalent behaviour to the `figure` directive above.

Note

Using this feature requires that HTML image parsing is enabled.

The figure block must contain **only** two components; an image, in either Markdown or HTML syntax, and a single paragraph for the caption. See below for an example.

As with admonitions, the figure can have additional classes set. The "title" of the admonition is used as the label that can be targeted by your cross-references.

For example, the code

```
:::{figure-md} markdown-fig
<img src="../images/fun-fish.png" alt="fishy" class="bg-primary mb-1" width="200px">

This is a caption in **Markdown**!
:::
```

generates this figure:

Fig. 5 This is a caption in **Markdown**!

As we see here, we can reference the figure:

Go to the fish!

We just have to use the title of the admonition as target:

```
[Go to the fish!](markdown-fig)
```

Referencing figures

You can then refer to your figures by using the `{ref}` role or Markdown style references like:

```
- {ref}`directive-fig`
- [](markdown-fig)
```

which will replace the reference with the figure caption like so:

- Here is my figure caption!
- This is a caption in Markdown!

Numbered references

Another convenient way to create cross-references is with the `{numref}` role, which referes to the labelled objects by the numbers they automatically get. For example, `{numref}`directive-fig`` will produce a reference like: Fig. 4.

If an explicit text is provided, this caption will serve as the title of the reference. For example,

```
- {ref}`Fly to the droid <directive-fig>`
- [Swim to the fish](markdown-fig)
```

produces the following cross-references:

- Fly to the droid
- Swim to the fish

With `numref` you can also access the figure number and caption individually: the sequences "%s" and "{number}" will be replaced with the figure number, while "{name}" will be replaced with the figure caption.

For example, `{numref}`Figure {number}: {name} <directive-fig>`` will produce Figure 4: Here is my figure caption!.

Margin captions and figures

You can include a figure caption on the margin using `:figclass: margin-caption`, as seen in Fig. 6:

Fig. 6 Here is my figure caption!

Another option is to include figures on the margin by using `:figclass: margin` as seen in Fig. 7:

Fig. 7 Here is my figure caption!

Figure scaling and aligning

Figures can also be aligned by using the option `:align: right` or `:align: left`. By default, figures are aligned to the center (see Fig. 4).

To align a figure on the left, you'd write

~~~
```{figure} ../images/cool.jpg
---
scale: 50%
align: left
---
Here is my figure caption!
```
~~~

to get

Fig. 8 Here is my figure caption!

Similarly, if you write

~~~
```{figure} ../images/cool.jpg
---
scale: 50%
align: right
---
Here is my figure caption!
```
~~~

your figure becomes right-aligned:

Fig. 9 Here is my figure caption!

Figure parameters

The following options are supported:

`scale` : ***integer percentage***
Uniformly scale the figure. The default is “100” which indicates no scaling. The symbol “%” is optional.

`width` : ***length or percentage***
You can set the figure width in the following units: “em”, “ex”, “px”,“in” ,“cm”, “mm”, “pt”, “pc”, “%”.

`height` : ***length***
You can set the figure height in the following units: “em”, “ex”, “px”, “in”, “cm”, “mm”, “pt”, “pc”.

`alt` : ***text***
Text to be displayed if the figure cannot be displayed or if the reader is using assistive technologies. Generally entails a short description of the figure.

`align` : ***“left”, “center”, or “right”***
Align the figure left, center, or right. Default alignment is center.

`name` : ***text***
A unique identifier for your figure that you can use to reference it with `{ref}` or `{numref}` roles. Cannot contain spaces or special characters.

`figclass` : ***text***
Value of the figure’s class attribute which can be used to add custom CSS or JavaScript. Predefined options include:

- *“margin”* : Display figure on the margin
- *“margin-caption”* : Display figure caption on the margin

Add metadata to your book pages

Metadata is information about a book or its content. It is often used to control the behavior of Jupyter Book and its features. This is a short guide to how metadata is added to various kinds of content in Jupyter Book.

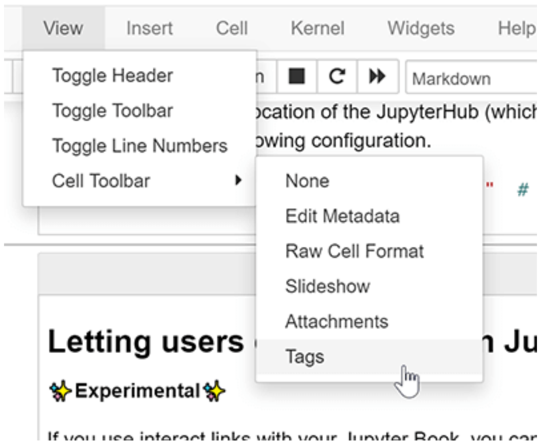
Add metadata to notebooks

You can control the behaviour of Jupyter Book by putting custom tags in the metadata of your cells. This allows you to do things like automatically hide code cells as well as add interactive widgets to cells.

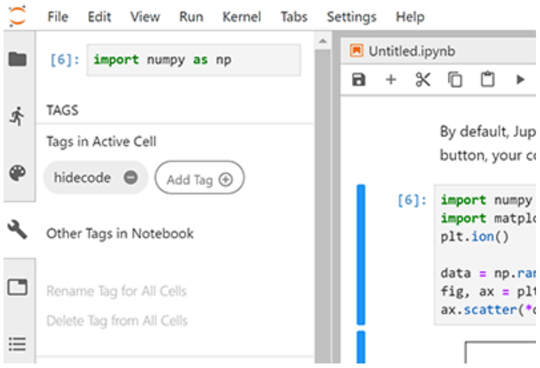
Adding tags using notebook interfaces

There are two straightforward ways to add metadata to cells:

1. **Use the Jupyter Notebook cell tag editor**. The Jupyter Notebook ships with a cell tag editor by default. This lets you add cell tags to each cell quickly.
   To enable the cell tag editor, click `View -> Cell Toolbar -> Tags`. This will enable the tags UI. Here’s what the menu looks like.



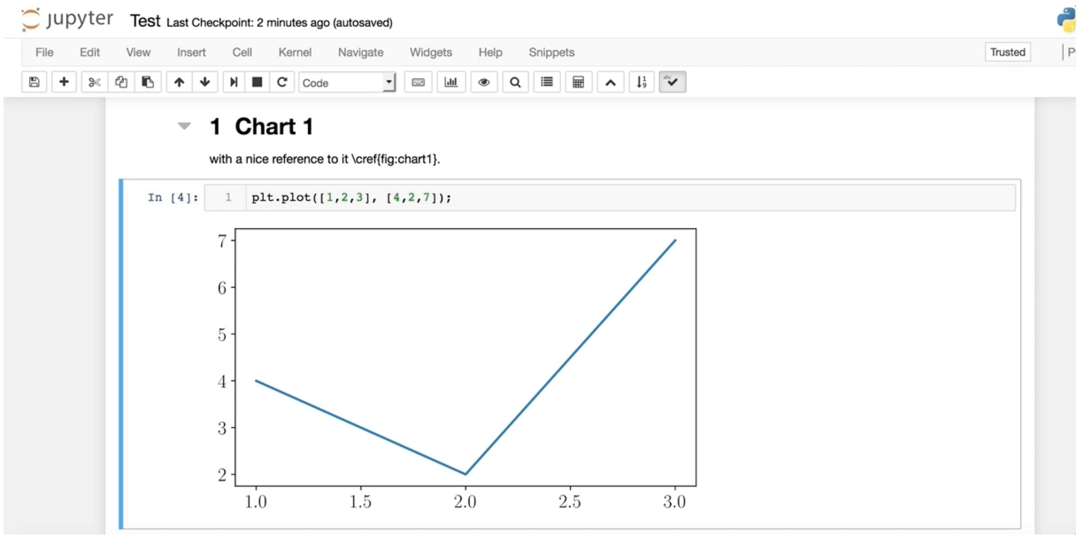
2. **Use the JupyterLab Cell Tags plugin**. JupyterLab is an IDE-like Jupyter environment that runs in your browser. It has a "cell tags" plugin built-in, which exposes a user interface that lets you quickly insert cell tags. You'll find tags under the "wrench" menu section. Here's what the tags UI in JupyterLab looks like.



```
[6]: import numpy as np
```

Tags are actually just a special section of cell level metadata. There are three levels of metadata:

- For notebook level: in the Jupyter Notebook Toolbar go to `Edit -> Edit Notebook Metadata`
- For cell level: in the Jupyter Notebook Toolbar go to `View -> Cell Toolbar -> Edit Metadata` and a button will appear above each cell.
- For output level: using e.g. `IPython.display.display(obj,metadata={"tags": [])`, you can set metadata specific to a certain output (but Jupyter Book does not utilize this just yet).


jupyter
Test Last Checkpoint: 2 minutes ago (autosaved)
File
Edit
View
Insert
Cell
Kernel
Navigate
Widgets
Help
Snippets
Trusted
Code
1 Chart 1
with a nice reference to it \cref{fig:chart1}.
In [4]:

```
plt.plot([1,2,3], [4,2,7]);
```

Add tags using MyST Markdown notebooks

If you're writing notebooks with MyST Markdown, then you can add tags to each code cell when you write the `{code-cell}` block. For example, below we:

~~~
```{code-cell}
:tags: [tag1,tag2,tag3]
print("some code")
```
~~~

Will create a code cell with those three tags attached to it. For more information about MyST Markdown notebooks, see Notebooks written entirely in Markdown.

Add tags using Python code

Sometimes you'd like to quickly scan through a notebook's cells in order to add tags based on the content of the cell. For example, you might want to hide any cell with an import statement in it using the `remove-input` tag.

Here's a short Python snippet to accomplish something close to this. First change directories into the root of your book folder, and then run the script below as a Python script or within a Jupyter Notebook (modifying as necessary for your use case). Finally, check the changes that will be made and commit them to your repository.

```python
import nbformat as nbf
from glob import glob

# Collect a list of all notebooks in the content folder
notebooks = glob("./content/**/*.ipynb", recursive=True)

# Text to look for in adding tags
text_search_dict = {
    "# HIDDEN": "remove-cell",  # Remove the whole cell
    "# NO CODE": "remove-input",  # Remove only the input
    "# HIDE CODE": "hide-input"  # Hide the input w/ a button to show
}

# Search through each notebook and look for the text, add a tag if necessary
for ipath in notebooks:
    ntbk = nbf.read(ipath, nbf.NO_CONVERT)

    for cell in ntbk.cells:
        cell_tags = cell.get('metadata', {}).get('tags', [])
        for key, val in text_search_dict.items():
            if key in cell['source']:
                if val not in cell_tags:
                    cell_tags.append(val)
        if len(cell_tags) > 0:
            cell['metadata']['tags'] = cell_tags

    nbf.write(ntbk, ipath)
```

Add metadata to notebook cells

Stuff about tags etc

Hide or remove content

It's possible to control which content shows up in your book. For example, you may want to display a complex visualization to illustrate an idea, but don't want the page to be cluttered with a large code cell that generated the visualization. In other cases, you may want to remove a code cell entirely.

In this case, you have two options:

- **Hiding** content provides a button that lets readers reveal the content.
- **Removing** content prevents it from making it into your book. It will be entirely gone (though still present in the source files)

There are two ways to hide content:

- To hide Markdown, use the `{toggle}` directive.
- To hide or remove code cells or their outputs, use **notebook cell tags**.

We'll cover each alternative below.

See also

Add metadata to notebooks

Hide Markdown using MyST Markdown

There are two ways to hide Markdown content

- you can use the `{toggle}` directive to hide arbitrary blocks of content
- you can use the `dropdown` class with admonitions to turn them into dropdowns

Both allow you to wrap chunks of Markdown in a button that lets users show and hide the content.

The `{toggle}` directive

You can activate a toggleable behavior in Markdown with the `{toggle}` directive like so:

~~~
```{toggle}
Some hidden toggle content!

![](../images/cool.jpg)
```
~~~

This results in:

Note that if you'd like to **show the toggle content by default**, you can add the `:show:` flag when you use `{toggle}`, like so:

~~~
```{toggle} Click the button to reveal!
:show:
Some hidden toggle content!

![](../images/cool.jpg)
```
~~~

Toggle admonition content with dropdowns

You can also **add toggle buttons to admonition blocks**, effectively making them dropdown blocks. Users will see the admonition title, but will need to click in order to reveal the content. To do so, add the `dropdown` class to any admonition. For example, the code

~~~
```{admonition} Click the button to reveal!
:class: dropdown
Some hidden toggle content!

![](../images/cool.jpg)
```
~~~

results in:

Click the button to reveal!

See Dropdown admonitions for more information on admonition dropdowns.

Hide code cell content

You can hide most cell elements of a page. The sections below describe how to hide each using cell tags in MyST Markdown. If you're working with `.ipynb` files, see the cell tags guide on adding cell tags to notebooks in Jupyter Notebook or JupyterLab.

If an element is hidden, Jupyter Book will display a small button to the right of the old location for the hidden element. If a user clicks the button, the element will be displayed.

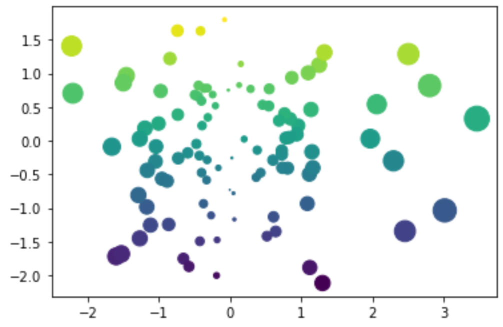
Hide cell inputs

If you add the tag `hide-input` to a cell, then Jupyter Book will hide the cell but display the outputs.

Here’s an example of cell metadata that would trigger the “hide code” behavior:

```
{
    "tags": [
        "hide-input",
    ]
}
```

For example, notice the cell below contains the `hide-input` tag:



Note how we only see the output by default. Now try clicking the button to the right of the empty spot above!

Hide cell outputs

You can also hide the *outputs* of a cell. For example, if you’d like to ask users to think about what the output will look like first before viewing an answer. To do so, add the following tag to your cell:

```
{
    "tags": [
        "hide-output",
    ]
}
```

```
# This cell should have its output hidden!
data = np.random.randn(2, 100)
fig, ax = plt.subplots()
ax.scatter(*data, c=data[1], s=100*np.abs(data[0]));
```

Hide entire code cells

If you’d like to hide the whole code cell (both inputs and outputs) just add this tag to the cell metadata, like so:

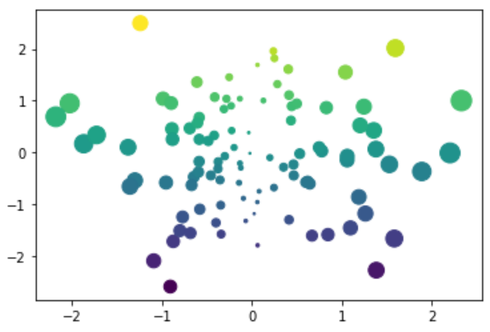
```
{
    "tags": [
        "hide-cell",
    ]
}
```

Removing code cell content

In the above examples, we are only *hiding* parts of the cell, with the option that readers can reveal them if they wish. However, if you’d like to completely **remove** the respective parts, so that their contents do not make it into the book’s HTML, you may use the appropriate `remove-` tags, i.e. `remove-input`, `remove-output` and `remove-cell`.

Remove cell inputs

The following cell has its inputs removed with `remove-input`. Note that in this case, there is no button available to show the input contents, the entire input cell is gone!



Remove cell outputs

Similar to hiding inputs, it is also possible to hide the outputs of a cell with `remove-output`:

```
{
    "tags": [
        "remove-output",
    ]
}
```

Remove an entire code cell

You can also remove **both** the inputs and outputs of a cell, in which case it won't show up in your book at all. These cells remain in the notebook file itself, so they'll show up if readers click on a JupyterHub or Binder link from a page.

To remove both the inputs and outputs of a cell, add the tag `remove-cell` to the tags of the cell. Here's an example of cell metadata that would trigger the "remove cell" behavior:

```
{
    "tags": [
        "remove-cell",
    ]
}
```

These cells will be entirely removed from each book page - remember that if you'd like to optionally display the inputs of a cell instead, you should use the `hide-input` tag.

For example, there's a cell below this text that won't make it into the final book, because it has been removed!

Remove empty cells

You don't need to do anything to remove empty cells from your pages. Jupyter Book will remove these automatically. Any cell with *only* whitespace will be removed.

For example, in the notebook for this page there are two cells above this text. Both only contain whitespace. Both are gone from the final output.

Types of content source files

Jupyter Book supports many kinds of source files for your book's content. These sections cover the major types of content and how you can control their behavior in Jupyter Book. See the list of sections to the left for information about each type.

Section table of contents

Markdown files

You can write content in regular Markdown files (e.g., files ending in `.md`). Jupyter Book supports any Markdown syntax that is supported by Jupyter notebooks. Jupyter Notebook Markdown is an extension of a flavour of Markdown called CommonMark Markdown. It has many elements for standard text processing, though it lacks a lot of features used for publishing and documentation.

Note

If you'd like a more in-depth overview and guide to CommonMark Markdown, see the CommonMark Markdown tutorial.

This page describes some basic features of the Jupyter Notebook Markdown, and how to include them with your book.

Jupyter Book also supports a fancier version of Markdown called **MyST Markdown**. This is a slightly extended flavour of Jupyter Notebook Markdowr It allows you to include citations and cross-references, and control more complex functionality like adding content to the margin. For more information, check out MyST Markdow overview.

Embedding media

Adding images

You can reference external media like images from your Markdown file. If you use relative paths, then they will continue to work when the Markdown files are copied over, so long as they point to a file that's inside of the repository.

Here's an image relative to the book content root

It was generated with this code:

```
![C-3PO_droid](../images/C-3PO_droid.png)
```

See also

Images and figures for more information.

Adding movies

You can even embed references to movies on the web! For example, here's a little GIF for you!

This will be included in your book when it is built.

Mathematics

For HTML outputs, Jupyter Book uses the excellent MathJax library, along with the default Jupyter Notebook configuration, for rendering mathematics from LaTeX-style syntax.

For example, here's a mathematical expression rendered with MathJax:

$$
\begin{aligned}
P(A_1 \cup A_2 \cup A_3) &= P(B \cup A_3) \\
&= P(B) + P(A_3) - P(BA_3) \\
&= P(A_1) + P(A_2) - P(A_1 A_2) + P(A_3) - P(A_1 A_3 \cup A_2 A_3) \\
&= \sum_{i=1}^{3} P(A_i) - \sum_{1 \le i < j \le 3} \sum P(A_i A_j) + P(A_1 A_2 A_3)
\end{aligned}
$$

See also

Math and equations for more information.

Block-level mathematics

You can include block-level mathematics by wrapping your formulas in `$$` characters. For example, the following block:

```
$$
wow = its^{math}
$$
```

Results in this output:

$$wow = its^{math}$$

You can also include math blocks by using LaTeX-style syntax using `\begin{align*}`. For example, the following block:

```
\begin{align*}
yep = its_{more}^{math}
\end{align*}
```

Results in:

$$yep = its_{more}^{math}$$

Important

This requires the `amsmath` MyST extension to be enabled.

Extended Markdown with MyST Markdown

In addition to CommonMark Markdown, Jupyter Book also supports a more fully-featured version of Markdown called **MyST Markdown**. This is a superset of CommonMark that includes syntactic pieces that are useful for publishing computational narratives. For more information about MyST Markdown, see MyST Markdown overview.

Jupyter Notebook files

You can create content with Jupyter notebooks. For example, the content for the current page is contained in `this notebook file`.

Jupyter Book supports all Markdown that is supported by Jupyter Notebook. This is mostly a flavour of Markdown called CommonMark Markdown with minor modifications. For more information about writing Jupyter-flavoured Markdown in Jupyter Book, see Markdown files.

If you'd like to write in plain-text files, but still keep a notebook structure, yo can write Jupyter notebooks with MyS Markdown, which are then automatically converted to notebooks. See Notebooks written entirely in Markdown for more details.

Code blocks and image outputs

Jupyter Book will also embed your code blocks and output in your book. For example, here's some sample Matplotlib code:

```
from matplotlib import rcParams, cycler
import matplotlib.pyplot as plt
import numpy as np
plt.ion()
```

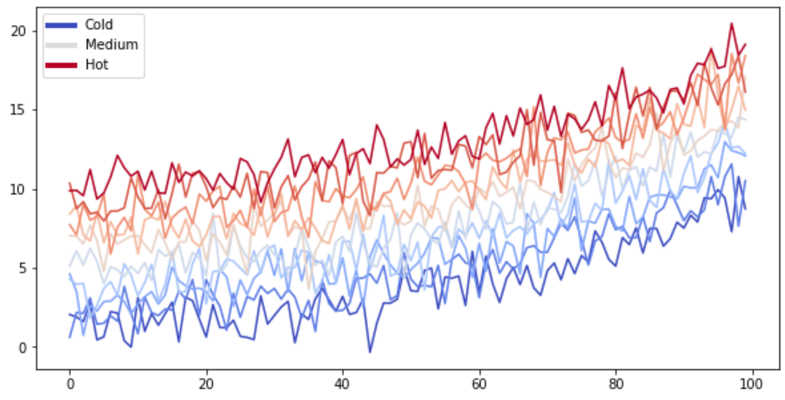
```
<matplotlib.pyplot._IonContext at 0x7fe08e796610>
```

```
# Fixing random state for reproducibility
np.random.seed(19680801)

N = 10
data = [np.logspace(0, 1, 100) + np.random.randn(100) + ii for ii in range(N)]
data = np.array(data).T
cmap = plt.cm.coolwarm
rcParams['axes.prop_cycle'] = cycler(color=cmap(np.linspace(0, 1, N)))


from matplotlib.lines import Line2D
custom_lines = [Line2D([0], [0], color=cmap(0.), lw=4),
                Line2D([0], [0], color=cmap(.5), lw=4),
                Line2D([0], [0], color=cmap(1.), lw=4)]

fig, ax = plt.subplots(figsize=(10, 5))
lines = ax.plot(data)
ax.legend(custom_lines, ['Cold', 'Medium', 'Hot']);
```



Note that the image above is captured and displayed in your site.

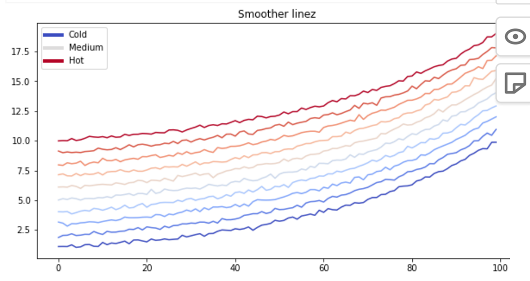
Removing content before publishing

You can also remove some content before publishing your book to the web. For reference, `you can download the notebook content for this page`.

You can **remove only the code** so that images and other output still show up.

this plot *will* show up in the textbook.

```
[Text(0.5, 1.0, 'Smoother linez')]
```



You can also pop out content to th side!

For more information on how to do this check out the [Sidebar content](#) section

Which works well if you'd like to quickly display cell output without cluttering your content with code. This works for any cell output, like a Pandas DataFrame.

	Word A	Word B
0	hi	there
1	this	is
2	a	DataFrame

See [Removing code cell content](#) for more information about hiding and removing content.

Interactive outputs

We can do the same for *interactive* material. Below we'll display a map using [folium](#). When your book is built, the code for creating the interactive map is retained.

This will only work for some packages. They need to be able to output standalone HTML/Javascript, and not depend on an underlying Python kernel to work.

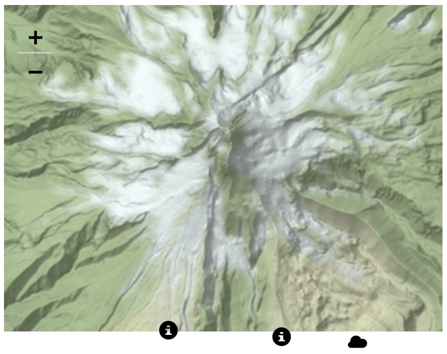
```python
import folium
m = folium.Map(
    location=[45.372, -121.6972],
    zoom_start=12,
    tiles='Stamen Terrain'
)

folium.Marker(
    location=[45.3288, -121.6625],
    popup='Mt. Hood Meadows',
    icon=folium.Icon(icon='cloud')
).add_to(m)

folium.Marker(
    location=[45.3311, -121.7113],
    popup='Timberline Lodge',
    icon=folium.Icon(color='green')
).add_to(m)

folium.Marker(
    location=[45.3300, -121.6823],
    popup='Some Other Location',
    icon=folium.Icon(color='red', icon='info-sign')
).add_to(m)

m
```


Trust Notebook

Leaflet (https://leafletjs.com) | Map tiles by Stamen Design (http://stamen.com), under CC BY 3.0 (http://creativecommons.org/licenses/by/3.0). Data by © OpenStreetMap (http://openstreetmap.org), under CC BY SA (http://creativecommons.org/licenses/by-sa/3.0).

Rich outputs from notebook cells

Because notebooks have rich text outputs, you can store these in your Jupyter Book as well! For example, here is the command line help menu, see how it is nicely formatted.

```
!jupyter-book build --help
```

```
Usage: jupyter-book build [OPTIONS] PATH_SOURCE

  Convert your book's or page's content to HTML or a PDF.

Options:
  --path-output TEXT              Path to the output artifacts
  --config TEXT                   Path to the YAML configuration file
                                  (default: PATH_SOURCE/_config.yml)

  --toc TEXT                      Path to the Table of Contents YAML file
                                  (default: PATH_SOURCE/_toc.yml)

  -W, --warningiserror            Error on warnings.
  -n, --nitpick                   Run in nit-picky mode, to generates warnings
                                  for all missing references.

  --keep-going                    With -W, do not stop the build on the first
                                  warning, instead error on build completion

  --all                           Re-build all pages. The default is to only
                                  re-build pages that are new/changed since
                                  the last run.

  --builder [html|dirhtml|pdfhtml|latex|pdflatex|linkcheck|custom]
                                  Which builder to use.
  --custom-builder TEXT           Specify alternative builder name which
                                  allows jupyter-book to use a builderprovided
                                  by an external extension. This can only be
                                  used when using--builder=custom

  -v, --verbose                   increase verbosity (can be repeated)
  -q, --quiet                     -q means no sphinx status, -qq also turns
                                  off warnings

  --individualpages               [pdflatex] Enable build of PDF files for
                                  each individual page

  -h, --help                      Show this message and exit.
```

And here is an error. You can mark notebook cells as “expected to error” by adding a `raises-exception` tag to them.

```
this_will_error
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-9-09f61459889d> in <module>
----> 1 this_will_error

NameError: name 'this_will_error' is not defined
```

More features with Jupyter notebooks

There are many other features of Jupyter notebooks to take advantage of, such as automatically generating Binder links for notebooks or connecting your content with a kernel in the cloud. For more information browse the pages in this site, and Formatting code outputs in particular.

Notebooks written entirely in Markdown

It is possible to store Jupyter notebooks in plain Markdown. This allows you to define a notebook structure entirely using MyST Markdown. For more information about MyST Markdown, see MyST Markdown overview.

Notebooks with Markdown can be read in, executed, and cached by Jupyter Book (see Execute and cache your pages for information on how to cache pages). This allows you to store all of your notebook content in a text format that is much nicer for version control software, while still having all the functionality of a Jupyter notebook.

Note

MyST notebooks uses [MyST-NB to convert between ipynb and text files][myst-nb:index]. See its documentation for more information.

To see an example of a MyST notebook, you can look at many of the pages of this documentation. For example, see `../interactive/hiding.md` and `../content/layout.md`.

Create a MyST notebook with Jupytext

The easiest way to create a MyST notebook is to use Jupytext, a tool that allows for two-way conversion between `.ipynb` and a variety of text files.

You can convert an `.ipynb` file to a MyST notebook with the following command:

```
jupytext mynotebook.ipynb --to myst
```

A resulting `mynotebook.md` file will be created. This can then be used as a page in your book.

Important

For full compatibility with `myst-parser`, it is necessary to use `jupytext>=1.6.0`.

Jupytext can also **automatically synchronize an `.ipynb` file with your Markdown**. To do so, use a Jupyter interface such as Jupyter Lab or the classic notebook interface and follow the Jupytext instructions for paired notebooks.

Markdown takes precedence

If **both** an `.ipynb` and a `.md` file exist in your book's folders, then the `.md` file will take precedence!

Convert a Markdown file into Jupytext MyST Markdown

Jupyter Book has a small CLI to provide common functionality for manipulating and creating MyST Markdown files that synchronize with Jupytext. To add Jupytext syntax to a Markdown file (that will tell Jupytext it is a MyST Markdown file), run the following command:

```
jupyter-book myst init mymarkdownfile.md --kernel kernelname
```

If you do not specify `--kernel`, then the default kernel will be used *if there is only one available*. If there are multiple kernels available, you must specify one manually.

Structure of MyST notebooks

Let's take a look at the structure that Jupytext creates, which you may also use to create a MyST notebook from scratch. First, let's take a look at a simple MyST notebook:

~~~
---
jupytext:
  formats: md:myst
  text_representation:
    extension: .md
    format_name: myst
kernelspec:
  display_name: Python 3
  language: python
  name: python3
---

# My simple notebook

Some **intro Markdown**!

```{code-cell} ipython3
:tags: [mytag]

print("A python cell")
```

## A section

And some more Markdown...
~~~

There are three main sections to notice:

Frontmatter YAML

MyST notebooks need special frontmatter YAML to tell Jupytext that they can be converted to `.ipynb` files. The frontmatter YAML block

```
---
jupytext:
  formats: md:myst
  text_representation:
    extension: .md
    format_name: myst
kernelspec:
  display_name: Python 3
  language: python
  name: python3
---
```

tells Jupytext that the file is in `myst` format, and that its code should be run with a Python 3 kernel.

Remember that Jupyter always define
one, and only one, kernel per notebo

Code cells

Code blocks in MyST notebooks are defined with the following MyST directive:

~~~
```{code-cell}
your-code
```
~~~

You can optionally add extra metadata to the code cell, which will be converted into cell metadata in the `.ipynb` file. For example, you can add tags to your code cell like so:

~~~
```{code-cell}
:tags: [tag1, tag2, tag3]
your-code
```
~~~

You may also explicitly pass the kernel name after `{code-cell}` to make it clear which kernel you are running. For example:

~~~
```{code-cell} python3
your-code
```
~~~

However, remember that there is only one kernel allowed per page.

Markdown content

Everything in-between your code cells is parsed as Markdown content using the MyST Markdown parser. See MyST Markdown overview for more information about MyST Markdown.

To explicitly split up Markdown content into two Markdown cells, use the following pattern:

```
Content in one Markdown cell

+++

Content in another Markdown cell
```

You may also attach metadata to the cell by adding a Python dictionary after the `+++`. For example, to add tags to the second cell above:

```
Content in one Markdown cell

+++ {"tags": ["tag1", "tag2", "tag3"]}

Content in another Markdown cell
```

Warning

Please note that cell breaks and metadata specified in MyST files via the `+++` syntax only propagate to their `.ipynb` counterpart. When generating the book's HTML, *Markdown cell* information is discarded to avoid conflicting hierarchies in the structure of the document. In other words, only *code cell* tags have an effect on the generated HTML.

Custom notebook formats and Jupytext

You can designate additional file types to be converted to notebooks and then executed/parsed in the same manner as regular notebooks.

Tip

This page itself is written as an RMarkdown notebook!

```
sphinx:
  config:
    nb_custom_formats:
        .mysuffix: mylibrary.converter_function
```

- The string should be a Python function that will be loaded by `import mylibrary.converter_function`
- The function should take a file's contents (as a `str`) and return an nbformat.NotebookNode

If the function takes additional keyword arguments, then you can specify these as a dictionary in a second argument. For example this is what the default conversion would look like:

```
sphinx:
  config:
    nb_custom_formats:
        .ipynb:
            - nbformat.reads
            - as_version: 4
```

Important

By default, Markdown cells in the notebook will be parsed using the same MyST parser configuration as for other Markdown files.

But, if this is incompatible with your file format, then you can specify for the Markdown to be parsed as **strictly CommonMark**, using a third argument:

```yaml
sphinx:
  config:
    nb_custom_formats:
        .ipynb:
            - nbformat.reads
            - as_version: 4
            - true
```

Finally, for text-based formats, MyST-NB also searches for an optional `source_map` key in the output notebook's metadata. This key should be a list mapping each cell to the starting line number in the original source file, for example for a notebook with three cells:

```json
{
  "metadata": {
    "source_map": [10, 21, 53]
  }
}
```

This mapping allows for "true" error reporting, as described in Sphinx Error Reporting.

Using Jupytext

Jupytext is an excellent Python tool for two-way conversion between Jupyter Notebook `.ipynb` files and a variety of text-based files.

Jupyter Book natively supports the Jupytext file format: notebooks with MyST Markdown, but you can add other formats like RMarkdown or Python files.

The configuration looks like:

```yaml
sphinx:
  config:
    nb_custom_formats:
        .Rmd:
            - jupytext.reads
            - fmt: Rmd
```

Warning

Note that some execution features (such as in-line code execution in RMarkdown) are not available in Jupyter Book.

Now you can use RMarkdown blocks:

~~~
```{python echo=TRUE}
print("Hallo I'm an RMarkdown block!")
```
~~~

```python
print("Hallo I'm an RMarkdown block!")
```

```
Hallo I'm an RMarkdown block!
```

Important

For full compatibility with `myst-parser`, it is necessary to use `jupytext>=1.6.0`.

Convert a Jupytext file into a MyST notebook

Alternatively, if you'd like to convert your pre-existing Jupytext files into the MyST notebook format, to use directly in your book, install Jupytext and then run the following command:

```
jupytext --to myst path/to/yourfile
```

Note that you may also pass a wildcard that will be used to convert multiple files. For example:

```
jupytext --to myst ./*.py
```

See the Jupytext CLI documentation for more information.

reStructuredText files

In addition to writing your content in Markdown, Jupyter Book also supports writing content in reStructuredText, another markup language that is common in the Python documentation community.

> **Warning**
>
> Writing content in reStructuredText is only recommended for users who are already familiar with it. For others, we recommend using MyST Markdown, which has all of the same features of rST and Sphinx, but with a Markdown flavour.

Because Jupyter Book uses Sphinx under the hood, any document that is written in rST for the Sphinx ecosystem should also work with Jupyter Book. This is particularly useful if you've already got a significant amount of documentation written in rST and you'd like to try it out with Jupyter Book.

For more information on writing content with reStructuredText, we recommend reading the Sphinx rST documentation.

Including reStructuredText in Markdown

To insert rST into Markdown, you can use the eval-rst directive:

~~~
```{eval-rst}
.. note::

   A note written in reStructuredText.

.. include:: ./include-rst.rst
```
~~~

> **Note**
>
> A note written in reStructuredText.

Hallo I'm from an rST file, myst-parser:with a reference about using autodoc.

Allowed content types

In general, these are the types of content supported by Jupyter Book (along with links to their section in this book):

Markdown files
These are text files written in either CommonMark or in MyST Markdown.

Jupyter notebooks
AKA, `.ipynb` files. These files can contain Markdown cells with MyST Markdown.
A Jupyter notebook can utilise any program kernel that implements the Jupyter messaging protocol for executing code. There are kernels available for Python, Julia, Ruby, Haskell and many other languages.

MyST Markdown notebooks
These are Markdown files (ending in `.md`) that will be *converted to a notebook and executed*.

reStructuredText.

These are text files used by the Sphinx documentation engine (which is used by Jupyter Book). It is recommended to use MyST Markdown instead.

Custom notebook formats

Any other file type can be *auto-converted* before execution by assigning it a custom Python function, for example those provided by the Jupytext conversion tool.

Rules for all content types

There are a few things that are true for all content types. Here is a short list:

- **Files must have a title**. Generally this means that they must begin with a line that starts with a single `#`
- **Use only one top-level header**. Because each page must have a clear title, it must also only have one top-level header. You cannot have multiple headers with single `#` tag in them.
- **Headers should increase linearly**. If you're inside of a section with one `#`, then the next nested section should start with `##`. Avoid jumping straight from `#` to `###`.

Two-way conversion between text-files and `.ipynb` files

For information about how to convert between text files and `.ipynb` files for use with Jupyter Book, see Convert a Jupytext file into a MyST notebook.

Build your book

Clean your book's generated files

It is possible to "clean up" the files that you generate when you build your book. This is often useful if you have recently changed a lot of content in order to ensure that you build your book from a clean slate.

You can clean up your book's generated content by running the following command:

```
jupyter-book clean mybookname/
```

By default, this will delete all folders inside `mybookname/_build` *except* for a folder called `.jupyter_cache`. This ensures that the *content* of your book will be regenerated, while the cache that is generated by *running your book's code* will not be deleted (because regenerating it may take some time).

To delete the `.jupyter_cache` folder as well, add the `--all` flag like so:

```
jupyter-book clean mybookname/ --all
```

This will entirely remove the folders in the `_build/` directory.

Disable building files that aren't specified in the TOC

By default, Jupyter Book will build all files that are in your book's folder, regardless of whether they are specified in the Table of Contents. To disable this behavior and *only* build files that are specified in the TOC, use the following pattern in `_config.yml`:

```
only_build_toc_files: true
```

Note that files that are in *hidden folders* (e.g. in `.github` or `.venv`) will still be built even if they are not specified in the TOC. You should exclude these files explicitly.

Debug your book's build process

When debugging your book build, the following options can be helpful:

```
jupyter-book build -W -n --keep-going mybookname/
```

This will check for missing references (`-n`), turning them into errors (`-W`), but will still attempt to run the full build (`--keep-going`), so that you can see all errors in one run.

You can also use `-v` or `-vvv` to increase verbosity.

A list of book output types

You can build a variety of outputs using Jupyter Book. To choose a different builder, use the `--builder <builder-name>` configuration when running `jupyter-book build` from the command-line. Here is a list of builders that are available to you:

- `html`: HTML outputs (default)
- `singlehtml`: A single HTML page for your book
- `dirhtml`: HTML outputs with `<filename>/index.html` structure.
- `pdfhtml`: Build a PDF via HTML outputs (see Build a PDF from your book HTML)
- `linkcheck`: Run the Sphinx link checker (see Check external links in your book)
- `latex`: Build Latex files for your book
- `pdflatex`: Build a PDF of your book via Latex (see Build a PDF using LaTeX)

Execute your computational content

This section covers topics related to **executing** your content and interweaving **computational material** with your narrative. For example, if you're writing your content as *Jupyter Notebooks* and wish for the output of cells to be included with your book.

See the sections below to get started.

Formatting code outputs

The formatting of code outputs is highly configurable. Below we give examples of how to format particular outputs and even insert outputs into other locations of the document.

The MyST cheat sheet provides a list of `code-cell` tags available

See also

The MyST-NB documentation, for how to fully customize the output renderer.

Library output formatting

Many libraries support their own HTML output formatting, and this generally carries over to Jupyter Book outputs as well.

For example, the following cell uses Pandas to format cells based on their values:

	A	B	C	D	E
0	1.000000	1.329212	nan	-0.316280	-0.990810
1	2.000000	-1.070816	-1.438713	0.564417	0.295722
2	3.000000	-1.626404	0.219565	0.678805	1.889273
3	4.000000	0.961538	0.104011	nan	0.850229
4	5.000000	1.453425	1.057737	0.165562	0.515018
5	6.000000	-1.336936	0.562861	1.392855	-0.063328
6	7.000000	0.121668	1.207603	-0.002040	1.627796
7	8.000000	0.354493	1.037528	-0.385684	0.519818
8	9.000000	1.686583	-1.325963	1.428984	-2.089354
9	10.000000	-0.129820	0.631523	-0.586538	0.290720

See the Pandas Styling docs for more information about styling DataFrames, and check out the documentation of your library of choice to see if they support similar features.

Scrolling cell outputs

The traditional Jupyter Notebook interface allows you to toggle **output scrolling** for your cells. This allows you to visualize part of a long output without it taking up the entire page.

You can trigger this behavior in Jupyter Book by adding the following tag to a cell's metadata:

```
{
    "tags": [
        "output_scroll",
    ]
}
```

For example, the following cell has a long output, but will be scrollable in the book:

```
for ii in range(40):
    print(f"this is output line {ii}")
```

```
this is output line 0
this is output line 1
this is output line 2
this is output line 3
this is output line 4
this is output line 5
this is output line 6
this is output line 7
this is output line 8
this is output line 9
this is output line 10
this is output line 11
this is output line 12
this is output line 13
this is output line 14
this is output line 15
this is output line 16
this is output line 17
this is output line 18
this is output line 19
```

Images

For any image types output by the code, we can apply formatting *via* cell metadata. Then for the image we can apply all the variables of the standard image directive:

- **width**: length or percentage (%) of the current line width
- **height**: length
- **scale**: integer percentage (the "%" symbol is optional)
- **align**: "top", "middle", "bottom", "left", "center", or "right"
- **classes**: space separated strings
- **alt**: string

Units of length are: 'em', 'ex', 'px', 'in', 'cm', 'mm', 'pt', 'pc'

We can also set a caption (which is rendered as CommonMark) and name by which to reference the figure. The code

~~~
```{code-cell} ipython3
---
render:
  image:
    width: 200px
    alt: fun-fish
    classes: shadow bg-primary
  figure:
    caption: |
      Hey everyone its **party** time!
    name: fun-fish
---
from IPython.display import Image
Image("../images/fun-fish.png")
```
~~~

produces the following code cell and figure:

```
from IPython.display import Image
Image("../images/fun-fish.png")
```

Hey everyone its **party** time!

Now we can link to the image from anywhere in our documentation: swim to the fish

See also

Add metadata to notebooks

Markdown

Markdown output is parsed by MyST-Parser, currently with the parsing set to strictly CommonMark.

The parsed Markdown is then integrated into the wider context of the document. This means it is possible, for example, to include internal references:

```
from IPython.display import display, Markdown
display(Markdown('**_some_ markdown** and an [internal reference]
(use/format/markdown)!'))
```

***some* markdown** and an internal reference!

and even internal images can be rendered, as the code below exemplifies:

```
display(Markdown('![figure](../images/logo.png)'))
```

jupyter {book}

ANSI outputs

By default, the standard output/error streams and text/plain MIME outputs may contain ANSI escape sequences to change the text and background colors.

```
import sys
print("BEWARE: \x1b[1;33;41mugly colors\x1b[m!", file=sys.stderr)
print("AB\x1b[43mCD\x1b[35mEF\x1b[1mGH\x1b[4mIJ\x1b[7m"
      "KL\x1b[49mMN\x1b[39mOP\x1b[22mQR\x1b[24mST\x1b[27mUV")
```

```
ABCDEFGHIJKLMNOPQRSTUV
```

```
BEWARE: ugly colors!
```

This uses the built-in `AnsiColorLexer` pygments lexer. You can change the lexer used in the `_config.yml`, for example to turn off lexing:

```
sphinx:
  config:
    nb_render_text_lexer: "none"
```

The following code shows the 8 basic ANSI colors it is based on. Each of the 8 colors has an "intense" variation, which is used for bold text.

```
text = " XYZ "
formatstring = "\x1b[{}m" + text + "\x1b[m"

print(
    " " * 6
    + " " * len(text)
    + "".join("{:^{}}".format(bg, len(text)) for bg in range(40, 48))
)
for fg in range(30, 38):
    for bold in False, True:
        fg_code = ("1;" if bold else "") + str(fg)
        print(
            " {:>4} ".format(fg_code)
            + formatstring.format(fg_code)
            + "".join(
                formatstring.format(fg_code + ";" + str(bg)) for bg in range(40, 48)
            )
        )
```

```
            40   41   42   43   44   45   46   47
   30  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ
 1;30  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ
   31  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ
 1;31  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ
   32  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ
 1;32  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ
   33  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ
 1;33  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ
   34  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ
 1;34  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ
   35  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ
 1;35  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ
   36  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ
 1;36  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ
   37  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ
 1;37  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ  XYZ
```

Note

ANSI also supports a set of 256 indexed colors. This is currently not supported, but we hope to introduce it at a later date (raise an issue on the repository if you require it!).

Render priority

When Jupyter executes a code cell it can produce multiple outputs, and each of these outputs can contain multiple MIME media types to use with different output formats (like HTML or LaTeX).

MyST-NB stores a default priority dictionary for most of the common output builders, which you can also update in your `_config.yml`. For example, this is the default priority list for HTML:

```
sphinx:
  config:
    nb_render_priority:
      html:
      - "application/vnd.jupyter.widget-view+json"
      - "application/javascript"
      - "text/html"
      - "image/svg+xml"
      - "image/png"
      - "image/jpeg"
      - "text/markdown"
      - "text/latex"
      - "text/plain"
```

Insert code outputs into page content

You often wish to run analyses in one notebook and insert them in your documents elsewhere. For example, if you'd like to include a figure, or if you want to cite an analysis that you have run.

The `glue` tool from MyST-NB allows you to add a key to variables in a notebook, then display those variables in your book by referencing the key. It follows a two-step process:

Currently, `glue` only works with Pythor

- **Glue a variable to a name**. Do this by using the `myst_nb.glue` function on a variable that you'd like to re-use elsewhere in the book. You'll give the variable a name that can be referenced later.
- **Reference that variable from your page's content**. Then, when you are writing your content, insert the variable into your text by using a `{glue:}` role.

We'll cover each step in more detail below.

For more information about roles, see MyST Markdown overview.

Gluing variables in your notebook

You can use `myst_nb.glue()` to assign the value of a variable to a key of your choice. `glue` will store all of the information that is normally used to **display** that variable (i.e., whatever happens when you display the variable by putting it at the end of a code cell). Choose a key that you will remember, as you will use it later.

The following code glues a variable inside the notebook to the key `"cool_text"`:

```
from myst_nb import glue
my_variable = "here is some text!"
glue("cool_text", my_variable)
```

```
'here is some text!'
```

You can then insert it into your text. Adding `` {glue:}`cool_text` `` to your content results in the following: `'here is some text!'`.

Gluing numbers, plots, and tables

You can glue anything in your notebook and display it later with `{glue:}`. Here we'll show how to glue and paste **numbers and images**. We'll simulate some data and run a simple bootstrap on it. We'll hide most of this process below, to focus on the glueing part.

In the cell below, `data` contains our data, and `bootstrap_indices` is a collection of sample indices in each bootstrap. Below we'll calculate a few statistics of interest, and `glue()` them into the notebook.

```
# Calculate the mean of a bunch of random samples
means = data[bootstrap_indices].mean(0)
# Calculate the 95% confidence interval for the mean
clo, chi = np.percentile(means, [2.5, 97.5])

# Store the values in our notebook
glue("boot_mean", means.mean())
glue("boot_clo", clo)
glue("boot_chi", chi)
```

```
2.99758724978736
```

```
2.985312582852057
```

```
3.0098125309029817
```

By default, `glue` will display the value of the variable you are gluing. This is useful for sanity-checking its value at glue-time. If you'd like to **prevent display**, use the `display=False` option. Note that below, we also *overwrite* the value of `boot_chi` (but using the same value):

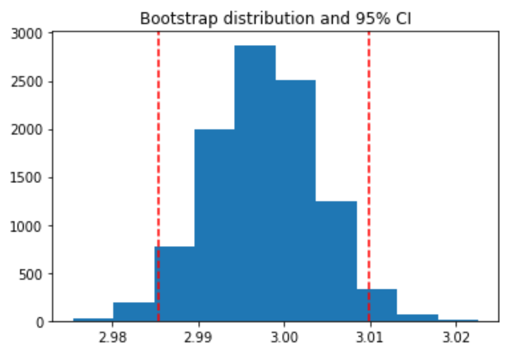
```
glue("boot_chi_notdisplayed", chi, display=False)
```

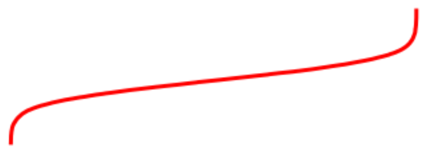
You can also glue visualizations, such as Matplotlib figures (here we use `display=False` to ensure that the figure isn't plotted twice):

```
# Visualize the historgram with the intervals
fig, ax = plt.subplots()
ax.hist(means)
for ln in [clo, chi]:
    ax.axvline(ln, ls='--', c='r')
ax.set_title("Bootstrap distribution and 95% CI")

# And a wider figure to show a timeseries
fig2, ax = plt.subplots(figsize=(6, 2))
ax.plot(np.sort(means), lw=3, c='r')
ax.set_axis_off()

glue("boot_fig", fig, display=False)
glue("sorted_means_fig", fig2, display=False)
```





The same can be done for `DataFrame`s (or other table-like objects) as well.

```
bootstrap_subsets = data[bootstrap_indices][:3, :5].T
df = pd.DataFrame(bootstrap_subsets, columns=["first", "second", "third"])
glue("df_tbl", df)
```

	first	second	third
0	2.864279	3.096170	3.040294
1	2.850520	3.582923	2.632284
2	3.182612	3.026727	3.013184
3	3.159771	2.793257	3.151598
4	3.087032	3.159038	3.132074

Tip

Since we are going to paste this figure into our document at a later point, you may wish to remove the output here, using the `remove-output` tag (see Removing code cell content).

Pasting glued variables into your page

Once you have glued variables to their names, you can then **paste** those variables into your text in your book anywhere you like (even on other pages). These variables can be pasted using one of the roles or directives in the `glue` *family*.

The `glue` role/directive

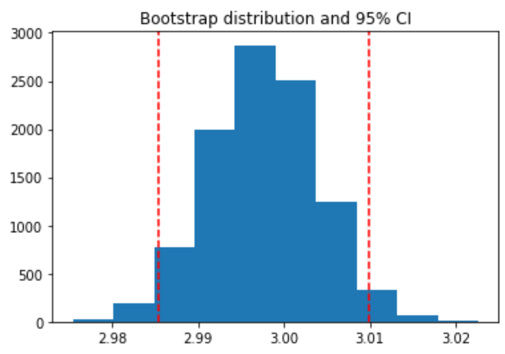
The simplest role and directive is `glue:any`, which pastes the glued output in-line or as a block respectively, with no additional formatting. Simply add this:

~~~
```{glue:} your-key
```
~~~

For example, we'll paste the plot we generated above with the following text:

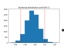
~~~
```{glue:} boot_fig
```
~~~

Here's how it looks:



Or we can paste in-line objects like so:

```
In-line text; {glue:}`boot_mean`, and a figure: {glue:}`boot_fig`.
```

In-line text; `2.99758724978736`, and a figure: .

Tip

We recommend using wider, shorter figures when plotting in-line, with a ratio around 6x2. For example, here's an in-line figure of sorted means from our bootstrap: . It can be used to make a visual point that isn't too complex! For more ideas, check out how sparklines are used.

Next we'll cover some more specific pasting functionality, which gives you more control over how the pasted outputs look in your pages.

Controlling the pasted outputs

You can control the pasted outputs by using a sub-command of `{glue:}`. These are used like so: `{glue:subcommand}`key``. These subcommands allow you to control more of the look, feel, and content of the pasted output.

Tip

When you use `{glue:}` you are actually using shorthand for `{glue:any}`. This is a generic command that doesn't make many assumptions about what you are gluing.

The `glue:text` role

The `glue:text` role is specific to text outputs. For example, the following text:

```
The mean of the bootstrapped distribution was {glue:text}`boot_mean` (95% confidence interval
{glue:text}`boot_clo`/{glue:text}`boot_chi`).
```

Is rendered as: The mean of the bootstrapped distribution was **2.99758724978736** (95% confidence interval **2.985312582852057**/**3.0098125309029817**)

Note

`glue:text` only works with glued variables that contain a `text/plain` output.

With `glue:text` we can **add formatting to the output**. This is particularly useful if you are displaying numbers and want to round the results. To add formatting, use this syntax:

- `{glue:text}`mykey:formatstring``

For example, `My rounded mean: {glue:text}`boot_mean:.2f`` will be rendered like this: My rounded mean: **3.00** (95% CI: **2.99**/**3.01**).

The `glue:figure` directive

With `glue:figure` you can apply more formatting to figure-like objects, such as giving them a caption and referenceable label. For example,

~~~
```{glue:figure} boot_fig
:figwidth: 300px
:name: "fig-boot"

This is a **caption**, with an embedded `{glue:text}` element: {glue:text}`boot_mean:.2f`!
```
~~~

produces the following figure:

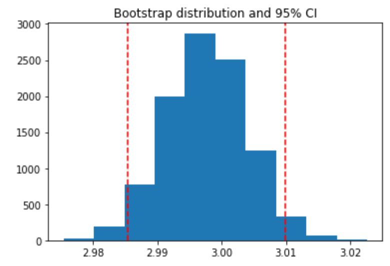


Fig. 10 This is a **caption**, with an embedded `{glue:text}` element: **3.00**!

Later, the code

```
Here is a {ref}`reference to the figure <fig-boot>`
```

can be used to reference the figure.

Here is a reference to the figure

Here's a table:

~~~
```{glue:figure} df_tbl
:figwidth: 300px
:name: "tbl:df"

A caption for a pandas table.
```
~~~

which gets rendered as

	first	second	third
0	2.864279	3.096170	3.040294
1	2.850520	3.582923	2.632284
2	3.182612	3.026727	3.013184
3	3.159771	2.793257	3.151598
4	3.087032	3.159038	3.132074

Fig. 11 A caption for a pandas table.

The `glue:math` directive

The `glue:math` directive is specific to LaTeX math outputs (glued variables that contain a `text/latex` MIME type), and works similarly to the Sphinx math directive. For example, with this code we glue an equation:

```
import sympy as sym
f = sym.Function('f')
y = sym.Function('y')
n = sym.symbols(r'\alpha')
f = y(n)-2*y(n-1/sym.pi)-5*y(n-2)
glue("sym_eq", sym.rsolve(f,y(n),[1,4]))
```

$$\left(\sqrt{5}i\right)^{\alpha}\left(\frac{1}{2}-\frac{2\sqrt{5}i}{5}\right)+\left(-\sqrt{5}i\right)^{\alpha}\left(\frac{1}{2}+\frac{2\sqrt{5}i}{5}\right)$$

and now we can use the following code:

~~~
```{glue:math} sym_eq
:label: eq-sym
```
~~~

to insert the equation here:

$$\left(\sqrt{5}i\right)^{\alpha}\left(\frac{1}{2}-\frac{2\sqrt{5}i}{5}\right)+\left(-\sqrt{5}i\right)^{\alpha}\left(\frac{1}{2}+\frac{2\sqrt{5}i}{5}\right) \quad ()$$

Note

`glue:math` only works with glued variables that contain a `text/latex` output.

Advanced `glue` use-cases

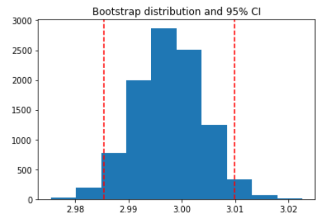
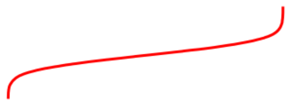
Here are a few more specific and advanced uses of the `glue` submodule.

Pasting into tables

In addition to pasting blocks of outputs, or in-line with text, you can also paste directly into tables. This allows you to compose complex collections of structured data using outputs that were generated in other cells or other notebooks. For example, the following Markdown table:

```
| name                             |       plot                     | mean                      | ci
|
|:-------------------------------:|:-----------------------------:|---------------------------|-----
----------------------------------------------|
| histogram and raw text           | {glue:}`boot_fig`              | {glue:}`boot_mean`          |
{glue:}`boot_clo`-{glue:}`boot_chi`                       |
| sorted means and formatted text | {glue:}`sorted_means_fig`      | {glue:text}`boot_mean:.3f` |
{glue:text}`boot_clo:.3f`-{glue:text}`boot_chi:.3f` |
```

Results in:

name	plot	mean	ci
histogram and raw text		2.99758724978736	2.985312582852057-3.0098125309029817
sorted means and formatted text		**2.998**	**2.985-3.010**

Execute and cache your pages

Jupyter Book can automatically run and cache any notebook pages. Notebooks can either be run each time the documentation is built, or cached locally so that notebooks will only be re-run when the code cells in a notebook have changed.

Caching behaviour is controlled with the `execute:` section in your `_config.yml` file. See the sections below for each configuration option and its effect.

> **Tip**
>
> If you'd like to execute code that is in your Markdown files, you can use the `{code-cell}` directive in MyST Markdown. See Notebooks written entirely in Markdown for more information.

Trigger notebook execution

By default, Jupyter Book will execute any content files that have a notebook structure and that are missing at least one output. This is equivalent to the following configuration in _config.yml`:

```
execute:
  execute_notebooks: auto
```

This will only execute notebooks that are missing at least one output. If the notebook has *all* of its outputs populated, then it will not be executed.

To force the execution of all notebooks, regardless of their outputs, change the above configuration value to:

```
execute_notebooks: force
```

To cache execution outputs with jupyter-cache, change the above configuration value to:

```
execute:
  execute_notebooks: cache
```

See Caching the notebook execution for more information.

To turn off notebook execution, change the above configuration value to:

```yaml
execute:
  execute_notebooks: 'off'
```

Exclude files from execution

To exclude certain file patterns from execution, use the following configuration:

```yaml
execute:
  exclude_patterns:
    - 'pattern1'
    - 'pattern2'
    - '*pattern3withwildcard'
```

Any file that matches one of the items in `exclude_patterns` will not be executed.

Tip

To auto-exclude all files outside of your table of contents, see Disable building files that aren't specified in the TOC

Caching the notebook execution

You may also **cache the results of executing a notebook page** using jupyter-cache. In this case, when a page is executed, its outputs will be stored in a local database. This allows you to be sure that the outputs in your documentation are up-to-date, while saving time avoiding unnecessary re-execution. It also allows you to store your `.ipynb` files in your `git` repository *without their outputs*, but still leverage a cache to save time when building your site.

When you re-build your site, the following will happen:

- Notebooks that have not seen changes to their **code cells** since the last build will not be re-executed. Instead, their outputs will be pulled from the cache and inserted into your site.
- Notebooks that **have had any change to their code cells** will be re-executed and the cache will be updated with the new outputs.

To enable caching of notebook outputs, use the following configuration:

```yaml
execute:
  execute_notebooks: cache
```

By default, the cache will be placed in the parent of your build folder. Generally, this is in `_build/.jupyter_cache`.

You may also specify a path to the location of a jupyter cache you'd like to use:

```yaml
execute:
  cache: path/to/mycache
```

The path should point to an **empty folder**, or a folder where a **jupyter cache already exists**.

Execution configuration

You can control notebook execution and how output content is handled at a project level using your `_config.yml` but, in some cases, also at a notebook and code cell level. Below we explore a number of ways to achieve this.

See also

Add metadata to notebooks and Formatting code outputs.

The execution working directory

Important

The default behaviour of `cache` is now to run in the local directory. This is a change from `v0.7`.

By default, the command working directory (cwd) in which a notebook runs will be the directory in which it is located (for both `auto` and `cache`). This means that notebooks requiring access to assets in relative paths will work.

Alternatively, if you wish for your notebooks to isolate your notebook execution in a temporary folder, you can use the following `_config.yml` setting:

```
execute:
  run_in_temp: true
```

Setting execution timeout

Execution timeout defines the maximum time (in seconds) each notebook cell is allowed to run for. If the execution takes longer an exception will be raised. The default is 30 seconds, so in cases of long-running cells you may want to specify a higher value. The timeout option can also be set to -1, to remove any restriction on execution time.

You can set the timeout for all notebook executions in your `_config.yml`:

```
execute:
  timeout: 100
```

This global value can also be overridden per notebook by adding this to your notebook metadata:

```
{
 "metadata": {
  "execution": {
      "timeout": 30
  }
}
```

Dealing with code that raises errors

In some cases, you may want to intentionally show code that doesn't work (e.g., to show the error message).

You can allow errors for all notebooks in your `_config.yml`:

```
execute:
  allow_errors: true
```

This global value can also be overridden per notebook by adding this to your notebook metadata:

```
{
 "metadata": {
  "execution": {
      "allow_errors": false
  }
}
```

Lastly, you can allow errors at a cell level, by adding a `raises-exception` tag to your code cell. This can be done via a Jupyter interface, or via the `{code-cell}` directive like so:

~~~
```{code-cell}
---
tags: [raises-exception]
---
print(thisvariabledoesntexist)
```
~~~

Which produces:

```
print(thisvariabledoesntexist)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-125c53ec1b82> in <module>
----> 1 print(thisvariabledoesntexist)

NameError: name 'thisvariabledoesntexist' is not defined
```

Dealing with code that produces stderr

You may also wish to control how stderr outputs are dealt with.

Alternatively, you can configure how stdout is dealt with at a global configuration level, using the `nb_output_stderr` configuration value.

You can configure the default behaviour for all notebooks in your `_config.yml`:

```
execute:
  stderr_output: show
```

Where the value is one of:

- `"show"` (default): show all stderr (unless a `remove-stderr` tag is present)
- `"remove"`: remove all stderr
- `"remove-warn"`: remove all stderr, but log a warning if any found
- `"warn"`, `"error"` or `"severe"`: log the stderr at a certain level, if any found.

You can also remove stderr at a cell level, using the `remove-stderr` cell tag, like so:

~~~
```{code-cell} ipython3
:tags: [remove-stderr]

import sys
print("this is some stdout")
print("this is some stderr", file=sys.stderr)
```
~~~

which produces

```
import sys
print("this is some stdout")
print("this is some stderr", file=sys.stderr)
```

```
this is some stdout
```

Dealing with code that produces stdout

Similar to stderr, you can remove stdout at a cell level with the `remove-stdout` tag, by which

~~~
```{code-cell} ipython3
:tags: [remove-stdout]

import sys
print("this is some stdout")
print("this is some stderr", file=sys.stderr)
```
~~~

produces the following:

```
import sys
print("this is some stdout")
print("this is some stderr", file=sys.stderr)
```

```
this is some stderr
```

Execution statistics

As notebooks are executed, certain statistics are stored on the build environment by MyST-NB. The simplest way to access and visualise this data is using the `{nb-exec-table}` directive.

See also

The MyST-NB documentation, for creating your own directives to manipulate this data.

The simple directive

~~~
```{nb-exec-table}
```
~~~

produces:

Document	Modified	Method	Run Time (s)	Status
basics/create	2021-05-03 15:27	cache	2.19	✅
content/code-outputs	2021-05-03 15:27	cache	5.02	✅
content/execute	2021-05-03 15:27	cache	1.28	✅
content/layout	2021-05-03 15:27	cache	2.0	✅
content/math	2021-05-03 15:27	cache	1.15	✅
content/references	2021-05-03 15:27	cache	1.15	✅
file-types/jupytext	2021-05-03 15:27	cache	0.9	✅
file-types/myst-notebooks	2021-05-03 15:27	cache	1.15	✅
file-types/notebooks	2021-05-03 15:27	cache	5.03	✅
interactive/hiding	2021-05-03 15:27	cache	1.82	✅
interactive/interactive	2021-05-03 15:27	cache	2.57	✅
interactive/launchbuttons	2021-05-03 15:27	cache	1.76	✅
reference/cheatsheet	2021-05-03 15:27	cache	4.03	✅
start/overview	2021-05-03 15:27	cache	2.33	✅

Interactive data visualizations

Jupyter Notebook has support for many kinds of interactive outputs, including the ipywidgets ecosystem as well as many interactive visualization libraries. These are supported in Jupyter Book, with the right configuration. This page has a few common examples.

First off, we'll download a little bit of data and show its structure:

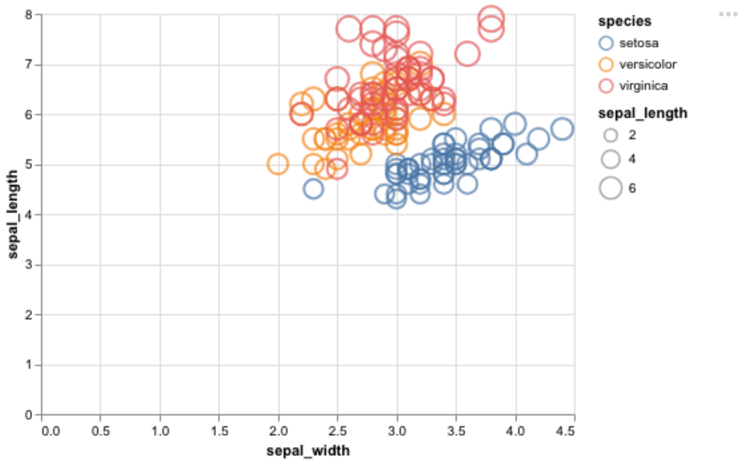
```python
import plotly.express as px
data = px.data.iris()
data.head()
```

	sepal_length	sepal_width	petal_length	petal_width	species	species_id
0	5.1	3.5	1.4	0.2	setosa	1
1	4.9	3.0	1.4	0.2	setosa	1
2	4.7	3.2	1.3	0.2	setosa	1
3	4.6	3.1	1.5	0.2	setosa	1
4	5.0	3.6	1.4	0.2	setosa	1

Altair

Interactive outputs will work under the assumption that the outputs they produce have self-contained HTML that works without requiring any external dependencies to load. See the `Altair` installation instructions to get set up with Altair. Below is some example output.

```
import altair as alt
alt.Chart(data=data).mark_point().encode(
    x="sepal_width",
    y="sepal_length",
    color="species",
    size='sepal_length'
)
```



Plotly

Plotly is another interactive plotting library that provides a high-level API for visualization. See the Plotly JupyterLab documentation to get started with Plotly in the notebook.

Plotly uses renderers to output differe kinds of information when you display plot. Experiment with renderers to get the output you want.

Below is some example output.

Important

For these plots to show, it may be necessary to load `require.js`, in your `_config.yml`:

```
sphinx:
  config:
    html_js_files:
    - https://cdnjs.cloudflare.com/ajax/libs/require.js/2.3.4/require.min.js
```

```
import plotly.io as pio
import plotly.express as px
import plotly.offline as py

df = px.data.iris()
fig = px.scatter(df, x="sepal_width", y="sepal_length", color="species",
size="sepal_length")
fig
```

Bokeh

Bokeh provides several options for interactive visualizations, and is part of the PyViz ecosystem. See the Bokeh with Jupyter documentation to get started.

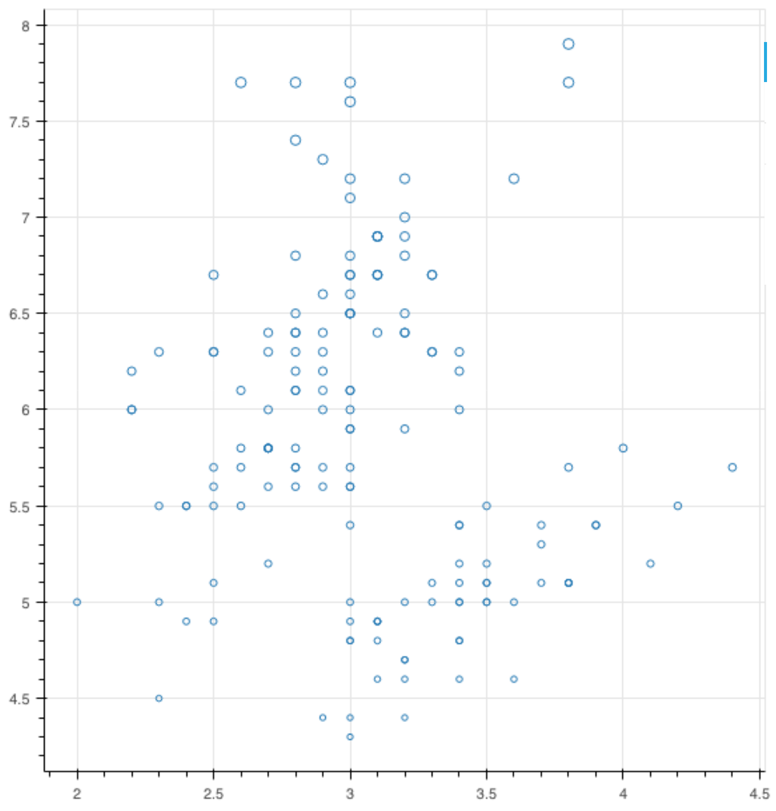
Below is some example output. First we'll initialized Bokeh with `output_notebook()`. This needs to be in a separate cell to give the JavaScript time to load.

```
from bokeh.plotting import figure, show, output_notebook
output_notebook()
```

```
BokehJS 2.3.0 successfully loaded.
```

Now we'll make our plot.

```
p = figure()
p.circle(data["sepal_width"], data["sepal_length"], fill_color=data["species"],
size=data["sepal_length"])
show(p)
```



ipywidgets

You may also run code for Jupyter Widgets in your document, and the interactive HTML outputs will embed themselves in your side. See [the ipywidgets documentation](#) for how to get set up in your own environment.

> **Widgets often need a kernel**
>
> Note that `ipywidgets` tend to behave differently from other interactive visualization libraries. They interact both with Javascript, and with Python. Some functionality in `ipywidgets` may not work in default Jupyter Book pages (because no Python kernel is running). You may be able to get around this with [tools for remote kernels, like thebe](#).

Here are some simple widget elements rendered below.

```
import ipywidgets as widgets
widgets.IntSlider(
    value=7,
    min=0,
    max=10,
    step=1,
    description='Test:',
    disabled=False,
    continuous_update=False,
    orientation='horizontal',
    readout=True,
    readout_format='d'
)
```

```
tab_contents = ['P0', 'P1', 'P2', 'P3', 'P4']
children = [widgets.Text(description=name) for name in tab_contents]
tab = widgets.Tab()
tab.children = children
for ii in range(len(children)):
    tab.set_title(ii, f"tab_{ii}")
tab
```

You can find [a list of existing Jupyter Widgets](#) in the jupyter-widgets documentation.

Build a standalone page

Sometimes you'd like to build a single page of content rather than an entire book. For example, if you'd like to generate a web-friendly HTML page from a Jupyter notebook for a report or publication.

You can generate a standalone HTML file for a single page of the Jupyter Book using the same command:

```
jupyter-book build path/to/mypage.ipynb
```

This will execute your content and output the proper HTML in a `_build/_page/html/<mypage>` folder. If the file is in a subdirectory relative to the `_build` folder, the HTML will be in a `_build/_page/html/<subdirectory-mypage>` folder.

Your page will be called `mypage.html`. This will work for any content source file that is supported by Jupyter Book.

> **Note**
>
> Users should note that building **single pages** in the context of a larger project can trigger warnings and incomplete links. For example, building `docs/start/overview.md` will issue a number of `unknown document`, `term not in glossary`, and `undefined links` warnings.

Host your book on the internet

A common way to publish your book content is to create an HTML version of your book (also called a *static website*) and host it on the web somewhere. There are a few ways to do this, and this section covers the methods available to you.

GitHub Pages and Actions

Once your content is on GitHub, you can easily host it as a GitHub Pages website. This is a service where GitHub hosts your static files as if they were a standalone website.

There are three ways you can quickly host your book with GitHub Pages:

- Copy/paste your book's HTML to a `docs/` folder, or a `gh-pages` branch of your repository.
- Use the `ghp-import` tool to automatically push your built documentation to a `gh-pages` branch.
- Use a GitHub Action to automatically build your book and update your website when you change the content.

Well cover each option below.

Manually put your book's contents online

In this case, you manually build your book's files, and then push them to a GitHub repository in order to be hosted as a website. There are two ways to do so

> **Make sure these steps are done first**
>
> Before you do any of the following, make sure that these two steps are completed:
>
> 1. Build HTML for your book (see Build your book). There should be a collection of HTML files in your book's `_build/html` folder.
> 2. Configure your GitHub repository to serve a website via GitHub Pages at the location of your choice (either a branch or the `docs/` folder). See the GitHub Pages documentation for more information.

(Option 1) Copy and paste your book's `_build` contents into a new folder

The simplest way to host your book online is to simply copy everything that is inside `_build` and put it in a location where GitHub Pages knows to look. There are two places we recommend:

In a separate branch
: You can configure GitHub Pages to build any books that are in a branch that you specify. By default, this is `gh-pages`.

In a `docs/` folder of your main branch
: If you'd like to keep your built book alongside your book's source files, you may paste them into a `docs/` folder.

> ⚠ **Warning**
>
> Note that copying all of your book's build files into the same branch as your source files will cause your repository to become very large over time, especially if you have many images in your book.

In either case, follow these steps:

1. Copy the contents of `_build/html` directory into `docs` (or your other branch).
2. Add a file called `.nojekyll` alongside your book's contents. This tells GitHub Pages to treat your files as a "static HTML website".
3. Push your changes to GitHub, and configure it to start hosting your documentation.

(Option 2) Automatically push your build files with `ghp-import`

The easiest way to use GitHub Pages with your built HTML is to use the `ghp-import` package. `ghp-import` is a lightweight Python package that makes it easy to push HTML content to a GitHub repository.

`ghp-import` works by copying *all* of the contents of your built book (i.e., the `_build/html` folder) to a branch of your repository called `gh-pages`, and pushes it to GitHub. The `gh-pages` branch will be created and populated automatically for you by `ghp-import`. To use `ghp-import` to host your book online with GitHub Pages follow the steps below:

1. Install `ghp-import`

   ```
   pip install ghp-import
   ```

2. From the `master` branch of your book's root directory (which should contain the `_build/html` folder) call `ghp-import` and point it to your HTML files, like so:

   ```
   ghp-import -n -p -f _build/html
   ```

> ⚠ **Warning**
>
> Make sure that you included the `-n`. This adds a file called `.nojekyll` to the output of your book, which tells GitHub *not* to build your book with Jekyll.

Typically after a few minutes your site should be viewable online at a url such as: `https://<user>.github.io/<myonlinebook>/`. If not, check your repository settings under **Options** -> **GitHub Pages** to ensure that the `gh-pages` branch is configured as the build source for GitHub Pages and/or to find the url address GitHub is building for you.

To update your online book, make changes to your book's content on the `main` branch of your repository, re-build your book with `jupyter-book build mybookname/` and then use `ghp-import -n -p -f mylocalbook/_build/html` as before to push the newly built HTML to the `gh-pages` branch.

> ⚠ **Warning**
>
> Note this warning from the `ghp-import` GitHub repository:
>
> *"...`ghp-import` will DESTROY your gh-pages branch... and assumes that the `gh-pages` branch is 100% derivative. You should never edit files in your `gh-pages` branch by hand if you're using this script..."*

Automatically host your book with GitHub Actions

GitHub Actions is a tool that allows you to automate things on GitHub. It is used for a variety of things, such as testing, publishing packages and continuous integration.

Note that if you're not hosting your book on GitHub, or if you'd like another, user-friendly service to build it automatically, see the guide to publishing your book on Netlify.

Note

You should be familiar with GitHub Actions before using them to automatically host your Jupyter Books. See the GitHub Actions documentation for more information.

To build your book with GitHub Actions, you'll need to create an action that does the following things:

- Activates when a *push* event happens on `master` (or whichever) branch has your latest book content.
- Installs Jupyter Book and any dependencies needed to build your book.
- Builds your book's HTML.
- Uses a `gh-pages` action to upload that HTML to your `gh-pages` branch.

For reference, here is a sample repository that builds a book with GitHub Actions.

Note

Ensure that Jupyter Book's version in your `requirements.txt` file is at least `0.7.0`.

Tip

You can use the Jupyter Book cookiecutter to quickly create a book template that already includes the GitHub Actions workflow file needed to automatically deploy your book to GitHub Pages:

```
jupyter-book create --cookiecutter mybookpath/
```

For more help, see the Jupyter Book cookiecutter GitHub repository, or run:

```
jupyter-book create --help
```

Here is a simple YAML configuration for a Github Action that will publish your book to a `gh-pages` branch.

```yaml
name: deploy-book

# Only run this when the master branch changes
on:
  push:
    branches:
    - master
    # If your git repository has the Jupyter Book within some-subfolder next to
    # unrelated files, you can make this run only if a file within that specific
    # folder has been modified.
    #
    # paths:
    # - some-subfolder/**

# This job installs dependencies, build the book, and pushes it to `gh-pages`
jobs:
  deploy-book:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v2

    # Install dependencies
    - name: Set up Python 3.7
      uses: actions/setup-python@v1
      with:
        python-version: 3.7

    - name: Install dependencies
      run: |
        pip install -r requirements.txt

    # Build the book
    - name: Build the book
      run: |
        jupyter-book build .

    # Push the book's HTML to github-pages
    - name: GitHub Pages action
      uses: peaceiris/actions-gh-pages@v3.6.1
      with:
        github_token: ${{ secrets.GITHUB_TOKEN }}
        publish_dir: ./_build/html
```

If you want to deploy your site to GitHub Pages at a User and Organization repository (`<username>.github.io`), check another example workflow and available options at the README of peaceiris/actions-gh-pages.

Publish with Netlify

Netlify is a continuous deployment service that can **automatically build an updated copy of your Jupyter Book** as you push new content. It can be used across git clients including GitHub, GitLab, and Bitbucket.

Note that these instructions assume you're keeping your source files under version control, rather than the built Jupyter Book HTML. If you're pushing your HTML to GitHub, you'll want to host your book on GitHub Pages instead.

Although Netlify has both free and paid tiers, the build process is the same across both Importantly, the free tier only allows for 100GB of bandwidth usage per month across all of your Netlify built projects.

In order to use Netlify, you'll need to create an account. Here, we'll walk through connecting your Jupyter Book to Netlify's continous deployment services using their UI. You can also check out their documentation on continuous deployment.

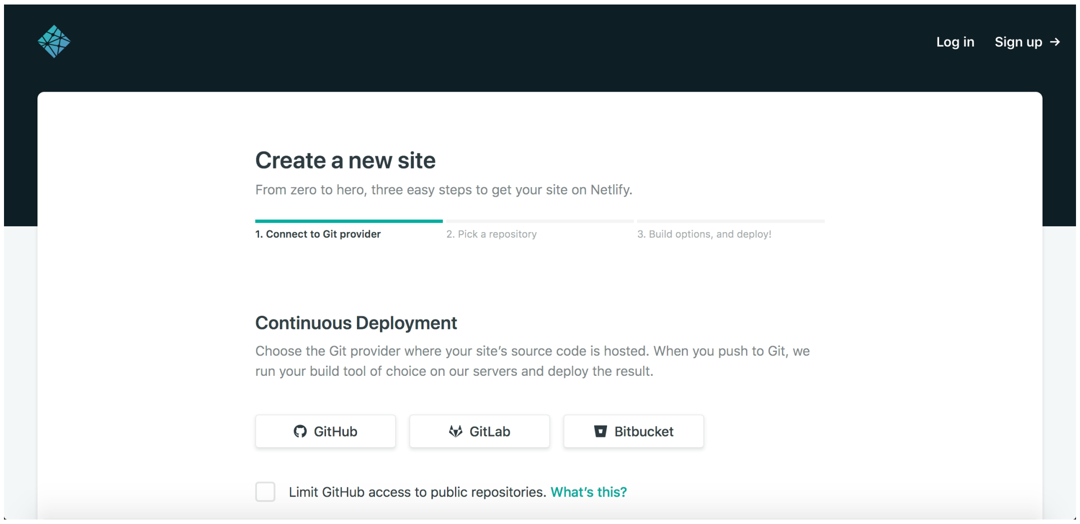
If your Jupyter Book will be used by a large audience, or if you're creating many Jupyter Books, you might want t consider registering for a paid accoun

> ⚠ **Warning**
>
> The default Netlify Python environment is Python 2.7. You should update the Python environment by including a `runtime.txt` file in your repository, as detailed in the Netlify documentation.
>
> For a full list of available environments, please see the Netlify build image details.

Step 1: Connect your GitHub repo to Netlify

After you've created a Netlify account, you'll need to log in. The home page will be a dashboard of all sites you're currently building with Netlify. We can then import a new site by clicking the "New Site from Git" button in the upper right.

This should launch the site builder:



Create a new site

From zero to hero, three easy steps to get your site on Netlify.

1. Connect to Git provider 2. Pick a repository 3. Build options, and deploy!

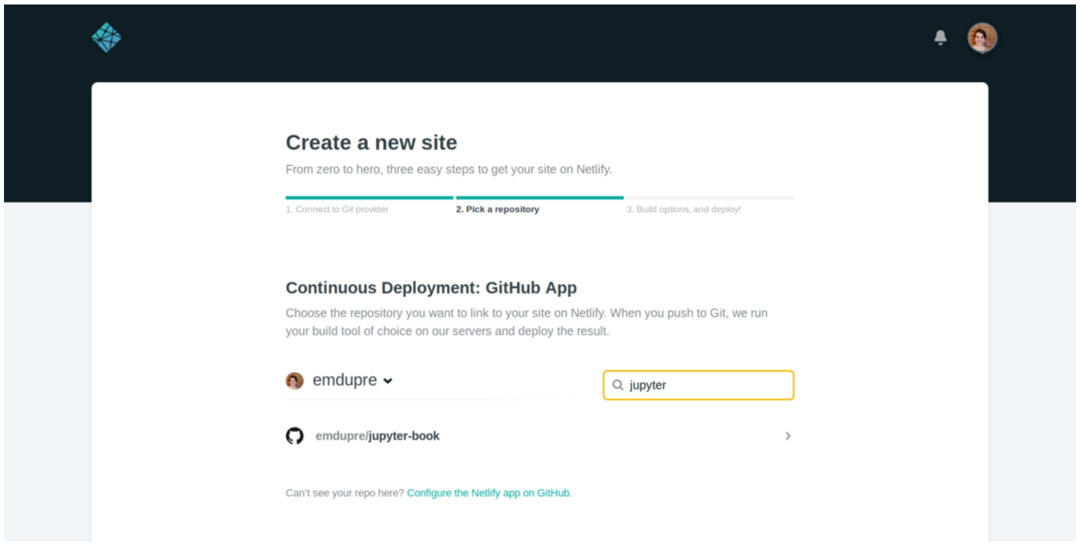
Continuous Deployment

Choose the Git provider where your site's source code is hosted. When you push to Git, we run your build tool of choice on our servers and deploy the result.

Limit GitHub access to public repositories. What's this?

Here, you can select the git client where your Jupyter Book is hosted. For the purposes of this tutorial, we'll assume that your book is hosted on GitHub.

When you select the "GitHub" option, you'll be asked to grant permission for Netlify to access your GitHub account. Authorizing access will take you to the next step of the build process, where you can select your Jupyter Book repository.



Create a new site

From zero to hero, three easy steps to get your site on Netlify.

1. Connect to Git provider 2. Pick a repository 3. Build options, and deploy!

Continuous Deployment: GitHub App

Choose the repository you want to link to your site on Netlify. When you push to Git, we run your build tool of choice on our servers and deploy the result.

emdupre/jupyter-book

Can't see your repo here? Configure the Netlify app on GitHub.

Step 2: Add the command to install and build your book

Once you've selected the correct repository, you'll need to supply build instructions. This is a command that Netlify runs before hosting your site. We'll use it to do the following:

- Install Jupyter Book and your book's dependencies
- Build your book's HTML

If your book content is not in the root o
your repository, make sure you point to
it in the `jupyter-book build` command

Assuming that your book's dependencies are in a `requirements.txt` file, **put the following command in the *Build command* section**:

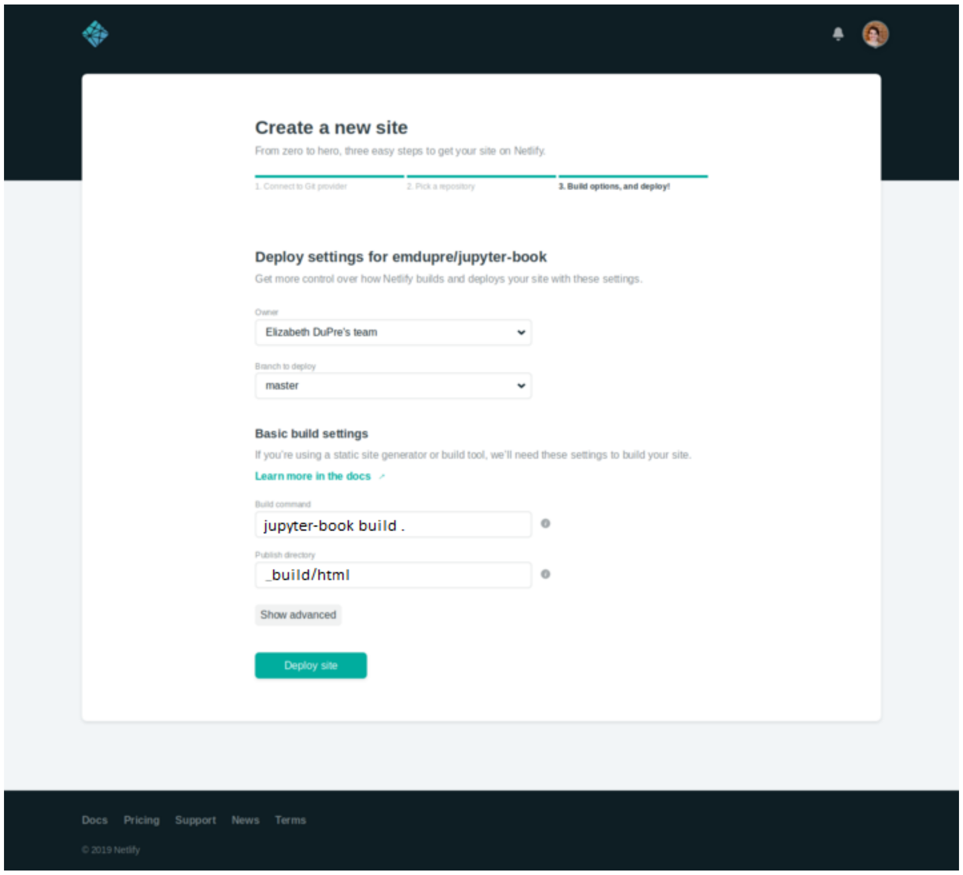
```
pip install -r requirements.txt && jupyter-book build .
```

Note

Ensure that Jupyter Book's version in your `requirements.txt` file is at least `0.7.0`.

Finally, the *Publish directory* should be `_build/html`.

You'll also need to select the appropriate branch to build your repository from. In this example, we'll use the `master` branch.



Create a new site

From zero to hero, three easy steps to get your site on Netlify.

1. Connect to Git provider
2. Pick a repository
3. Build options, and deploy!

Deploy settings for emdupre/jupyter-book

Get more control over how Netlify builds and deploys your site with these settings.

Owner	Elizabeth DuPre's team
Branch to deploy	master

Basic build settings

If you're using a static site generator or build tool, we'll need these settings to build your site.

Learn more in the docs

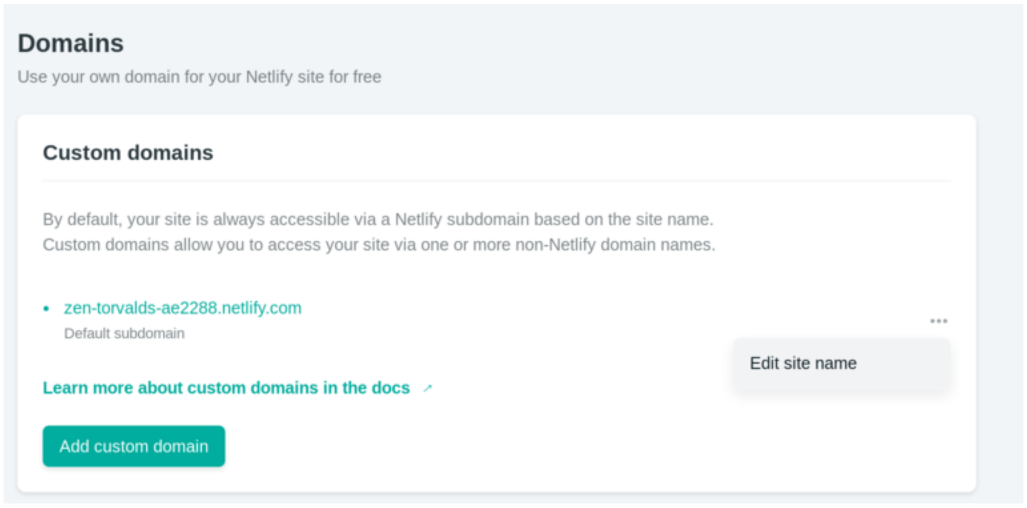
Build command	jupyter-book build .
Publish directory	_build/html

You can then select *Deploy site* and wait for the site to build. You'll be redirected to the site dashboard during the build process.

Step 3: Updating your domain name

If your site has successfully built, it will be assigned a random domain name. In order to have a more memorable address, you can update your site's name.

From the site dashboard, select *Domain settings*. This will take you to a sub-menu, where you can choose to update your site name.



Domains

Use your own domain for your Netlify site for free

Custom domains

By default, your site is always accessible via a Netlify subdomain based on the site name. Custom domains allow you to access your site via one or more non-Netlify domain names.

- zen-torvalds-ae2288.netlify.com
  Default subdomain

You can enter a memorable, unique name here to describe your Jupyter Book! Note that it will be prepended to `.netlify.com` so, `MY-BOOK` will become `MY-BOOK.netlify.com`.

You can also use a custom domain (i.e., one that you have purchased through a DNS registrar). See the Netlify documentation on custom domains for more details on this process.

Connect your book to a code repository

There are many ways that you can connect your book's content back to the source files in a public repository. Below we cover a few options.

Add source repository buttons

There is a collection of buttons that you can use to link back to your source repository. This lets users browse the repository or take actions like suggesting an edit or opening an issue. In each case, they require the following configuration to be set:

```yaml
repository:
  url: https://github.com/{your-book-url}
```

Add a link to your repository

To add a link to your repository, add the following configuration:

```yaml
repository:
  url: https://github.com/{your-book-url}
html:
  use_repository_button: true
```

Add a button to open issues

To add a button to open an issue about the current page, use the following configuration:

```yaml
repository:
  url: https://github.com/{your-book-url}
html:
  use_issues_button: true
```

Add a button to suggest edits

You can add a button to each page that will allow users to edit the page text directly and submit a pull request to update the documentation. To include this button, use the following configuration:

```yaml
repository:
  url: https://github.com/{your-book-url}
  path_to_book: path/to/your/book  # An optional path to your book, defaults to repo root
  branch: yourbranch  # An optional branch, defaults to `master`
html:
  use_edit_page_button: true
```

Connect with interactive environments

Because Jupyter Books are built with Jupyter notebooks, you can allow users to launch live Jupyter sessions in the cloud directly from your book. This lets readers quickly interact with your content in a traditional coding interface. There now exist numerous online notebook services - a good comparison is provided in this article - and the following sections describes the available integrations provided by Jupyter Book.

In each case, you'll need to tell Jupyter Book where your book content lives online. To do so, use this configuration in `_config.yml`:

```
# Information about where the book exists on the web
repository:
  url                       : https://github.com/yourusername/yourbookrepo  # Online location of
your book
  path_to_book              : path/to/book  # Optional path to your book, relative to the repository
root
  branch                    : master  # Which branch of the repository should be used when creating
links (optional)
```

A quick description of each option is below:

`url`: A GitHub repository that includes your source files used to build the Jupyter Book. These files can either be in the root of the repository, or in a sub-folder (in which case you should use `path_to_book`).

`path_to_book`: A path, relative to your repository's root, to your book's source files. Use this if your book is in a sub-folder of your repository (e.g. `docs/` or `book/`).

`branch`: The branch where your book's **source files** are stored (not your book's *build files*, which generally exist in `gh-pages/` branch).

Control the notebook interface that opens

Binder and JupyterHub sessions can be opened using either the "classic" Jupyter Notebook or the new JupyterLab interface backend (see jupyter.org for more details). This is configured using:

```
launch_buttons:
  notebook_interface: "jupyterlab"  # or "classic"
```

One thing to take into account when choosing the interface is that notebooks written in the MyST Markdown text-based format will not be opened as notebooks out-of-the-box. If you wish for these files to be opened as notebooks then firstly you must ensure that `jupytext>=0.16` is installed in the Binder/JupyterHub environment for your book (no support for this feature exists in Google Colab). You then have two options:

- Use the "classic" interface, which will then immediately open these files as notebooks.
- The "jupyterlab" interface (at the time of writing) has not yet implemented this behaviour, and so you will need to instruct readers to right-click the Markdown file and click "Open in notebook editor".

Add a Launch on Binder button

BinderHub can be used to build the environment needed to run a repository, and provides a link that lets others interact with that repository. If your Jupyter Book is hosted online on GitHub, you can automatically insert buttons that link to the Jupyter Notebook running in a BinderHub. When a user clicks the button, they'll be taken to a live version of the page. If your code doesn't require a significant amount of CPU or RAM, you can use the free, public BinderHub running at https://mybinder.org.

To automatically include Binder link buttons in each page of your Jupyter Book, use the following configuration in `_config.yml`:

```
launch_buttons:
  binderhub_url: "https://mybinder.org"  # The URL for your BinderHub (e.g., https://mybinder.org)
```

By adding this configuration, along with the repository url configuration above, Jupyter Book will insert Binder links to any pages that were built from notebook content.

Add a Launch on JupyterHub button

JupyterHub lets you host an online service that gives users their own Jupyter servers with an environment that you specify for them. It allows you to give users access to resources and hardware that you provision in the cloud, and allows you to authenticate users in order to control who has access to your hardware.

Similar to Binder link buttons, you can also automatically include interact links that send your readers to a JupyterHub that is running a live, interactive version of your page. This is accomplished using the nbgitpuller server extension.

You can configure the location of the JupyterHub (which you may set up on your own using a guide such as zero to jupyterhub for kubernetes or the littlest jupyterhub) with the following configuration:

```
launch_buttons:
  jupyterhub_url: "your-hub-url"  # The URL for your JupyterHub. (e.g.,
https://datahub.berkeley.edu)
```

On your JupyterHub server, you need two dependencies installed:

1. To clone the notebook with the launch link, the server needs `nbgitpuller`.
2. To open myst-markdown as notebooks, the server needs `jupytext>=0.16`.

You can add the following line to your `DockerFile`:

```
RUN pip install jupytext nbgitpuller
```

Add a Launch on Google Colab button

If your Jupyter Book is hosted online on GitHub, you can automatically insert buttons that link to the Jupyter Notebook running on Google Colab. When a user clicks the button, they'll be taken to a live version of the page.

Similar to Binder link buttons, you can automatically include Google Colab link buttons with the following configuration in `_config.yml`:

```
launch_buttons:
  colab_url: "https://colab.research.google.com"
```

Note

Google Colab links will only work for pages that have the `.ipynb` extension.

Live interactive pages with Thebe

This section describes how to bring interactivity to your book. This lets users run code and see outputs *without leaving the page*. Interactivity is provided by a kernel running on the public **MyBinder** service.

Warning

This is an experimental feature, and may change in the future or work unexpectedly.

To make your content interactive without requiring readers to leave the current page, you can use a project called Thebe. This provides you with a `Live Code` button that, when clicked, will convert each code cell into an **interactive** cell that can be edited. It also adds a "run" button to each cell, and connects to a Binder kernel running in the cloud.

To add a Thebe button to your Jupyter Book pages, use the following configuration:

```
launch_buttons:
  thebe                  : true
```

In addition, you can configure the Binder settings that are used to provide a kernel for Thebe to run the code. These use the same configuration fields as the BinderHub interact buttons described above.

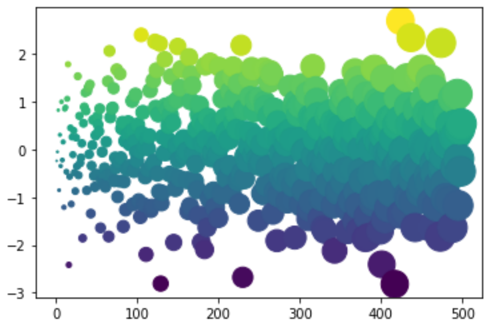
For an example, click the **Thebe** button above on this page, and run the code below.

```
import numpy as np
import matplotlib.pyplot as plt
plt.ion()

x = np.arange(500)
y = np.random.randn(500)

fig, ax = plt.subplots()
ax.scatter(x, y, c=y, s=x)
```

```
<matplotlib.collections.PathCollection at 0x7fe75fce1b50>
```



Running cells in Thebe when it is initialized

Sometimes you'd like to initialize the kernel that Thebe uses by running some code ahead of time. This might be code that you then hide from the user in order to narrow the focus of what they interact with. This is possible by using Jupyter Notebook tags.

Adding the tag `thebe-init` to any code cell will cause Thebe to run this cell after it has received a kernel. Any subsequent Thebe cells will have access to the same environment (e.g. any module imports made in the initialization cell).

You can then pair this with something like `hide-input` in order to run initialization code that your user doesn't immediately see. For example, below we'll initialize a variable in a hidden cell, and then tell another cell to print the output of that variable.

```
# The variable for this is defined in the cell above!
print(my_hidden_variable)
```

```
wow, it worked!
```

Commenting and annotating

Danger

This is an experimental feature that may change quickly! If you're interested in trying this out, we recommend using the `master` branch version of Jupyter Book.

You can add functionality for web commenting and annotating through a number of web services and tools. These allow readers of your book to share conversations, ideas and questions in a centralized location. It is useful for classes, communities of practice, asking for user feedback, etc.

Currently, these commenting engines are supported:

- Hypothesis provides a web overlay that allows you to annotate and comment collaboratively.
- utterances is a web commenting system that uses GitHub Issues to store and manage comments.

Note

Jupyter Book uses the `sphinx-comments` Sphinx extension for this functionality.

Documentation for supported services

For examples of each service, as well as instructions on how to activate it, click on the links for each service in the left navigation bar or below.

Hypothesis

Hypothesis is a centralized web service that allows you to comment and annotate arbitrary web pages across the web. It allows your readers to log in and comment on your book.

> **Note**
>
> Hypothesis is activated on this page. You can see the web overlay by clicking on the `<` button in the upper-right corner of this page.

Activate `Hypothesis`

You can activate `Hypothesis` by adding the following to your `_conf.yml` file:

```yaml
html:
  comments:
    hypothesis: true
```

This will add a Hypothesis overlay to your documentation. This extension simply activates the Hypothesis JavaScript bundle on your Sphinx site.

When you build your documentation, you will see the Hypothesis overlay to the right of your screen.

Utterances

Utterances is a commenting engine built on top of GitHub Issues. It embeds a comment box in your page that users (with a GitHub account) can use to ask questions. These become comments in a GitHub issue in a repository of your choice.

> **Note**
>
> Utterances is activated on this page. You can see the comment box at the bottom of this page's content. Click the "log in" button and you'll be able to post comments!

Activate `utterances`

You can activate `utterances` by adding the following to your `_conf.yml` file:

```yaml
html:
  comments:
    utterances:
      repo: "github-org/github-repo"
```

Note that the `utterances` UI will not show up when you are previewing your book locally, it must be hosted somewhere on the web to function.

Configure `utterances`

You can pass optional extra configuration for utterances. You may do so by providing `key:val` pairs alongside `repo:` in the configuration. See the utterances documentation for your options.

When you build your documentation, pages will now have a comment box at the bottom. If readers log in via GitHub they will be able to post comments that will then map onto issues in your GitHub repository.