

sphinx-lesson: structured lessons with Sphinx

→ See also

See a real demo lesson at <https://coderefinery.github.io/github-without-command-line/>.

sphinx-lesson is a set of Sphinx extensions and themes for creating interactive, hands-on lessons. It was originally made to replace the CodeRefinery jekyll themes, but is designed to be used by others.

The broad idea is that making teaching materials should be quite similar to good software documentation. Sphinx-lesson adds a model and a few nice extras to bring them closer together.

As the name says, it is based on the [Sphinx documentation generator](#). It is also inspired by and based on [jupyter-book](#), but both is jupyter-book and isn't. It is because jupyter-book is based on Sphinx and modular, we reuse all of those same Sphinx extensions which jupyter-book has made. It *isn't* jupyter-book because we configure Sphinx directly, instead of wrapping it through jupyter-book configuration and building. Thus, we get full control and high compatibility.

Features:

- Separate content and presentation: easy to adjust theme or control the parts independently.
- Based on jupyter-book, cross-compatible.
- Built with Sphinx, providing a structured, controlled output.
- Distributed as Python pip packages
- Markdown and ReStructured equally supported as input formats (yes, including all directives).
- Jupyter notebooks as an input format. Can execute code (in jupyter and other formats, too)
- Transparent transformation of jekyll-style markdown styles into CommonMark with directives (mainly of use for old CodeRefinery lessons)
- Also renders with sphinx-book-theme (theme of jupyterbook) ([preview](#)) and other Sphinx themes.

This is in heavy internal use, and about ready for other users who are interested.

- If you know Sphinx, it helps some. If not, it's easy to copy
- Markdown or ReStructured text
- Hosting is usually by github-pages

Getting started

From a template repository

You can get started by making a Sphinx project and configuring the extension. We recommend you use the sphinx-lesson-template repository (<https://github.com/coderefinery/sphinx-lesson-template>).

This template repository is updated with new copies of base files as the lesson develops - you might want to check back for them, later.

Convert an existing jekyll lesson

See [Converting an old lesson](#). This hasn't been used in years so may be out of date.

From scratch

See the next page, [Installation](#), for raw Python packages to install and how to configure a arbitrary Sphinx project.

Github Pages initial commit

The included Github Actions file will automatically push to Github Pages, but due to some quirk/bugs in gh-pages *the very first non-human gh-pages push won't enable Github Pages*. So, you have to do one push yourself (or go to settings and disable-enable gh-pages the first time).

You can make an empty commit to gh-pages this way, which will trigger the gh-pages deployment (and everything will be automatic after that):

```
git checkout -b gh-pages origin/gh-pages
git commit -m 'empty commit to trigger gh-pages' --allow-empty
git push
```

Demo lessons

This guide can't currently stand alone. It is probably good to look at and copy from existing lessons for some things:

- Python for Scientific Computing uses many of the features:
<https://aaltoscicomp.github.io/python-for-scicomp/> .
- Github without the command line is a complete lesson using the theme:
<https://coderefinery.github.io/github-without-command-line/> .

Installation

Sphinx Python package

This is distributed as a normal Sphinx extension, so it is easy to use. To use it, install `sphinx_lesson` via PyPI.

Then, enable the extension in your Sphinx `conf.py`. This will both define our special directives, and load the other required extensions (`myst_nb`). The `myst_nb` extension can be configured normally:

```
extensions = [  
    'sphinx_lesson',  
]
```

HTML theme

We are in theory compatible with any theme, but are most tested with the `sphinx_rtd_theme` (which you need to set yourself):

```
html_theme = 'sphinx_rtd_theme'
```

The Jupyter Book (Executable Books Project) Sphinx theme ([sphinx-book-theme](#)) has some very nice features and also deserves some consideration. Using it should be clear: `html_theme = "sphinx_book_theme"`. You can see a preview of it [as a branch on github-pages](#).

Under the hood

Adding `sphinx_lesson` as an extension adds these sub-extensions, and you could selectively enable only the parts you want:

- `sphinx_lesson.directives` - see [Directives](#).
- `sphinx_lesson.md_transforms` - see [Markdown transforms](#).
- `sphinx_lesson.exerciselist` - see [Exercise list](#).
- `sphinx_lesson.term_role_formatting` - makes glossary term references bold
- Enables the [myst_notebook extension](#), which also enables [myst_parser](#) (included as a dependencies)

- Enables the [sphinx-copybutton extension](#) (included as a dependency)
- Same for [sphinx-tabs](#)
- Same for [sphinx-togglebutton](#)

Quickstart: Contributing to a lesson

If you are at this page, you might want to quickly contribute to some existing material using the `sphinx-lesson` format. Luckily, this is fairly easy:

- Get the source
- Edit the material in the `content/` directory
- (optional) Set up the Python environment and preview
- Send your contribution

In summary, each lesson is like a Python project, with the lesson content as its documentation in the `content/` directory (and no Python code). Everything is fairly standard: it uses the Sphinx documentation system, which is a popular, extendable tool. We have only minor extensions to make it suitable to lessons.

Instead of going through this process, you can also open an issue instead with your proposed change, and let someone else add it.

Get the lesson material

You need to consult with the lesson you would like to edit. If this is using the `git` version control system on Github, you could clone it like this:

```
$ git clone git://github.com/ORGANIZATION/LESSON.git
```

[CodeRefinery's git-intro lesson](#) explains more.

Edit the material

The material is in the `content/` directory. Depending on the lesson, it may be in MyST Markdown, ReStructured Text, or Jupyter notebooks.

ReStructured Text and MyST Markdown

You will probably copy existing examples, but you can also see [our quick guide](#). The main thing to note is that this is not unstructured Markdown, but there are particular (non-display) **directives** and **roles** to tag blocks and inline text. (In fact, “markdown” is a broad concept and everyone uses some different extensions of it).

- [sphinx-lesson directives for markup](#)
- [Markdown and ReST](#)
- [MyST reference](#)
- [ReStructured Text reference](#)

Do not worry about getting syntax right. Send your improvement, and editing is easy and you will learn something.

Jupyter notebooks

Jupyter notebooks are a common format for computational narratives, and can be natively used with Sphinx via [myst-nb](#). Note that you should use MyST Markdown directives and roles (see previous section) in the notebook to give structure to the material.

Again, do not worry about getting the syntax right. This is the least important part of things.

Build and test locally

Generic: The `requirements.txt` file includes all Python dependencies to build the lesson. The lesson can be built with `sphinx-build -M html content/ _build`, or `make html` if you have Make installed.

Or in more detail:

Create a virtual environment to install the requirements (a conda environment would work just as well):

```
$ python3 -m venv venv/  
$ source venv/bin/activate
```

Note

if `python3 -m venv venv/` does not work, try with `python -m venv venv/`

Then upgrade pip inside the virtual environment and install dependencies (it is recommended that conda base environment is deactivated):

```
$ pip install --upgrade pip  
$ pip install -r requirements.txt
```

You can build it using either of these commands:

```
$ sphinx-build -M html content/ _build
$ make html    # if you have make installed
```

And then view it with your web browser. Remove the `_build` directory to force a clean rebuild (or `make clean`).

Or you can use the **Sphinx autobuilder**, which will start a process that rebuilds it on every change, and starts a web server to view it. It will tell you how to access the server:

```
$ sphinx-autobuild content/ _build/
...
[I ...] Serving on http://127.0.0.1:8000
```

Sending your changes back

This depends on the project, but can be done using Github pull requests. [CodeRefinery's git-collaborative lesson](#) goes into details about pull requests.

Other things to keep in mind

- Make sure that you have rights to submit your change. In general, if you reuse anything else that already exists, explain this in your pull request.
- *Content and ideas are more important than markup.* Don't worry about doing something wrong, that is why we have review!
- Many different people use the lessons. Ask before doing things that make the lesson too specific to your use case.

Building the lesson

It is built the normal Sphinx ways. Using Sphinx directly, one would run:

```
$ sphinx-build -M html content/ _build
```

If you have `make` installed, you can:

```
$ make html
## or
$ make dirhtml
## full build
$ make clean html
```

However, there are different ways to set up a Sphinx project. CodeRefinery lessons puts the results in `_build/`.

Changelog

This isn't very up to date right now. There haven't been significant backwards-incompatible changes, so this hasn't been kept in earnest.

0.8.0

- First PyPI release

Markdown and ReST

Sites can be written in Markdown or ReStructured Text. Actually, in theory any format that has a Sphinx parser could be used, however you will be slightly limited without directive support.

The most important thing to note is that: To make a structured lesson, one needs to write in something a bit more structured than HTML.

Markdown

Markdown is the most common syntax for CodeRefinery lessons. It's not raw markdown but the MyST flavor, which has *much* more structured directives than plain markdown. These come straight from Sphinx and what we use to

[MyST syntax reference](#)

Note

What is Markdown? Markdown isn't a single language. Its native form is a simple syntax for HTML, and isn't very structured. There are many different flavors, some of which add extra syntax which gets it closer to enough, but for our purposes these are different enough that they should count as different languages (as similar as "markdown" and ReST). Since the Markdown creator says that [Markdown shouldn't evolve or be strictly defined](#), Markdown is essentially a dead syntax: we should always specify which living sub-syntax you are referring to.

sphinx-lesson uses the [MyST-parser](#) (Markedly Structured Text), which is both completely compatible with CommonMark, and also supports *all ReStructured Text directives*, unlike most other non-ReST Sphinx parsers. Thus, we finally have a way to write equivalent ReST and Markdown without any compromises (though other CommonMark parsers aren't expected to know Sphinx directives).

ReStructured Text

ReStructured Text has native support for roles and directives, which makes it a more structured language such as LaTeX, rather than HTML. It came before Sphinx, but Sphinx made it even more popular.

[ReST reference \(from Sphinx\)](#)

MD and ReST syntax

This is a brief comparison of basic syntax:

ReST syntax (Sphinx has a good [restructured text primer](#):

MyST markdown syntax:

```
*italic*
**bold**
`literal`
# Heading

[inline link](https://example.com)
[link to page](relative-page-path)

structured links:
{doc}`link to page <page-filename>`
{ref}`page anchor link <ref-name>`
{py:mod}`intersphinx link <multiprocessing>`

...
code block
...
```



```

*italic*
**bold**
``literal``

Heading
-----

`inline link <https://example.com>`__
`out-of-line link <example_>`__

.. _example: https://example.com

:doc:`page-filename`
:ref:`ref-name`
:py:mod:`multiprocessing`

:doc:`link to page <page-filename>`
:ref:`page anchor link <ref-name>`
:py:mod:`intersphinx link <multiprocessing>`

::

    code block that is standalone (two
    colons before it and indented)

Code block after paragraph::

    The paragraph will end with
    a single colon.

```

The most interesting difference is the use of single backquote for literals in Markdown, and double in ReST. This is because ReST uses single quotes for *roles* - notice how there is a dedicated syntax for inter-page links, references, and so on (it can be configured to “figure it out” if you want). This is very important for things like verifying referential integrity of all of our pages. But this is configurable with `default_role`: set to `any` to automatically detect documents/references/anything, or `literal` to automatically be the same as literal text.

Directives

A core part of any Sphinx site is the directives: this provides structured parsing for blocks of text. For example, we have an `exercise` directive, which formats a text block into an exercise callout. This is not just a CSS class, it can do anything during the build phase (but in practice we don’t do such complex things).

MyST directives

MyST-parser directives are done like this:

```
 :::{exercise}
:option: value

content
:::
```

ReST directives

ReST directives are done like this:

```
.. exercise:: Optional title, some default otherwise
   :option: value

   This is the body

   You can put arbitrary syntax here.
```

Roles

Roles are for inline text elements. A lot like directives, they can be as simple as styling or do arbitrary transformations in Python.

ReST roles

Like this:

```
:rolename: `interpreted text`
```

MyST roles

Like this:

```
{rolename}`interpreted text`
```

Table of Contents tree

Pages are not found by globbing a pattern: you can explicitly define a table of contents list. There must be one master toctree in the index document, but then they can be nested down. For the purposes of sphinx-lesson, we probably don't need such features

MyST:

```
```${toctree}

caption: Episodes
maxdepth: 1

basics
creating-using-web
creating-using-desktop
contributing
doi
websites
```
```

ReST:

```
.. toctree::
caption: Episodes
maxdepth: 1

basics
creating-using-web
creating-using-desktop
contributing
doi
websites
```

The pages are added by filename (no extension needed). The name by default comes from the document title, but you can override it if you want.

You can have multiple toctrees: check sphinx-test-lesson, we have one for the episodes, and one for extra material (like quick ref and instructor guide).

See also

- Read more about the [Sphinx toctree directive](#).

Directives

→ See also

In [Sphinx/Docutils](#), [directives have a different meaning](#). Directives in sphinx-lesson are actually the special case of the generic directive class called **admonitions**.

Directives are used to set off a certain block of text. They can be used as an aside or block (e.g. `exercise`, `instructor-note`). If the content of the box would be long (e.g. an entire episode is a `type-along`, or an entire section is an `exercise`), you could use the `type-along` directive to introduce the start of it but not put all content in that directive.

Sphinx and docutils calls these type of directives **admonitions** (and “directive” is a more general concept)

How to use

Example of `exercise`:

 **Exercise**

Some body text

 **Custom title**

Some body text

 **Exercise**

| Markdown | ReST: |
|---|---|
| <pre>```{exercise} Some body text ```</pre> | <pre>.. exercise:: Some body text</pre> |
| <pre>```{exercise} Custom title Some body text ```</pre> | <pre>.. exercise:: Custom title Some body text</pre> |
| <pre>```{exercise} ```</pre> | <pre>.. exercise:: </pre> |

You notice these directives can have optional a custom title. This is an addition from regular Sphinx admonitions, and is *not* usable in regular Sphinx admonition directives. Also, unlike regular Sphinx admonitions, the content in our directives is optional, if you want to use it as a simple section header.

The `solution` directive begins collapsed (via [sphinx-togglebutton](#)):

✓ **Solution**

This is a solution

Directives are implemented in the Python package `sphinx_lesson.directives` and can be used independently of the rest of `sphinx-lesson`.

List

Many directives are available.

The following directives are used for exercises/solutions/homework. They all render as green (“important” class:

- `demo`
- `exercise`
- `solution` (toggleable, default hidden)
- `type-along` (most of the lessons are hands-on, so this is a bit redundant. Use this when emphasizing a certain section is “follow along”, as opposed to watching or working alone.)
- `homework`

Other miscellaneous directives:

- `discussion`
- `instructor-note`
- `prerequisites`

The following are Sphinx default directives that may be especially useful to lessons. These do *not* accept an optional Title argument, the title is hard-coded.

- `see-also`
- `note`
- `important` (green)
- `warning` (yellow)
- `danger` (red)

The following are available, for compatibility with Carpentries styles:










- `callout`
- `challenge` (alias to `exercise`)
- `checklist`
- `keypoints` (bottom of lesson)
- `objectives` (top of lesson)
- `prereq` (use `prerequisites` instead)
- `solution` (begins collapsed)

- `testimonial`
- `output` (use code blocks instead)
- `questions` (top of lesson)

Gallery

This is a demonstration of all major directives

sphinx-lesson

| | |
|---|-----------------|
|  Demo | demo |
|  Demo | |
|  Type-Along | type-along |
|  Type-Along | |
|  Exercise | exercise |
|  Solution | solution |
|  Homework | homework |
|  Discussion | discussion |
| Instructor note | instructor-note |
|  Prerequisites | |

prerequisites

Sphinx default

Note

note

Important

important

→ See also

seealso

Warning

warning

Danger

danger

Carpentries holdovers

Questions

questions

Objectives

objectives

Keypoints

keypoints

Callout

callout

Exercise

challenge

❗ Checklist

checklist

⚙️ Prerequisites

prereq

❗ Testimonial

testimonial

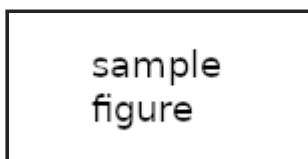
❗ Output

output

Figures

The `figure` directive inserts an image and also provides a caption and other material.

- The path is the relative or absolute path *within the sphinx source directory*.
- You can give optional CSS classes, `with-border` gives it a black border. Remove this if you don't want it - the examples below include it.



This is the caption.

In ReST, this is:

```
.. figure:: img/sample-image.png
   :class: with-border

   This is the caption.
```

In MyST Markdown, this is:


```
```{figure} img/sample-image.png

class: with-border

This is the figure caption.
```
```

When adding figures, optimize for narrow columns. First off, many themes keep it in a small column but even if not, learners will usually need to keep the text in a small column anyway, to share the screen with their workspace. If you are 690 or less, then sphinx_rtd_theme has no image scaling. 600 or less may be best. If larger, then they may be scaled down (in some themes) or scroll left (others).

Sample episode in MyST (Markdown)

? Questions

- What syntax is used to make a lesson?
- How do you structure a lesson effectively for teaching?

questions are at the top of a lesson and provide a starting point for what you might learn. It is usually a bulleted list. (The history is a holdover from carpentries-style lessons, and is not required.)

! Objectives

- Show a complete lesson page with all of the most common structures.
- ...

This is also a holdover from the carpentries-style. It could usually be left off.

A first paragraph really motivating *why* you would need the material presented on this page, and why it is exciting. Don't go into details.

Then, another paragraph going into the big picture of *what* you will do and *how* you will do it. Not details, but enough so that someone knows the overall path.

[For the syntax of ReST, you really want to browse this page alongside the source of it, to see how this is implemented. See the links at the to right of the page.]

Section titles should be enough to understand the page

The first paragraph of each section should again summarize what you will do in it.

Top-level section titles are the map through the page and should make sense together.


This is text.

A code block with preceeding paragraph:

```
import multiprocessing
```

- A bullet list
- Bullet list
 - Sub-list:

```
code block (note indentation)
```


 **Note**

directive within a list (note indentation)

Python

R

```
import bisect
a = 1 + 2
```


 This is a **discussion** directive

Discussion content.

Exercise: [the general topic]

These exercises will show basic exercise conventions. It might be useful for the first paragraph of a multi-exercise section to tie them together to the overall point, but that isn't necessary.

[Exercises get their own section, so that they can be linked and found in the table of contents.]

 **ReST-1 Imperative statement of what will happen in the exercise.**

An intro paragraph about the exercise, if not obvious. Expect that learners and exercise leaders will end up here without having browsed the lesson above. Make sure that they understand the general idea of what is going on and *why* the exercise exists (what the learning objective is roughly, for example there is a big difference between making a commit and focusing on writing a good commit message and knowing the command line arguments!)

1. Bullet list if multiple parts.
2. Despite the names, most exercises are not really “exercises” in that they are difficult. Most are rather direct applications of what has been learned (unless they are **(advanced)**).
3. When writing the exercise steps, try to make it clear enough that a helper/exercise leader who knows the general tools somewhat well (but doesn’t know the lesson) can lead the exercise just by looking at the text in the box.
 - Of course that’s not always possible, sometimes they actually are difficult.

✓ Solution

- Solution here.

(optional) ReST-2 Imperative statement of what will happen in the exercise.

1. Optional exercises are prefixed with **(optional)**
2. It’s better to have more exercises be optional than many that are made optional ad-hoc. Every instructor may do something different, but it’s better to seem like you are covering all the main material than seem like you are skipping parts.

✓ Solution

- Solution to that one.

(optional) ReST-3: Exercise with embedded solution

1. This exercise has the solution within its box itself. This is a stylistic difference more than anything.

✓ Solution

- Solution to that one.

(advanced) ReST-4: Exercise with embedded solution

1. `(advanced)` is the tag for things which really require figuring out stuff on your own.
Can also be `(advanced, optional)` but that's sort of implied.
2. This also demonstrates an exercise with a [link](#), or [internal reference](#).

This entire section is an exercise

! Exercise leader setup

This admonition is a drop-down and can be used for instructor or exercise-leader specific setup. (see also / compare with `instructor-note` .

In this section, we will [do something]

Standard intro paragraph of the exercise.

Describe how this exercise is following everything that is in this section.

Do this.

Then do that.

And so on.

Another section

Instructor note

This is an instructor note. It may be hidden, collapsed, or put to the sidebar in a later style. You should use it for things that the instructor should see while teaching, but should be de-emphasized for the learners. Still, we don't hide them for learners (instructors often present from the same view.)

These tab synchronize with those above:

Python

R

```
import cmath  
a = 10 / 2
```

! Advanced info that should be hidden

Any advanced information can be hidden behind any admonition by adding a `dropdown` class to it (syntax: `:class: dropdown` as first line separated by a space).

This can be useful for advanced info that should not be show in the main body of text..

A subsection

Subsections are fine, use them as you want. But make sure the main sections tell the story and provide a good table of contents to the episode.

sample
figure

Figure caption here.

sample
figure

Figure caption here, which explains the content in text so that it's accessible to screen readers.

Other directives

→ See also

A reference to something else. Usually used at the top of a section or page to highlight that the main source of information is somewhere else. Regular-importance “see also” is usually at a section at the bottom of the page or an a regular paragraph text.

📌 Important

This is used for things that should be highlighted to prevent significant confusion. It's not *that* often used.

⚠ Warning

Something which may result in data loss, security, or massive confusion. It's not *that* often used.

What's next?

Pointers to what someone can learn about next to expand on this topic, if relevant.

Summary

A summary of what you learned.

See also

A “see also” section is good practice to show that you have researched the topic well and your lesson becomes a hub pointing to the other best possible resources.

- Upstream information
- Another course

! Keypoints

- What the learner should take away
- point 2
- ...

This is another holdover from the carpentries style. This perhaps is better done in a “summary” section.

Sample episode in ReST

? Questions

- What syntax is used to make a lesson?
- How do you structure a lesson effectively for teaching?

`questions` are at the top of a lesson and provide a starting point for what you might learn. It is usually a bulleted list. (The history is a holdover from carpentries-style lessons, and is not required.)

! Objectives

- Show a complete lesson page with all of the most common structures.
- ...

This is also a holdover from the carpentries-style. It could usually be left off.

A first paragraph really motivating *why* you would need the material presented on this page, and why it is exciting. Don’t go into details.

Then, another paragraph going into the big picture of *what* you will do and *how* you will do it. Not details, but enough so that someone knows the overall path.

[For the syntax of ReST, you really want to browse this page alongside the source of it, to see how this is implemented. See the links at the to right of the page.]

Section titles should be enough to understand the page

The first paragraph of each section should again summarize what you will do in it.

Top-level section titles are the map through the page and should make sense together.

This is text.

A code block with preceding paragraph:

```
import multiprocessing
```

- A bullet list
- Bullet list
 - Sub-list:

```
code block (note indentation)
```

! Note

directive within a list (note indentation)

Python

R

```
import bisect
a = 1 + 2
```

💬 This is a *discussion* directive

Discussion content.

Exercise: [the general topic]

These exercises will show basic exercise conventions. It might be useful for the first paragraph of a multi-exercise section to tie them together to the overall point, but that isn't necessary.

[Exercises get their own section, so that they can be linked and found in the table of contents.]

ReST-1 Imperative statement of what will happen in the exercise.

An intro paragraph about the exercise, if not obvious. Expect that learners and exercise leaders will end up here without having browsed the lesson above. Make sure that they understand the general idea of what is going on and *why* the exercise exists (what the learning objective is roughly, for example there is a big difference between making a commit and focusing on writing a good commit message and knowing the command line arguments!)

1. Bullet list if multiple parts.
2. Despite the names, most exercises are not really “exercises” in that they are difficult. Most are rather direct applications of what has been learned (unless they are **(advanced)**).
3. When writing the exercise steps, try to make it clear enough that a helper/exercise leader who knows the general tools somewhat well (but doesn’t know the lesson) can lead the exercise just by looking at the text in the box.
 - Of course that’s not always possible, sometimes they actually are difficult.

✓ Solution

- Solution here.

(optional) ReST-2 Imperative statement of what will happen in the exercise.

1. Optional exercises are prefixed with **(optional)**
2. It’s better to have more exercises be optional than many that are made optional ad-hoc. Every instructor may do something different, but it’s better to seem like you are covering all the main material than seem like you are skipping parts.

✓ Solution

- Solution to that one.

(optional) ReST-3: Exercise with embedded solution

1. This exercise has the solution within its box itself. This is a stylistic difference more than anything.

✓ Solution

- Solution to that one.

(advanced) ReST-4: Exercise with embedded solution

1. `(advanced)` is the tag for things which really require figuring out stuff on your own.
Can also be `(advanced, optional)` but that's sort of implied.
2. This also demonstrates an exercise with a [link](#), or [internal reference](#).

This entire section is an exercise

Exercise leader setup

This admonition is a drop-down and can be used for instructor or exercise-leader specific setup. (see also / compare with `instructor-note` .

In this section, we will [do something]

Standard intro paragraph of the exercise.

Describe how this exercise is following everything that is in this section.

Do this.

Then do that.

And so on.

Another section

Instructor note

This is an instructor note. It may be hidden, collapsed, or put to the sidebar in a later style. You should use it for things that the instructor should see while teaching, but should be de-emphasized for the learners. Still, we don't hide them for learners (instructors often present from the same view.)

These tab synchronize with those above:

Python

R

```
import cmath  
a = 10 / 2
```

❗ Advanced info that should be hidden

Any advanced information can be hidden behind any admonition by adding a `dropdown` class to it (syntax: `:class: dropdown` as first line separated by a space).

This can be useful for advanced info that should not be show in the main body of text..

A subsection

Subsections are fine, use them as you want. But make sure the main sections tell the story and provide a good table of contents to the episode.

sample
figure

Figure caption here.

sample
figure

Figure caption here, which explains the content in text so that it's accessible to screen readers.

Other directives

→ See also

A reference to something else. Usually used at the top of a section or page to highlight that the main source of information is somewhere else. Regular-importance “see also” is usually at a section at the bottom of the page or an a regular paragraph text.

❗ Important

This is used for things that should be highlighted to prevent significant confusion. It's not *that* often used.

❗ Warning

Something which may result in data loss, security, or massive confusion. It's not *that* often used.

What's next?

Pointers to what someone can learn about next to expand on this topic, if relevant.

Summary

A summary of what you learned.

See also

A “see also” section is good practice to show that you have researched the topic well and your lesson becomes a hub pointing to the other best possible resources.

- Upstream information
- Another course

! Keypoints

- What the learner should take away
- point 2
- ...

This is another holdover from the carpentries style. This perhaps is better done in a “summary” section.

Intersphinx: easy linking

There is a common problem: you want to link to documentation in other sites, for example the documentation of `list.sort`. Isn't it nice to have a structured way to do this so you don't have to a) look up a URL yourself b) risk having links break? Well, what do you know, Sphinx has a native solution for this: `Intersphinx`.

Enable the extension

It's built into Sphinx, and in the sphinx-lesson-template `conf.py` but commented out. Enable it:

```
extensions.append('sphinx.ext.intersphinx')
intersphinx_mapping = {
    'python': ('https://docs.python.org/3', None),
}
```

Configuration details and how to link to other sites are found at [the docs for intersphinx](#). For most Sphinx-documented projects, use the URL of the documentation base. See “Usage” below for how to verify the URLs.

Usage

Just like `:doc:` is a structured way to link to other documents, there are other **domains** of links, such as `:py:class:`, `:py:meth:`, and so on. So we can link to documentation of a class or method like this:

! Rendered (note the links)

The `list` class `sort` method.

```
# Restructured Text
The :py:class:`list` class :py:meth:`sort <list.sort>` method.
```

```
# MyST markdown
The {py:class}`list` class {py:meth}`sort <list.sort>` method.
```

Note that this is structured information, and thus has no concept in Markdown, only MyST “markdown”. This is, in fact, a major reason why plain markdown is not that great for structured documentation.

Available linking domains and roles

Of course, the domains are extendable. Presumably, when you use sphinx-lesson, you will be referring to other things. The most common roles in the Python domain are:

- `:py:mod:` : modules, e.g. `multiprocessing`
- `:py:func:` : modules, e.g. `itertools.combinations()`
- `:py:class:` : modules, e.g. `list`
- `:py:meth:` : modules, e.g. `list.sort()`
- `:py:attr:` : modules, e.g. `re.Pattern.groups`
- `:py:data:` : modules, e.g. `datetime.MINYEAR`
- Also `:py:exc:`, `:py:data:`, `:py:obj:`, `::`, `:::`
- There are also built-in domains for C, C++, JavaScript (see [the info on Sphinx domains](#) for what the roles are). Others are added by Sphinx extensions.

You can list all available reference targets at some doc using a command line command. You can get the URL from the conf.py file (and use this to verify URLs before you put it in the conf.py file):

```
# Note we need to append `objects.inv`:
python -m sphinx.ext.intersphinx https://docs.python.org/3/objects.inv
# In conf.py: 'python': ('https://docs.python.org/3', None),
```

You usually use the fully qualified name of an object, for example `matplotlib.pyplot.plot`. In Python this is usually pretty obvious, due to clear namespacing. You'll have to look at other languages yourself.

See also

- [Sphinx: domains](#) - how to document classes/functions to be referrable this way, and link to them.
- [Intersphinx](#).

Markdown transforms

This is mostly obsolete and unmaintained now that almost everything is transitioned to MyST-markdown. If this is important for you, get in touch and we can revive it.

To ease the transition from other Markdown dialects (like the one used in software-carpentry), we implement some transformations in Sphinx which happen as source preprocessing. These are implemented in the `sphinx_lessons.md_transforms` module and are implemented using regular expressions, so they are a bit fragile.

Code fences

Code fence syntax is translated to CommonMark. Input:

```
```
blah
```

{: output}
```

Output:

```
```${output}`
blah
```
```

Block quotes

Transform CSS styles into MyST directives (implemented as code fences. Input:

```
> ## some-heading
> text
> text
{: .block-class}
```

Output:

```
``{block-class} some-heading
text
text
``
```

The `block-class` is the directive name (we maintain compatibility with old jekyll-common)

Raw HTML images

Raw HTML isn't a good idea in portable formats. Plus, in the old jekyll formats, bad relative path handling caused absolute paths to be embedded a lot. Transform this:

```

```

into this:

```
``{figure} /path/to/img.png
``
```

Exclude any possible `{{ ... }}` template variables used to semi-hard code absolute paths.

Jupyter notebook page

This is a raw Jupyter notebook page, in `.ipynb` format.

Usage

Basic code execution works:

```
!hostname
print(sum(range(10)))
```

```
fv-az1676-418
```

Directives work as well, since it is recursively executed by the same parser as normal Markdown files. Of course, Sphinx directives won't be interpreted within Jupyter itself, but that is probably OK for development purposes:

Exercise

This is a challenge block

You can learn more about this at the [myst-nb docs](#). Most this applies equally to the `.md` files.

Configuration

The `nb_execution_mode` variable in `conf.py` controls execution. Currently, we don't set a default for this, which makes it `auto`. You can read more about this [upstream](#). Possible values include:

- `off`: don't execute
- `auto`: execute *only if there are missing outputs
- `force`: always re-execute
- `cache`: always execute, but cache the output. Don't re-execute if the input is unchanged.

Executing code cells in MyST markdown

This is a `.md` file that has a special header and a `{code-cell}` directive, that lets it

```
!hostname
```

```
fv-az1676-418
```

You can see how this works [on the myst-nb docs](#).

Substitutions and replacement variables

Like most languages, ReST and MyST both have direct ways to substitute variables to locally customize lessons. While this works for simple cases, this can quickly become difficult to manage with the “main copy” of possibly important content being separated from the main body, and keeping the substitutions up to date.

[sphinx_ext_substitution](#) tries to solve these problems by keeping the default values within the document and providing tools to manage the substitution as the context changes over time. It is tested with ReST and should work with MyST as well.

Deployment with Github Actions

In sphinx-lesson-template (and this lesson...), there is a `.github/workflows/sphinx.yml` file that contains a Github Action that:

- Installs dependencies
- Builds the project with Sphinx
- Deploys it
 - If branch = `main`, deploy to github pages normally
 - For other branches, deploy to github-pages but put the result in the `branch/{branch-name}` subdirectory. If the branch name has a `/` in it, replace it with `--`.
 - Keep all previous deployments, but remove branch subdirectories for branches that no longer exist.

This allows you to view builds from pull requests or other branches.

Usage

It is recommended to copy from [sphinx-lesson-template](#), since that is the primary copy that is updated with all of the latest developments.

Direct link: <https://github.com/coderefinery/sphinx-lesson-template/blob/main/.github/workflows/sphinx.yml>

Presentation mode

Note

This is a technical demo, included by default to make it easy to use. It may be removed in a future release.

Using [minipres](#), any web page can be turned into a presentation. As usual, there is nothing very specific to sphinx-lesson about this, but currently minipres is only tested on `sphinx_rtd_theme`, but theoretically can work on others.

Using minipres, you only have to write one page: the material your students read. Then, you hide the unnecessary elements (table of contents), focus on one section, and provide a quick way to jump between sections. Then you can get the focused attention of a presentation and the readability of a single page.

How it works:

- Add `?minipres` to the URL of any page, and it goes into presentation mode.
- Add `?plain` to the URL of any page to go to plain mode.

In presentation mode:

- The sidebars are removed (this is the only thing that happens in `plain` mode).
- Extra space is added between each section (HTML headings), so that you focus on one section at a time
- The **left/right arrow keys** scroll between sections.

Examples:

- [View this page as an example of minipres.](#)
- [View this page in “plain” mode.](#)

Indexing

An index lets you efficiently look up any topic. In the age of full-text search, you are right to wonder what the point of indexes are. They could be seen as companion of cheatsheet: instead of searching and hoping you find the right place, you can index the actual locations to which one would refer.

As you might expect, there is nothing special to in sphinx-lesson about indexing: see the Sphinx documentation on `index`.

Basic concepts

Headings are the terms which can be looked up in the index. When choosing headings, consider:

- What is useful to a reader to locate
- How would a reader look it up? For example, `commit` or `committing` is useful, but `how to commit` is not. Phrase it with the most important terms first (big-endian)
- And index can have sub-entries. For example, under the entry `git`, there can be subentries for each git command, such as `commit`.

Syntax

→ See also

The Sphinx documentation on `index`.

The `{index}` directive and role are the main ways to add index entries. The semicolon (`;`) character separates entries and subentries.

Index a block with a directive

MyST:

```
```${index} commit; amend
```

```${index}
commit
commit; message
pair: commit; amend
```
```

ReST:

```
.. index:: commit; amend
```

Or ReST, multiple:

```
.. index::
    commit
    commit; message
    pair: commit; amend
```

Index a single word with the role

MyST:

```
Simple entry: {index}`commit`
Pair entry: {index}`loop variables <pair: commit; amend>`
```

ReST:

This sentence has an index entry for `:index:`commit``. If you want the indexed term to be different, standard syntax applies such as `:index:`loop variables <pair: commit; amend>``.

Styles of indexes

- `TERM`, same as below
- `single: TERM` (the default): create just a single entry
- `pair: TERM; TERM`: create entries for `x; y` and `y; x`
- `see: TOPIC; OTHER`: creates a “see other” entry for “topic”.
- `seealso: TOPIC; OTHER`: creates a “seealso” entry, like above

Glossaries

If you make a glossary using the `glossary directive`, the terms automatically get added to the index

See also

- `index` directive
- Sphinx `glossary`

Exercise list

The `exercise-list` directive inserts a list of all exercise and solution [directives](#) (and maybe more). This can be useful as an overall summary of the entire lesson flow, onboarding new instructors and helpers, and more.

Usage

MyST:

```
```${exerciselist}```
```

ReST:

```
.. exerciselist::
```

One can give the optional directive arguments to specify lists of admonition classes to include (default: `exercise`, `solution`, `exerciselist-include`) or exclude (default: `exerciselist-exclude`) if you want to (any [directives](#) which match any `include`, and do not match any `exclude` are included). Specify the options this way (ReST):

MyST:

```

```{exerciselist}
:include: exercise solution instructor-note
:exclude: exclude-this
```

:::{exercise} Exercise title
:class: exclude-this

Exercise content
:::

```

ReST:

```

.. exerciselist::
 :include: exercise solution instructor-note
 :exclude: exclude-this

.. exercise:: Exercise title
 :class: exclude-this

 Exercise content

```

This feature is new as of early 2022, there may be possible problems in it still - please report. Currently, only sphinx-lesson admonitions can be included due to technical considerations (see source for hint on fixing).

## Recommendations to make a useful list

- Context is important! Give your exercises a name other than the default of “Exercise”, so that someone quickly scanning the exercise list can follow the overall flow.
  - Making good summaries is really an important skill for organizing anything - give this the attention it needs.
  - Think of an exercise leader or helper coming to help someone, seeing the exercise, and needing to help someone: not just what to do, but what the core lesson and task is, so that they can focus on giving the right help (and telling the learners what they don’t need to worry about).
  - Context can be both in the exercise title and in the exercise body itself.
- Name the exercises well. Best is to think of it like a version control commit: an imperative sentence stating what the person will do in the exercise. For example:

```

Make your first git commit
Resolve the conflict

```

- In the title, include any other important information (see below):

```
Create a setup.py file for your package (15 min)
(optional) Install your package in editable mode using ``pip install -e`` (5 min)
(advanced) Also create packaging using pyproject.toml and compare (20 min)
```

- Consider giving your exercises permanent identifiers. They are intentionally not auto-numbered yet (what happens when more exercises are added/removed?), but if you give them an ID, they will be findable even later. Suggestion is `Episodetopic-N`:

```
Basic-1: Verify git is installed
Basic-2: Initialize the repository
Conflicts-2: Create a new branch for the other commit.
Internals-1: (advanced): Inspect individual objects with ``git cat-file``
```

- It could include not just what you do, but a bit about why you are doing it and what you are learning.
- The list includes only `exercise`, `type-along`, and `solution`. For backwards compatibility, `challenge` is also included.
- Optional or advanced exercises should clearly state it in the exercise title, since people will browse the list separate from the main lesson material.
- Try to minimize use of `:include:` and `:exclude:` and use the defaults and adjust your directives to match sphinx-lesson semantics. Excess use of this may over-optimize for particular workshops

## Example

This section contains the exercise list of sphinx-lesson. Since sphinx-lesson has many examples of exercises, the list below is confusing and doesn't make a lot of sense. You can see a better example at [git-intro's exercise list](#).

Example begins:

## Converting an old lesson

This was the old CodeRefinery procedure. It is now unmaintained but might be useful for someone. Let us know if you need it revived.

## Convert a Jekyll lesson

This brings in the necessary files

Add the template lesson as a new remote:

```
git remote add s-l-t https://github.com/coderefinery/sphinx-lesson-template.git
git fetch s-l-t
```

Check out some basic files into your working directory. Warning: if you add a `.github/workflows/sphinx.yml` file, even a push to a branch will **override github pages**:

```
git checkout s-l-t/main -- requirements.txt
git checkout s-l-t/main -- .github/workflows/sphinx.yml
```

If you need more Sphinx files:

```
git checkout s-l-t/main -- content/conf.py
git checkout s-l-t/main -- .gitignore Makefile make.bat
```

If you need the full content (only `index.rst` for now):

```
git checkout s-l-t/main -- content/
```

(if jekyll conversion) Move content over:

```
git mv _episodes/* content/
```

(if jekyll conversion) Copy stuff from `index.md` into `content/index.rst`.

(if jekyll conversion) Remove old jekyll stuff:

```
git rm jekyll-common/ index.md _config.yml Gemfile .gitmodules
```

Set up github pages (first commit to trigger CI), see [Installation](#):

```
git checkout -b gh-pages origin/gh-pages
git commit -m 'empty commit to trigger gh-pages' --allow-empty
git push
```

Do all the rest of the fixing... all the bad, non-portable, non-relative markdown and so on. This is the hard part. Common problems:

- Non-consecutive section headings

- Multiple top-level section headings (there should be one top-level section heading that is the page title)
- Weird relative links (most work though)

You can also update your local view of the default branch:

```
git remote set-head origin --auto
```

## Joint history

This option joins the histories of the two repositories, so that you could merge from the template repository to keep your files up to date. **This may not currently work**, and also may not have any value (but is kept here for reference until later).

Merge the two unrelated histories:

```
$ git remote add template https://github.com/coderefinery/sphinx-lesson-template
$ git fetch template
$ git merge template/main --allow-unrelated-histories
Resolve any possible merge conflicts
$ git checkout --theirs .gitignore
$ git checkout --ours LICENSE
$ git add .gitignore LICENSE
$ git commit -m 'merge sphinx-lesson history to this lesson'
```

Then proceed like the previous section shows.

## Cheatsheet

Nothing here yet. See the sample episodes:

- [MyST markdown](#)
- [ReST](#)
- [Index](#)
- [Search Page](#)