How to document your research software

In this lesson we will discuss different solutions for implementing and deploying code documentation.

We will start with a discussion about **what makes a good README**. For many projects, a README is more than enough.

We will then learn how to build documentation with the **documentation generator** Sphinx (and compare it with others) and how to deploy it to Read the Docs, a service which hosts open documentation for free.

This demonstration will be **independent of programming languages**.

We will also learn how to deploy a **project website or personal homepage** to GitHub Pages. The approach that we will learn will be transferable to GitLab Pages and Bitbucket Pages.

⚙ Prerequisites

1. Basic understanding of Git.
2. For the Sphinx part, You need to have sphinx and sphinx_rtd_theme installed (they are part of the coderefinery environment).
3. For the GitHub Pages part you need a GitHub account.

If you wish to follow in the terminal and are new to the command line, we recorded a short shell crash course.

10 min	Motivation and wishlist
5 min	Popular tools and solutions
10 min	In-code documentation
20 min	Writing good README files
10 min	Sphinx and Markdown: Quickstart
10 min	BREAK
22 min	Sphinx and Markdown: Exercises
22 min	Deploying Sphinx documentation to GitHub Pages
15 min	Hosting websites/homepages on GitHub Pages

5 min	Summary

Motivation and wishlist

Motivation

💬 Motivation-1: Why documenting code?

Use the collaborative document:

- Is project documentation important? Why?
- How would you describe a useful documentation?
- How can you motivate your colleagues to contribute to the documentation?

✔ Our motivation (but let us brainstorm first)

- You will probably use your code in the future and may forget details.
- You may want others to use your code (almost impossible without documentation).
- You may want others to contribute to the code.
- Shield your limited time and let the documentation answer FAQs.

What do we expect from a suitably good documentation?

❗ Note

Documentation comes in different forms - what *is* documentation?

- **Tutorials**: learning-oriented, allows the newcomer to get started
- **How-to guides**: goal-oriented, shows how to solve a specific problem
- **Explanation**: understanding-oriented, explains a concept
- **Reference**: information-oriented, describes the machinery

Documentation should be written with the reader in mind, and there are more than one kind of reader!

Read more:

- https://www.divio.com/blog/documentation/
- https://diataxis.fr/
- https://www.writethedocs.org/guide/writing/beginners-guide-to-docs/

There is no one size fits all: often for small projects a `README.md` or `README.rst` can be enough (more about these formats later).

Creating a checklist

💬 Motivation-2: Create a wishlist

Use the collaborative document:

- Let us create a wishlist for how we would like documentation to be.
- Below are some of our ideas but please do not look at them yet.
- We are sure you will come up with ideas we did not think about.

✔ Our wishlist (but let us brainstorm first)

First of all: we want documentation to stay correct, that is, it must be easy to maintain.

Versions

- Your code project should be versioned (version control).
- Enable reproducibility and avoid confusion: **documentation should be versioned** as well.
- Have you ever seen: *"We will soon release a new version and are updating the documentation. Some features may not be available in the version you have downloaded."*?

Documentation should be placed and tracked close to the source code

- Documenting **close to the source code** (e.g. subdirectory `doc/`) minimizes barrier to contribute.
- I should not need to log in to another machine or service and jump through hoops to contribute.
- It is often good enough to have a `README.md` or `README.rst` along with your code/script.

Use a standard markup language

Markup

Markup is a set of human readable instructions that is used to tell the computer how a document shall be styled and structured. By using a markup language we can for example write a `*` or `-` where we want a bullet point to appear in the rendered document.

- offers formatting flexibility, enforces a basic document structure and the rendered documents can be exported to other formats (e.g. for printing). Also, the source can be read by humans without knowledge of the language in case the rendered document is unavailable.

- We suggest to use either reStructuredText (RST) or Markdown markup.
- GitHub and GitLab automatically render `README.md` or `README.rst` files.

Copy-paste-able

- PDF alone is not enough since **copy-pasting out of a PDF document can be difficult**.
- It is OK to provide a generated PDF in addition to a copy-paste-able format.

Written by humans

- Automatically generated documentation (e.g. API documentation) is useful as complementary documentation but it does not replace tutorials written by humans.

  Humans know what other similar humans need

Installation instructions

- Give **step by step instructions for the basic case**. Additional information and caveats can be linked from there.
- List requirements and dependencies (libraries, compilers, environment).
- Include instructions for how to test for correctness after installation.

Make the license explicit

- **Include a LICENSE file** with your source code.
- Without a license, your work is under exclusive copyright by default: others are not allowed to re-use or modify anything.
- GitHub and GitLab allows to choose a license from common license templates.

Information for contributors

- Make it easy for others to contribute: **document how you prefer others to contribute**.
- Users of your code may be shy to contribute code. Your **documentation provides a platform for your first contributions**.

Correctness of some kind of documentation is testable automatically! See our episodes about Jupyter Notebooks later and about testing tomorrow.

Documentation checklist

Which items to include depends on the number of users apart from yourself.

- Purpose
- Authors
- License

- Recommended citation
- Copy-paste-able example to get started
- Dependencies and their versions or version ranges
- Installation instructions
- Tutorials covering key functionality ... and if you make them using Jupyter notebooks they can also be validated!
- Reference documentation (e.g. API) covering all functionality
- How do you want to be asked questions (mailing list or forum or chat or issue tracker)
- Possibly a FAQ section
- Contribution guide

Keypoints

- Documentation is part of the code and should be versionable.
- Documentation (sources) should be tracked with the corresponding code in the same repository.
- Use standard markup languages such as reStructuredText or Markdown.

Popular tools and solutions

Questions

- What tools are out there?
- What are their pros and cons?

Objectives

- Choose the right tool for the right reason.

Can code self-document? Some people say so, but it is hard!

In-code documentation

- Comments, function docstrings, ...
- Advantages
  - Good for programmers
  - Version controlled alongside code
  - Can be used to auto-generate documentation for functions/classes
- Disadvantage
  - Probably not enough for users of the code

We will have a closer look at this in the In-code documentation episode.

README files

- Advantages
  - Versioned (goes with the code development)
  - It is often good enough to have a `README.md` or `README.rst` along with your code/script
- If you use README files, use either RST or Markdown
- A great guide to README files: MakeaREADME

We will have a closer look at this in the Writing good README files episode.

Talking about READMEs: they are possibly the most AI-Friendly documentation format (you can just copy&paste the whole README in the context window of the LLM of your choice)

---

reStructuredText and Markdown

```
# This is a section in Markdown   This is a section in RST
                                  ========================

## This is a subsection           This is a subsection
                                  --------------------

Nothing special needed for        Nothing special needed for
a normal paragraph.               a normal paragraph.

                                  ::

    This is a code block          This is a code block


**Bold** and *emphasized*.        **Bold** and *emphasized*.

A list:                           A list:
- this is an item                 - this is an item
- another item                    - another item

There is more: images,            There is more: images,
tables, links, ...                tables, links, ...
```

- Two of the most popular lightweight markup languages.
- reStructuredText (RST) has more features than Markdown but the choice is a matter of taste.
- There are (unfortunately) many flavors of Markdown.
- Motivation to stick to a standard text-based format: **They make it easier to move the documentation to other tools which also expect a standard format, as the project/organization grows**.
- We will use MyST flavored Markdown in the Sphinx and Markdown episode and the Hosting websites/homepages on GitHub Pages example.

- Nice resource to learn Markdown: Learn Markdown in 60 seconds
- Pandoc can convert between MD and RST (and many other formats).

HTML static site generators

There are many tools that can turn RST or Markdown into beautiful HTML pages:

- Sphinx ← **we will exercise this, this is how this lesson material is built**
  - Generate HTML/PDF/LaTeX from RST and Markdown.
  - Basically all Python projects use Sphinx but **Sphinx is not limited to Python**.
  - Read the docs hosts public Sphinx documentation for free!
  - Also hostable anywhere else, like Github pages.
  - API documentation possible.
- Jekyll
  - Generates HTML from Markdown.
  - GitHub supports this without adding extra build steps.
- pkgdown
  - Popular in the R community
- MkDocs
- GitBook
- Hugo
- Hexo
- Zola **<- this is what we use for our project website and workshop websites**
- There are many more …

GitHub, GitLab, and Bitbucket make it possible to serve HTML pages:

- GitHub Pages
- Bitbucket Pages
- GitLab Pages

Wikis

- Popular solutions (but many others exist):
  - MediaWiki
  - Dokuwiki
- Advantage
  - Barrier to write and edit is low
- Disadvantages
  - Typically disconnected from source code repository (**reproducibility**)
  - Difficult to serve multiple versions
  - Difficult to check out a specific old version
  - Typically needs to be hosted and maintained

LaTeX/PDF

- Advantage
  - Popular and familiar in the physics and mathematics community
- Disadvantages
  - PDF format is not ideal for copy-pasting of examples
  - Possible, but not trivial to automate rebuilding documentation after every Git push

---

Doxygen

- Auto-generates API documentation
- Documented directly in the source code
- Popular in the C++ community
- Has support for C, Fortran, Python, Java, etc., see Doxygen Github Repo
- Many keywords are understood by Doxygen: Doxygen special commands
- Can be used to also generate higher-level ("human") documentation
- Can be deployed to GiHub/GitLab/Bitbucket Pages

If you prefer the look&feel of Sphinx, you can still use Doxygen as a backend and Sphinx through Breathe.

Other tools

- Fortran
  - Fortran Documenter (FORD)
- Julia
  - Franklin: static site generator
  - Documenter.jl
- Quarto converts markdown to websites, pdfs, ebooks and many other things

---

Keypoints

- Some popular solutions make reproducibility and maintenance of multiple code versions difficult.

In-code documentation

Questions

- What can I do to make my code more easily understandable?
- What information should go into comments?

- What are docstrings and what information should go into docstrings?

In this episode we will learn how to write good documentation inside your code.

Exercise - Writing good comments

✍ In-code-1: Comments

Let's take a look at two example comments (comments in Python start with `#`):

Comment A

```
# now we check if temperature is below -50
if temperature < -50:
    print("ERROR: temperature is too low")
```

Comment B

```
# we regard temperatures below -50 degrees as measurement errors
if temperature < -50:
    print("ERROR: temperature is too low")
```

Which of these comments is more useful? Can you explain why?

✓ Solution

- Comment A describes **what** happens in this piece of code. This can be useful for somebody who has never seen Python or a program, but for somebody who has, it can feel like a redundant commentary.
- Comment B is probably more useful as it describes **why** this piece of code is there, i.e. its **purpose**.

Sometimes version control is better than a comment

❗ Examples for code comments where Git is a better solution

Keeping zombie code "just in case" (rather use version control):

```
# do not run this code!
# if temperature > 0:
#     print("It is warm")
```

Instead: Remove the code, you can always find it back in a previous version of your code in Git.

Emulating version control:

```
# John Doe: threshold changed from 0 to 15 on August 5, 2013
if temperature > 15:
    print("It is warm")
```

Instead: You can get this information from `git log` or `git show` or `git annotate` or similar.

What are “docstrings” and how can they be useful?

Here is function `fahrenheit_to_celsius` which converts temperature in Fahrenheit to Celsius, implemented in a couple of different languages. Your language is missing? Please contribute an example.

The first set of examples uses **regular comments**:

Python | R | Julia | Fortran | C++ | Rust

```
# This function converts a temperature in Fahrenheit to Celsius.
def fahrenheit_to_celsius(temp_f: float) -> float:
    temp_c = (temp_f - 32.0) * (5.0/9.0)
    return temp_c
```

The second set uses **docstrings or similar concepts**. Please compare the two (above and below):

Python | R | Julia | Fortran | C++ | Rust

```python
def fahrenheit_to_celsius(temp_f: float) -> float:
    """
    Converts a temperature in Fahrenheit to Celsius.

    Parameters
    ----------
    temp_f : float
        The temperature in Fahrenheit.

    Returns
    -------
    float
        The temperature in Celsius.
    """

    temp_c = (temp_f - 32.0) * (5.0/9.0)
    return temp_c
```

Read more: https://peps.python.org/pep-0257/

Docstrings can do a bit more than just comments:

- Tools can generate help text automatically from the docstrings.
- Tools can generate documentation pages automatically from code.

It is common to write docstrings for functions, classes, and modules.

Good docstrings describe:

- What the function does
- What goes in (including the type of the input variables)
- What goes out (including the return type)

Naming is documentation: Giving explicit, descriptive names to your code segments (functions, classes, variables) already provides very useful and important documentation. In practice you will find that for simple functions it is unnecessary to add a docstring when the function name and variable names already give enough information.

Coming up with good comments/docstrings (as with good variable and function names) typically requires some iterations. Module/file names help too! Also, remove clutter! Unused files, unused functions...

Keypoints

- Comments should describe the why for your code not the what.

- Writing docstrings can be a good way to write documentation while you type code since it also makes it possible to query that information from outside the code or to auto-generate documentation pages.

Writing good README files

The README file (often `README.md` or `README.rst`) is usually the first thing users/collaborators see when visiting your GitHub repository.

Use it to communicate important information about your project! For many smaller or mid-size projects, this is enough documentation. It's not that hard to make a basic one, and it's easy to expand as needed.

Exercise: Have fun testing some README features

✍️ Exercise README-1: Have fun testing some README features you may not have heard about

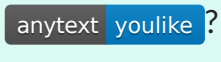
- Test the effect of adding the following to your GitHub README (read more):

```
> [!NOTE]
> Highlights information that users should take into account, even when
skimming.

> [!IMPORTANT]
> Crucial information necessary for users to succeed.

> [!WARNING]
> Critical content demanding immediate user attention due to potential risks.
```

- For more detailed descriptions which you don't want to show by default you might find this useful (please try it out):

```
<details>
<summary>
Short summary
</summary>

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod
tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam,
quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo
consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse
cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non
proident, sunt in culpa qui officia deserunt mollit anim id est laborum.
</details>
```

(curiosity) This is an example of the fact that, if one absolutely wants that, it is legal to use HTML directly in markdown files

- Would you like to add a badge like this one: 

?

Badge that links to a website (see also https://shields.io/):

```
[![please replace with alt text](https://img.shields.io/badge/anytext-youlike-blue)](https://example.org)
```

Badge without link:

```
![please replace with alt text](https://img.shields.io/badge/anytext-youlike-blue)
```

- Know about other tips and tricks? Please share them (send a pull request to this lesson).

Exercise: Improve the README for your own project

✍ Exercise README-2: Draft or improve a README for one of your recent projects

Try to draft a brief README or review a README which you have written for one of your projects.

- You can do that either by screensharing and discussing or working individually.
- Use the checklist which we have discussed earlier.
- Think about the user (**which can be a future you**) of your project, what does this user need to know to use or contribute to the project? And how do you make your project attractive to use or contribute to?
- (Optional): Try the https://hemingwayapp.com/ to analyse your README file and make your writing bold and clear.
- Please note observations and recommendations in the collaborative notes.

Exercise: Discuss the README of a project that you use

✍ Exercise README-3: Review and discuss a README of a project that you have used

In this exercise we will review and discuss a README of a project which you have used. You can also review a library which is popular in your domain of research and discuss their README.

- You can do that either by screensharing and discussing or working individually.
- When discussing other people's projects please remember to be respectful and constructive. The goal of this exercise is not to criticize other projects but to learn from other projects and to collect the aspects that you enjoyed finding in a README and to also collect aspects which you have searched for but which are sometimes missing.
- Please note observations and recommendations in the collaborative notes.

Example: https://github.com/UppASD/UppASD

Example: https://github.com/sa2c/sunpyter

Optional Example: https://gitlab.kit.edu/kit/scs-public/gitlab-ci-custom-executor-slurm-enroot_apptainer

About last one: problem was, how to reduce the burden of maintaining the documentation?

A comment: use semantic line breaks!

Table of contents in README files

- GitHub automatically generates a table of contents for README.md files.
- On GitLab you can generate a TOC in Markdown with:

```
[[_TOC_]]
```

- With ReST you can generate a table of contents (TOC) automatically by adding:

```
.. contents:: Table of Contents
```

This is an example of the anarchy due to the limited specification of markdown, and why makes ReST/MyST attractive

Sphinx and Markdown

❗ Objectives

- Understand how static site generators build websites out of plain text files.
- Create example Sphinx documentation and learn some Markdown along the way.

We will take the first steps in creating documentation using Sphinx, and learn some MyST flavored Markdown syntax along the way.

💬 This lesson is built with Sphinx

Try to compare the source code and the result side by side.

Our goal in this episode is to build HTML pages locally on our computers.

⚙ Before we start, let us verify whether we have the software we need

You may need to activate your CodeRefinery conda environment we set up in the installation instructions. This was covered as part of the installation instructions, but the most usual command to do this is:

```
$ conda activate coderefinery
```

Check whether Python is available (you should see a version; precise version is not so important):

```
$ python --version

Python 3.11.5
```

Check whether Sphinx is available (you should see a version; precise version is not so important):

```
$ sphinx-build --version

sphinx-build 5.3.0
```

Check whether the quickstart tool is available (you should see a version; precise version is not so important):

```
$ sphinx-quickstart --version

sphinx-quickstart 5.3.0
```

Check whether MyST parser is available (you should see no output):

```
$ python -c "import myst_parser"
```

If the above commands produce an error (**command not found** or **module not found** or **ModuleNotFoundError**), please follow our installation instructions. But please don't give up if you don't have these - the episodes after this one will work even without these tools.

Exercise: Sphinx quickstart

✍ Sphinx-1: Generate the basic documentation template

Create a directory for the example documentation, step into it, and inside generate the basic documentation template:

```
$ mkdir doc-example
$ cd doc-example
$ sphinx-quickstart
```

The quickstart utility will ask you some questions. For this exercise, you can go with the default answers except to specify a project name, author name, and project release:

```
> Separate source and build directories (y/n) [n]: <hit enter>
> Project name: <your project name>
> Author name(s): <your name>
> Project release []: 0.1
> Project language [en]: <hit enter>
```

A couple of files and directories are created:

File/directory	Contents
conf.py	Documentation configuration file
index.rst	Main file in Sphinx
_build/	Directory where docs are built (you can decide the name)
_templates/	Your own HTML templates
_static/	Static files (images, styles, etc.) copied to output directory on build
Makefile	Makefile to build documentation using make
make.bat	Makefile to build documentation using make (Windows)

`Makefile` and `make.bat` (for Windows) are build scripts that wrap the sphinx commands, but we will be doing it explicitly.

Let's have a look at the `index.rst` file, which is the main file of your documentation:

```
.. myproject documentation master file, created by
   sphinx-quickstart on Sat Sep 23 17:35:26 2023.
   You can adapt this file completely to your liking, but it should at least
   contain the root `toctree` directive.

Welcome to myproject's documentation!
=====================================

.. toctree::
   :maxdepth: 2
   :caption: Contents:


Indices and tables
==================

* :ref:`genindex`
* :ref:`modindex`
* :ref:`search`
```

- We will not use the `Indices and tables` section now, so remove it and everything below.
- The top four lines, starting with `..`, are a comment.
- The next lines are the table of contents. We can add content below:

```
.. toctree::
   :maxdepth: 2
   :caption: Contents:

   some-feature.md
```

Note that `some-feature.md` needs to be indented to align with `:caption:`.

We now need to tell Sphinx to use markdown files. To do this, we open `conf.py` and replace the line:

```
extensions = []
```

with this line so that Sphinx can parse Markdown files:

```
extensions = ['myst_parser']
```

Let's create the file `some-feature.md` (in Markdown format) which we have just listed in `index.rst` (which uses reStructured Text format).

```
# Some feature

## Subsection

Exciting documentation in here.
Let's make a list (empty surrounding lines required):

- item 1

  - nested item 1
  - nested item 2

- item 2
- item 3
```

We now build the site:

```
$ ls -1

_static
_templates
conf.py
index.rst
make.bat
Makefile
some-feature.md

$ sphinx-build . _build

... lots of output ...
build succeeded.

The HTML pages are in _build.

$ ls -1 _build

_sources
_static
genindex.html
index.html
objects.inv
search.html
searchindex.js
some-feature.html
```

Now open the file `_build/index.html` in your browser.

- Linux users, type:

  ```
  $ xdg-open _build/index.html
  ```

- macOS users, type:

```
$ open _build/index.html
```

- Windows users, type:

```
$ start _build/index.html
```

- If the above does not work: Enter `file:///home/user/doc-example/_build/index.html` in your browser (adapting the path to your case).

Hopefully you can now see a website. If so, then you are able to build Sphinx pages locally. This is useful to check how things look before pushing changes to GitHub or elsewhere.

Note that you can change the styling by editing `conf.py` and changing the value `html_theme` (for instance you can set it to `sphinx_rtd_theme` (if you have that Python package installed) to have the Read the Docs look).

Exercise: Adding more Sphinx content

✍ Sphinx-2: Add more content to your example documentation

1. Add a entry below `some-feature.md` labeled `another-feature.md` (or a better name) to the `index.rst` file.
2. Create a file `another-feature.md` in the same directory as the `index.rst` file.
3. Add some content to `another-feature.md`, rebuild with `sphinx-build . _build`, and refresh the browser to look at the results.
4. Use the MyST Typography page as help.

Experiment with the following Markdown syntax:

- *Emphasized text* and **bold text**
- Headings:

```
# Level 1

## Level 2

### Level 3

#### Level 4
```

- An image: `![alt text](image.png)`
- `[A link](https://www.example.org)`

- Numbered lists (numbers adjusted automatically):

```
1. item 1
2. item 2
3. item 3
1. item 4
1. item 5
```

- Simple tables:

```
| No.  |  Prime |
| ---- | ------ |
| 1    |  No    |
| 2    |  Yes   |
| 3    |  Yes   |
| 4    |  No    |
```

- Code blocks:

````
The following is a Python code block:
```python
  def hello():
      print("Hello world")
```

And this is a C code block:
```c
#include <stdio.h>
int main()
{
    printf("Hello, World!");
    return 0;
}
```
````

- You could include an external file (here we assume a file called “example.py” exists; at the same time we highlight lines 2 and 3):

````
```{literalinclude} example.py
:language: python
:emphasize-lines: 2-3
```
````

- We can also use Jupyter notebooks (*.ipynb) with Sphinx. It requires the myst-nb extension to be installed.

Exercise: Sphinx and LaTeX

✍ Sphinx-3: Rendering (LaTeX) math equations

Math equations should work out of the box. In some older versions, you might need to edit `conf.py` and add `sphinx.ext.mathjax`:

```
extensions = ['myst_parser', 'sphinx.ext.mathjax']
```

Actually, adding this extension is not needed anymore and mathematical formulas are displayed using Mathjax by default.
Try this (result below):

````
This creates an equation:
```{math}
a^2 + b^2 = c^2
```

This is an in-line equation, {math}`a^2 + b^2 = c^2`, embedded in text.
````

This creates an equation:

$$a^2 + b^2 = c^2$$

This is an in-line equation, $a^2 + b^2 = c^2$, embedded in text.

Exercise: Sphinx autodoc

✍ (optional) Sphinx-4: Auto-generating documentation from Python docstrings

1. Write some docstrings in functions and/or class definitions of an `example` python module:

```
def multiply(a: float, b: float) -> float:
    """
    Multiply two numbers.

    :param a: First number.
    :param b: Second number.
    :return: The product of a and b.
    """
    return a * b
```

2. In the file `conf.py` modify “extensions” and add 3 lines:

```
extensions = ['myst_parser', "autodoc2"]

autodoc2_packages = [
    "multiply.py"
]
```

4. List `apidocs/index` in the toctree in `index.rst`.

```
.. toctree::
   :maxdepth: 2
   :caption: Contents:

   some-feature.md
   apidocs/index
```

5. Re-build the documentation and check the “API reference” section.

Confused about reStructuredText vs. Markdown vs. MyST?

- At the beginning there was reStructuredText and Sphinx was built for reStructuredText.
- Independently, Markdown was invented and evolved into a couple of flavors.
- Markdown became more and more popular but was limited compared to reStructuredText.
- Later, MyST was invented to be able to write something that looks like Markdown but in addition can do everything that reStructuredText can do with extra directives.

Good to know

- The `_build` directory is a generated directory and should not be part of the Git repository. We recommend to add `_build` to `.gitignore` to prevent you from accidentally adding files below `_build` to the Git repository.
- sphinx-autobuild provides a local web server that will automatically refresh your view every time you save a file - which makes writing and testing much easier.
- This is useful if you want to check the integrity of all internal and external links:

```
$ sphinx-build . -W -b linkcheck _build
```

References

- Sphinx documentation
- Sphinx + ReadTheDocs guide
- For more Markdown functionality, see the Markdown guide.

- For Sphinx additions, see Sphinx Markup Constructs.
- https://docs.python-guide.org/writing/documentation/

Keypoints

- Sphinx and Markdown is a powerful duo for writing documentation.
- Another option is to use reStructuredText, see the Sphinx documentation and the quick-reference
- In the next episode we will learn how to deploy the documentation to a cloud service and update it upon every `git push`.

Deploying Sphinx documentation to GitHub Pages

Objectives

- Create a basic workflow which you can take home and adapt for your project.

GitHub Pages

- Serve websites from a GitHub repository.
- It is no problem to serve using your own URL `https://myproject.org` instead of `https://myuser.github.io/myproject`.

GitHub Actions

- Automatically runs code when your repository changes.
- We will let it run `sphinx-build` and make the result available to GitHub Pages.

Our goal: putting it all together

- Host source code with documentation sources on a public Git repository.
- Each time we `git push` to the repository, a GitHub action triggers to rebuild the documentation.
- The documentation is pushed to a separate branch called 'gh-pages'.

Exercise - Deploy Sphinx documentation to GitHub Pages

✍ GH-Pages-1: Deploy Sphinx documentation to GitHub Pages

In this exercise we will create an example repository on GitHub and deploy it to GitHub Pages.

Step 1: Go to the documentation-example project template on GitHub and create a copy to your namespace.

- Give it a name, for instance “documentation-example”.
- You don’t need to “Include all branches”
- Click on “Create a repository”.

Step 2: Browse the new repository.

- It will look very familar to the previous episode.
- However, we have moved the documentation part under `doc/` (many projects do it this way). But it is still a Sphinx documentation project.
- The source code for your project could then go under `src/`.

Step 3: Add the GitHub Action to your new Git repository.

- Add a new file at `.github/workflows/documentation.yml` (either through terminal or web interface), containing:

```yaml
name: documentation

on: [push, pull_request, workflow_dispatch]

permissions:
  contents: write

jobs:
  docs:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
      - name: Install dependencies
        run: |
          pip install sphinx sphinx_rtd_theme myst_parser
      - name: Sphinx build
        run: |
          sphinx-build doc _build
      - name: Deploy to GitHub Pages
        uses: peaceiris/actions-gh-pages@v3
        if: ${{ github.event_name == 'push' && github.ref == 'refs/heads/main' }}
        with:
          publish_branch: gh-pages
          github_token: ${{ secrets.GITHUB_TOKEN }}
          publish_dir: _build/
          force_orphan: true
```

- You don’t need to understand all of the above, but you might spot familiar commands in the `run:` sections.
- After the file has been committed (and pushed), check the action at `https://github.com/USER/documentation-example/actions` (replace `USER` with your GitHub username).

Step 4: Enable GitHub Pages

- On GitHub go to “Settings” -> “Pages”.
- In the “Source” section, choose “Deploy from a branch” in the dropdown menu.
- In the “Branch” section choose “gh-pages” and “/root” in the dropdown menus and click save.
- You should now be able to verify the pages deployment in the Actions list).

Step 5: Verify the result

- Your site should now be live on `https://USER.github.io/documentation-example/` (replace USER).

Step 6: Verify refreshing the documentation

- Commit some changes to your documentation
- Verify that the documentation website refreshes after your changes (can take few seconds or a minute)

Alternatives to GitHub Pages

- GitLab Pages and GitLab CI can create a very similar workflow. And have a guided procedure to add pages to your repository (if gitlab pages are enabled)
- Read the Docs is the most common alternative to hosting in GitHub Pages.
- Sphinx builds HTML files (this is what static site generators do), and you can host them anywhere, for example your university’s web space or own web server.

Migrating your own documentation to Sphinx

- First convert your documentation to Markdown using Pandoc.
- Create a file `index.rst` which lists all other Markdown files and provides the table of contents.
- Add a `conf.py` file. You can generate a starting point for `conf.py` and `index.rst` with `sphinx-quickstart`, or you can take the examples in this lesson as inspiration.
- Test building the documentation locally with `sphinx-build`.
- Once this works, follow the above steps to build and deploy to GitHub Pages or some other web space.

Keypoints

- Sphinx makes simple HTML (and more) files, so it is easy to find a place to host them.
- Github Pages + Github Actions provides a convenient way to make sites and host them on the web.

Hosting websites/homepages on GitHub Pages

Often we don't need more than a static website

You can host your personal homepage or group webpage or project website on GitHub using GitHub Pages. GitLab and Bitbucket also offer a very similar solution.

Unless you need user authentication or a sophisticated database behind your website, GitHub Pages can be a very nice alternative to running your own web servers. This is how all https://coderefinery.org material is hosted.

How to find the website URL

Here below, NAMESPACE can either be a username or an organizational account.

Personal homepage or organizational homepage

- Generated URL: https://**NAMESPACE**.github.io
- Generated from: https://github.com/**NAMESPACE**/**NAMESPACE**.github.io (main branch)

Project website

- Generated URL: https://**NAMESPACE**.github.io/**REPOSITORY**
- Generated from: https://github.com/**NAMESPACE**/**REPOSITORY** (gh-pages branch)

Exercise - Your own website on GitHub Pages

✍ GH-Pages-2: Host your own github page

- Deploy own website reusing a template:
  - Actually it's better to follow these instructions directly from this quickstart because the ones mentioned at pages.github.com do not actually work.
  - Follow the steps from GitHub Pages https://pages.github.com/. The documentation there is very good so there is no need for us to duplicate the screenshots.
  - Select "Project site".
  - Select "Choose a theme".
  - Follow the instructions on https://pages.github.com/.
  - The following one actually works
  - Browse your page on https://**USERNAME**.github.io/**REPOSITORY** (adjust "USERNAME" and "REPOSITORY").
- Make a change to the repository after the webpage has been deployed for the first time.
- Please wait few minutes and then verify that the change shows up on the website.

Real-life examples

- CodeRefinery website (built using Zola)
  - Source
  - Result: https://coderefinery.org/lessons/core/
- This lesson (built using Sphinx and MyST and sphinx-lesson)
  - Source
  - Result: this page

Note

- You can use HTML directly or another static site generator if you prefer to not use the default Jekyll.
- It is no problem to use a custom domain instead of `*.github.io`.

Summary

❓ Questions

- What recommendations can we take home?

There is not the one right way: it is always a balance

Jupyter notebooks can be good documentation for scripts

- For simple scripts and post-processing, Jupyter notebooks can form a nice self-documenting pipeline.
- They can be a nice way to accompany a paper that analyzed some data.

READMEs or Sphinx?

- For smaller projects READMEs can be absolutely enough.
- If the code is closed-source (and hence nobody can see the README), you probably prefer Sphinx (or similar).
- If you need math equations, Sphinx might be a good fit.

How to make sure that code changes come together with documentation changes?

- Make documentation part of your code review.

Read the Docs or GitHub pages or both?

- GitHub pages typically serves one version (one branch). However, it is possible to build several or all branches as part of a workflow.

- Read the Docs can serve several versions (several branches/tags) at the same time.
- Some projects use both.

Consider making your development tutorial-driven

- Writing documentation is as important as writing software.
- Focus on how you use the software.
- If there is no tutorial on it, the feature “doesn’t exist”.
- Don’t keep tutorial in sync with code, keep code in sync with tutorial - change the tutorial first.
- Read more in this fantastic slide-deck about tutorial-driven development.

List of exercises

Full list

This is a list of all exercises and solutions in this lesson, mainly as a reference for helpers and instructors. This list is automatically generated from all of the other pages in the lesson. Any single teaching event will probably cover only a subset of these, depending on their interests.

Instructor guide

Why we teach this lesson

Everyone should document their code, even if they’re working alone.

These are the main points:

- Code documentation has to be versionnable and branchable
- Code documentation should be tracked together with the source code
- README is often enough

Please do not skim over the two above points. Please take few minutes to explain why documentation (sources) should be tracked together with the source code. Please discuss this aspect with workshop participants and connect it to **reproducibility**. This is for me (Radovan) the most important take-home message.

Specific motivations:

- Code documentation becomes quickly unmanageable if not part of the source code.
- It helps people to quickly use your code thus reducing the time spent to explain over and again to new users.
- It helps people to collaborate.
- It improves the design of your code.

Intended learning outcomes

By the end of this lesson, learners should:

- Understand the importance of writing code documentation together with the source code
- Know what makes a good documentation
- Learn what tools can be used for writing documentation
- Be able to motivate a balanced decision: sometimes READMEs are absolutely enough

Timing

As an instructor you should prepare all bullet points but do not go through each bullet point in detail. Only highlight the main points and rather give time for a discussion. Leave details for a later lecture for those who want to find out more. If you go through each bullet point in detail, the motivation can easily take up 30 minutes and you will run out of time.

The lesson does not fit into 1.5 hours if you go through everything. Optimize for discussions and prepare well to be able to jump over bullet points which can be left for a later lecture. Some sections can be skipped if needed (see below). However, we recommend to have a discussion with your learners to make them aware of what the training material contains.

- Do not insist on practicing Markdown or RST syntax.
- The section *Rendering (LaTeX) math equations* may be optional if your attendees do not have to deal with equations.
- In the GitHub Pages episode, the goal is not anymore to write code documentation but to show how to build project website with GitHub. If time is tight, the GitHub pages episode can be skipped or can be done as demonstration instead of exercise.

Detailed schedule

- 09:00 - 09:10 Motivation and tools
  - create a wishlist in collaborative notes
- 09:10 - 09:20 Writing good README files
  - brief discussion
- 09:20 - 09:40 **Exercises**: README-1, README-2, README-3 (choose one or multiple)
- 09:40 - 10:00 Sphinx and Markdown: Sphinx-1 as type along
- 10:00 - 10:10 Break
- 10:10 - 10:40 **Exercises**, Sphinx-2, Sphinx-3, GH-Pages-1
- 10:40 - 11:00 Discussion, GH Pages, Summary

Place this lesson towards the end of the workshop

Reason is that with collaborative Git we can create more interesting documentation exercises. Currently there are some elements of forking and pushing and this is only really introduced on day two. We have tried this lesson on day one and it felt too early and disconnected/abrupt. It works best after the reproducibility lesson since we then reuse the example and it feels familiar.

Troubleshooting

Character encoding issues

Can arise when using non-utf8 characters in `conf.py`. Diagnose this with `file -i conf.py` and `locale`.

Live better than reading the website material

It is better to demonstrate the commands live and type-along. Ideally connecting to examples discussed earlier.

In online workshops most of the type-along becomes group exercise work where groups can share screen and discuss.

Field reports

2022 September

We were pressed for time (we started 5-10 minutes late, relative to the schedule below), so we made most of the first lessons fast. In the schedule below, note that we had the first 10 minutes for "Motivation" *and* "Popular tools", which we didn't fully realize so that put us even further behind. Doing these introduction parts quickly was hard but was probably worth it since we had plenty of time in the end. For the "tools", one person summarized the point of each section on the page quickly. The README episode was done quickly, we basically skipped the exercises to get to Sphinx, and this put us back on schedule.

For Sphinx, we did it a lot like you see in the schedule: first exercise (the basic setup) was type-along, but it was a bit too much to do in the 10 minutes we had allotted (we typed too fast). But, people then had a nice long time to make it up and do everything. It seemed to work well. The GitHub pages deployment could then be done as a nice, slow demo, and we had plenty of time to ask questions.

Overall, I think this was the right track, but we could have practiced doing the first parts even faster, and warned people that we focus on the Sphinx exercises.

Credit and license

This material is provided by CodeRefinery under the licenses stated below.

Website template

The website template is maintained by CodeRefinery and rendered with sphinx-lesson: structured lessons with Sphinx.

Instructional material

All CodeRefinery instructional material is made available under the Creative Commons Attribution license (CC-BY-4.0). The following is a human-readable summary of (and not a substitute for) the full legal text of the CC-BY-4.0 license.

You are free:

- to **Share** - copy and redistribute the material in any medium or format
- to **Adapt** - remix, transform, and build upon the material

for any purpose, even commercially. The licensor cannot revoke these freedoms as long as you follow these license terms:

- **Attribution** - You must give appropriate credit (mentioning that your work is derived from work that is Copyright (c) CodeRefinery and, where practical, linking to https://coderefinery.org, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

No additional restrictions - You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits. With the understanding that:

- You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation.
- No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as publicity, privacy, or moral rights may limit how you use the material.