Writing Effective Documentation - CSS Bootcamp

Will Styler

This document is just a summary of Will's take on how to write documentation that's worth a damn.

1 Writing Good Documentation

1.1 What kinds of things to include in documenting a research procedure

When writing good documentation, it's valuable to include a variety of *kinds* of documentation. A few worthwhile considerations are to include:

- Overviews
  - "This chunk of code is written to do the following ten things, in order"
  - "This experimental protocol will generate, for each participant, a set of eyetracking trials (Section 2), an acoustic recording (Section 3), an airflow recording (Section 4), and a sociolinguistic interview (Section 5)"
- Definitions
  - "Host Laptop - The Black Dell Laptop which comes with the Eye-Tracker"
- Checklists
  - "These are the necessary preparations you'll need to make in order to accomplish this task"
  - Include if statements for common problems, "You should see "Desktop Remote Monocular 25mm" in the upper right corner of the screen. If not, see Section 7.3 for instructions to repair"
- Walkthroughs
  - Step-by-step, granular walkthroughs of difficult or complicated analysis or data collection steps.
  - Write it such that somebody who had never done the task would have a fighting chance of accomplishing it
- Pitfalls
  - "It makes a lot of sense at first to do this thing, but if you do, you enter a world of suffering unlike anything you can imagine"
  - "If you turn the control computer on before the eyetracker, it will not be able to find the eyetracker"
- Verbal scripts
  - If you're working with humans, don't hesitate to offer scripted responses to common questions or situations for inexperienced experimenters. "In this case, say something like…"
- Troubleshooting
  - Watch somebody else do the tasks, and make note of areas where they struggle or problems they face
  - Explain what to do for (or at least the meaning of) errors and error messages
  - Give hints or solutions for anything you needed time to solve that might come up again
- Version History
  - This helps people 'catch up' on what they've missed
- Metadata
  - If researchers should be collecting metadata (e.g. URL, date of capture, etc), what are they

storing and where?
- Screenshots
  - If you're using an interface, explain where to click!
- What else?

1.2 What kinds of things to include when documenting code

When documenting code, many of the same ideas above still make sense. But a few specific things are worth noting:

- "Small but critical" bits of code
  - "This is just one line, but without this, we can't do any of the position calculation"
- Notes of fragility
  - "This is why the process will fail if the input value is negative"
- Pitfalls for future coders
  - Seemingly innocuous 'optimizations' of your code which cause everything to fail
  - "This feels unnecessary, but it turns out that…"
- Simple descriptive statements about what lines (or blocks) of code are doing
  - "This next chunk sanitizes the names to lowercase, removes titles, and stores them in a dictionary"

1.3 What kinds of things to include when documenting an analysis

When writing a Jupyter Notebook or writing documentation of your analysis for future release or public analysis, make sure to include:

- Explicit descriptions of each variable in the dataframe and their units (where relevant)
  - "`a1p0_comp` is a measurement of A1 (the harmonic under F1) minus P0 (here, the highest of H1 or H2) using the compensation function described in Chen 1997"
- Version numbers for critical packages
- What a file contains, each time you read one in
  - "rawdata.csv contains a dataframe which was generated from our airflow capture script containing all participants"
- Highlight the *most important* elements of an analysis
  - "The key takeaway from the graph below is…"
  - "The number we care about here is…"
- **Do not copy and paste figures, tables, and numbers**
  - Save figures, variables, and tables to files and import them

2 How to create documentation

2.1 Don't use Word

Seriously, **just don't**. Documents have patterns of formatting which should be consistent and modifiable easily across the whole document, rather than needing to be specified for *every single heading*. You want to be in charge of things like font type, indentation, lists, and more, rather than a 'helpful wizard' fighting you every step of the way. You don't want adding a small image to mess up your entire document.

And most of all, you don't want to be in charge of things like numbering sections, managing references, generating tables of contents, adhering to style guides, and more.

Word makes sense for people who don't code and want to 'make these letters big and those ones small and then print it and give it to their grandchildren to web it onto the facebook'. You are past this. Be past this. Your time is worth more than doing things in Word.

2.2 Markdown

Markdown is an excellent formatting language for plaintext documents originally developed by John Gruber and Aaron Swartz in the early 2000s.

It focuses on using ASCII symbols within plaintext documents to specify formatting. There are many 'flavors' of Markdown, and many syntax guides out there for it. Most use a set of syntax basics derived from this with particular extensions

Markdown allows headers (with levels denominated with the number of # symbols), allows *italics*, **bold**, `code blocks`, and more

- Bulleted lists work
- Bulleted lists work
- Bulleted lists work

1. So do numbered lists
2. You don't have to actually change the numbers
3. It'll work anyways

> You can make blockquotes of arbitrary length

```
echo "You can also make code blocks"
```

With Markdown, you can also intersperse bits of HTML to insert links, detailed images with CSS (cascading style sheets), or otherwise. You can also use HTML to make tables if you're exporting to HTML or slides, or use the native Markdown table format (which is a bit awkward, but which converts to LaTeX cleanly). `https://tablesgenerator.com/` is a nice, free tool for swapping formats of a given table. Here's a sample of markdown code for a table:

```
|              | Frog           | Dog            | Rabbit        | Velociraptor   |
|--------------|----------------|----------------|---------------|----------------|
| Frog         | Frog           | Drog           | Frabbit       | Velociribbeter |
| Dog          | Drog           | Dog            | Roggo         | Veldogciraptor |
| Rabbit       | Frabbit        | Roggo          | Rabbit        | Velocirabbitor |
| Velociraptor | Velociribbeter | Veldogciraptor | Velocirabbitor| Velociraptor   |
```

If you're using a Markdown program which supports converting to LaTeX, you can even include chunks of LaTeX. This allows you to use BibTeX references, typeset math, and more.

$\int_0^\infty \mathrm{e}^{-x}\,\mathrm{d}x$

$$\sum_{i=1}^{10} t_i$$

Markdown is a really, really effective way to generate large amounts of formatted text without messing around with awkward GUI editors. It's faster than writing raw LaTeX (**italics** rather than *italics*), it's readily understood by many programs and interfaces (e.g. github, Discord, obsidian, pandoc, reveal.js), and it's very fluid, allowing you to 'just write' and let the formatting take care of itself.

2.3 Generating Documents from Markdown

There exist roughly as many Markdown parsers and renderers as grains of sand on your average beach. Most of them are fine. Some of them support different standards or flavors of markdown. Github has one built in, which isn't bad.

The best of them is Pandoc. It turns Markdown of several flavors into HTML, EPUB, LaTeX, PDF (via TeX), reveal.js and other slide standards, Jupyter notebooks, Microsoft Word (if you hate yourself), and more. It is free and GPL licensed. It is my favorite software. I've used Pandoc to write my dissertation, to write several articles in journals, to write and maintain my website, my lecture slides, and more.

Using Pandoc is very easy: You call pandoc with an input file and output filename. But it's powerful. You can give it stylesheets for HTML and templates for LaTeX. You can (and should!) incorporate it into shell scripts, call it within your favorite text editor, and use it to do nearly everything.

The below commands turn `doc.md`, a markdown document, into a PDF and HTML using LaTeX:

```
# Markdown to PDF
pandoc -s -o doc.pdf doc.md
# Markdown to HTML
pandoc -s -o doc.html doc.md
```

I use the below code, which includes a template for UCSD letterhead, to generate PDF letters of recommendation:

```
pandoc -s fname.md --pdf-engine=xelatex --from markdown --template $HOME/Documents/bin/pandoc/temp
```

And this code generated the HTML for this document, using a custom CSS (cascading stylesheets) document:

```
pandoc -s fname.md -H ~/Documents/bin/pandoc.css -o fname.html
```

```
cd ~/Documents/docs
find ~/Documents/text/docs -name "*.md" | while read ffname; do
    filename=$(basename -- "$ffname")
    filedir=$(dirname -- "$myfile")
    fn="${filename%.*}"
    if [[ $ffname -nt ~/Documents/docs/$fn.html ]]; then
        echo $fn
            # Remove the flag from the file, and adjust the path for future compilation
            ffname="/tmp/edfile.md"
            # Render the pdf
            /opt/homebrew/bin/pandoc -s "$ffname" --pdf-engine=xelatex --from markdown --template
        fi
        # Render the HTML
        /opt/homebrew/bin/pandoc -s "$ffname" -H ~/Documents/bin/pandoc.css -o "$fn".html
```

```
    fi
done
```

This, precisely, is how this document came to be. (Note that LaTeX does not wrap code blocks by default, read this bit in the HTML)

2.4 Jupyter exporting

You can actually export Jupyter Notebooks to HTML, LaTeX, Markdown, Reveal.js using the 'File -> Save and Export Notebook as…' command.

This allows your final 'paper' to effectively *be* your analysis, and allows you to directly generate tables and plots *into* your document. This is incredibly powerful.