Collaborative Guide: Quarto, Git, and Project Work

Contributors from [Course Name]

Table of contents

1 Introduction

Welcome to our class's collaborative guide! This living document was created by students to collect useful practices, workflows, and tips for writing Quarto documents, collaborating with Git, and organizing projects.

It will evolve as we work on it together, and serve as a reference for your group projects.

---

2 Authoring with Quarto

2.1 Anatomy of a Quarto Report

2.1.1 Metadata Header

The *metadata header* is the beginning of a Quarto document. It displays information like author or date and tells Quarto how the document should be configurated. As an example, you can look at the top of this document in the in its qmd file version, *not in the rendered version.* There are a lot of possibilities to customize a Quarto document in the *metadata header.* In this document, we will be looking at the most commonly used configurations. If you want to see all the options that are available, you can visit the [official Quarto website] (https://quarto.org/docs/output-formats/html-basics.html).

The formatting applies only to the current document.

Use triple dashes to delineate the block:

```
---
some metadata
---
```

Most commonly, the title block contains the **title**, **subtitle**, **authors**, **date**, **doi** and the **abstract** of the article.

Other important options:

```
format: html / pdf / docx
```

This option defines the format of the final product.

```
date: YYYY-MM-DD
```

Sets the publication date (find supported date formats here)

```
execute:
  echo: true / false
  eval: true / false
  warning: true / false ...
```

Controls how code cells are executed:

- **echo**: shows the code in the output if **true**
- **eval**: executes the code if **true**
- **warning**: includes warnings in the output if **true**
- **message**: includes messages from code if **true**
- **error**: displays error messages in the output if **true**

2.1.2 Sections

To better structure and improve the appearance of the document, a Quarto report can be divided into sections with headings. If you want to number the headings or add a table of contents *toc*, which serves as an interactive section menu, you can do so in the *metadata header*. For example, in this document, we use the following configurations:

```
  toc: true
  toc-depth: 3
  number-sections: true
```

You can also add the option `number-depth: 4` to set the deepest level of heading that should be numbered and you can use `### Example Heading {.unnumbered}` to exclude a heading from numbering.

2.2 Document Sections and Header Levels

In Quarto (and Markdown), section headers are defined using the hash symbol `#`. The number of hash symbols indicates the depth of the section:

- `#` creates a top-level (H1) section
- `##` creates a second-level (H2) subsection
- `###` creates a third-level (H3) sub-subsection
  … and so on, up to six levels (`######`)

This system allows the document to be structured hierarchically, which improves both **readability** and **navigability**. Header levels are also used to build features like the **table of contents** (`toc: true`) and for **numbering sections** (`number-sections: true` in YAML).

Always include a space after the hash symbols. For example:

```
# Introduction
## Methodology
### Data Collection
#### Survey Design
```

```
##### Question Format
###### Notes
```

To ensure consistency, avoid skipping levels (e.g., going directly from `#` to `###`) unless intentionally needed for visual clarity or document structure.

For more, see Quarto documentation on headers.

2.2.1 Layout Options

Depending on what format you use, there are different layout options available. For a **HTML document**, like this one, there are the following options to configurate the document:

2.2.1.1 Page Layout

1. `page-layout: article` - default layout with single centered content column
2. `page-layout: full` - expanding content to full browser width
3. `page-layout: sidebar` - adding sidebar for navigation

2.2.1.2 Themes

There are 25 themes from [*Bootswatch*] (https://bootswatch.com/) available in Quarto. These themes can further be customized in the metadata header. If you want to check out all the theme options and configurations, you can go to [Quarto - HTML themes] (https://quarto.org/docs/output-formats/html-themes.html).

2.2.1.3 Table of Contents

If you set `toc: true` in the metadata header, Quarto creates a table of contents. This table can further be configurated with options such as: `toc-depth: 4` - specify number of section/heading levels to include in the toc, *default is set to 3* `toc-expand: 3` - specify how much of toc to show initially, *default is set to 1* `toc-title: Example` - customize title of toc `### Example Heading {.unnumbered .unlisted}` - exclude heading from toc `toc-location: left` - set location of toc For more information, you can visit [Quarto - HTML basics] https://quarto.org/docs/output-formats/html-basics.html and scroll to the section **Table of Contents**.

2.2.1.4 Other Layout Options

There are many other layout options available to configurate spacing, typography, character width, etc. For all the options you can visit [Quarto - HTML basics] https://quarto.org/docs/output-formats/html-basics.html.

2.3 Basic Formatting & Markdown

2.3.1 Lists

2.3.1.1 Unordered Lists

1. Use `-`, `+`, or `*` followed by a space (choosing one style is enough)
2. To create a **nested list**, indent (with 2 or 4 spaces) the sublist items under the parent item

Example

```
* unordered list
    + sub-item 1
    + sub-item 2
        - sub-sub-item 1
```

Rendered as

- unordered list
  - sub-item 1
  - sub-item 2
    - sub-sub-item 1

2.3.1.2 Ordered Lists

1. Ordered lists work just like unordered lists — just replace the bullets (`-`, `+`, or `*`) followed by a period `.`

2. In a nested ordered list, formats like `A.` or `i)`can be recognized if they are properly indented

   - Note: it's also better to use **two spaces after the marker** to ensure the item is parsed as part of the list

Example

```
1. ordered list
2. item 2
    i) sub-item 1
         A.  sub-sub-item 1
```

Rendered as

1. ordered list
2. item 2
   i) sub-item 1
      A. sub-sub-item 1

2.3.1.3 Mix Ordered and Unordered lists

It's also possible to mix ordered and unordered lists.

Example

```
1. Main item one
   - Sub item A
   - Sub item B
2. Main item two
   * Sub item C
     1. Sub-sub item (ordered)
     2. Another sub-sub item
```

Rendered as

1. Main item one
    - Sub item A

    - Sub item B

2. Main item two
    - Sub item C
        1. Sub-sub item (ordered)

        2. Another sub-sub item

For more formatting options, see the Quarto Markdown Basics documentation on Lists.

2.3.2 Emphasis and Text Styles

In Markdown and Quarto, text formatting is done by surrounding words or phrases with special symbols. This method makes formatting quick, consistent, and readable even in plain text. It also helps writers focus on content without relying on visual formatting tools.

The most common symbols are asterisks `*` and tildes `~`. These symbols are placed directly before and after the text that should be formatted.

2.3.2.1 Bold

Bold text is created by placing double asterisks around a word or phrase.

```
**This text is bold**
```

This text is bold

2.3.2.2 Italic

Italic text is created by placing single asterisks around the text. Italic can be used for emphasis, thoughts, or non-English terms.

```
*This text is italic*
```

This text is italic

2.3.2.3 Strikethrough

Strikethrough text is created using double tildes. This format is useful for marking deletions or outdated information.

```
~~This text is crossed out~~
```

~~This text is crossed out~~

2.3.2.4 Combining Styles

It is also possible to combine multiple formatting styles. For example:

```
***bold and italic***
```

bold and italic

```
**~~bold and strikethrough~~**
```

~~bold and strikethrough~~

```
*~~italic and strikethrough~~*
```

~~italic and strikethrough~~

When combining symbols, the order matters: the outermost symbols should match (for example, `**~~text~~**` is correct, while `*~text~*` is not).
Also make sure all opening symbols are properly closed — otherwise, the formatting might not render as expected.

These combinations allow flexibility in tone:
- Use `***text***` when something is **important and personal**
- Use `**~~text~~**` when something was once important but is now removed
- Use `*~~text~~*` when marking a tentative or soft correction

For more formatting options, see the Quarto Markdown Basics documentation.

2.3.3 Inserting External Links

External links can be added verbatim by copying said link into the document and bracketing it with < > like this:

```
<https://en.wikipedia.org/wiki/Bumblebee>
```

If one does not want the link to appear in full, one can add another string in square brackets and the link, this time in regular brackets, to the right as exemplified below. The output will then display the link in the square brackets and clickin on it will redirect the reader to the landing page of the deposited link - here a page on Bumblebees.

```
[Bumblebees](https://en.wikipedia.org/wiki/Bumblebee)
```

2.3.4 Linking to Document-Internal Sections

For this, the header one wishes to link to must be augmented with a tag of the following syntax (including the squiggly brackets). The NAME part can be adapted by the author. The tag is added to the header's right boundary.

```
{#sec-NAME}
```

To later reference said header, the same tag but with an at sign instead of the octothorpe must be added. This can then be integrated into prose text. The NAME part will be displayed.

```
As discussed in @sec-NAME,...
```

2.4 Code Cells and Quarto Options

2.5 Images and Plots

2.5.1 How to reference images in text

2.6 Tables

2.6.1 Quarto Markdown Table

Markdown Source and Display

```
| Default  | Left  | Right  | Center  |
|---------|:-----|------:|:------:|
| 12       | 12    |     12 |   12    |
| 123      | 123   |    123 |  123    |
| 1        | 1     |      1 |   1     |

: Demonstration of pipe table syntax {#tbl-syntax .striped .hover}
```

Table 1: Demonstration of pipe table syntax

Default	Left	Right	Center
12	12	12	12
123	123	123	123
1	1	1	1

- **Basic syntax**
  - Pipe tables use `|` to indicate column boundary, at least three dashes `---` to define the header row and a line starting with `:` after the table to add a caption.
- **A header row is required**
  - You can simulate a "headerless" table by using an empty header (e.g., `| | |`).
- **Leading and trailing pipes `|` are optional but pipes between all columns are necessary**
  - Example: `fruit|price` is valid.
- **Alignment in source code is just for readability**
  - Pipe characters need not to be vertically aligned. Uneven tables still render correctly.
- **Alignment type is set using colons `:` in the separator line**
  - Three alignment types are shown above: *Left (Default)*; *Right*; *Center.*
- **Relative column widths can be influenced by the number of dashes `---` in the separator line**
  - More dashes = wider column.
  - Example: `------|---` will make the first column approximately 2/3 width, the second 1/3.
  - Alternative: `: Table Caption {tbl-colwidths="[67,33]"}`
- **Block elements are not allowed in table cells**
  - No paragraphs, lists, or multi-line content inside a cell.

- **Add the Bootstrap classes after the table caption**
  - Example: `:Table Caption {.hover .striped}`
- **Cross-reference with `:Table Caption {#tbl-label}` and `@tbl-label`**
  - Example: See `@tbl-syntax` -> See Table 1

For more details, see the Quarto documentation on tables.

2.6.2 Creating Tables Programmatically with knitr::kable()

`knitr::kable()` is a R function used to format data frames or matrices as clean, publication-ready tables in Markdown, HTML, or LaTeX outputs.

Syntax

```r
kable(x, format, digits = getOption("digits"), row.names = NA,
      col.names = NA, align, caption = opts_current$get("tab.cap"),
      label = NULL, format.args = list(), escape = TRUE, ...)
```

Argument	Type	Description
`x`	data.frame / matrix	The data to be rendered as a table. Required.
`format`	character	Output format: `"markdown"`, `"html"`, `"latex"`, etc. Usually auto-detected.
`digits`	integer	Number of decimal places to display. Defaults to `getOption("digits")`.
`row.names`	logical / `NA`	Whether to include row names: `TRUE`, `FALSE`, or `NA` for auto-detect.
`col.names`	character vector	Custom column names for the table.
`align`	character vector	Column alignment: `"l"` (left), `"c"` (center), `"r"` (right).
`caption`	character	Table caption shown above the table.
`label`	character	Label used for cross-referencing (e.g., `@tbl-summary`).
`format.args`	list	Additional arguments passed to `format()`.
`escape`	logical	Escape special characters like `_`, `&`. Use `FALSE` to allow raw HTML/LaTeX.
`...`	—	Additional arguments passed to formatting methods.

Example

```r
library(knitr)
# Create a data frame similar in structure to the markdown example
data <- data.frame(
  Default = c("12", "123", "1"),
  Left = c("12", "123", "1"),
  Right = c("12", "123", "1"),
  Center = c("12", "123", "1")
)
# Generate the table using knitr::kable()
# Set column alignment: left, left, right, and center
# Add a caption to the table
kable(
  data,
  align = c("l", "l", "r", "c"),
  caption = "Table 1. Demonstration of knitr::kable() table with alignment"
)
```

Table 3: Table 1. Demonstration of knitr::kable() table with alignment

Default	Left	Right	Center
12	12	12	12
123	123	123	123
1	1	1	1

For more details on using `knitr::kable()`, see the R Markdown Cookbook – Section 6.5.

2.6.3 Comparison: Quarto Markdown Table vs knitr::kable()

Feature	Quarto Markdown Table	`knitr::kable()` Table
Syntax Style	Pure Markdown	R code chunk using `kable()`
Data Source	Written manually	R data frame or matrix
Captions Support	Yes (with `: ...` line after the table)	Yes (`caption = "..."`)
Auto Numbering	Yes	Yes
Cross-referencing	Yes (e.g., `@tbl-id`)	Yes (with `label = "tbl-id"`)
Output Formats	HTML, PDF, DOCX	HTML, PDF, DOCX
Styling (basic)	Manual (using colons `:` for alignment)	Basic via `align = c("l", "r", ...)`
Styling (advanced)	Limited	Supports `kableExtra`, `booktabs`, etc.
Dynamic Content	No	Yes – generated from live R data

Feature	Quarto Markdown Table	`knitr::kable()` Table
Best For	Simple static tables or syntax demos	Data-driven tables, dynamic reports

2.7 Citations and Bibliography

2.8 Citations and Bibliography

Citations in Quarto are managed using external bibliography files, typically with a `.bib` extension (in BibTeX format). This method keeps references organized and separates citation data from the main text.

2.8.1 Step 1: Create a `.bib` file

A `.bib` file contains citation entries such as `@book`, `@article`, or `@misc`. Each entry stores metadata like author, title, and year. This format allows Quarto to format and render citations consistently.

Example entry:

```
@book{manning1999,
  author    = {Christopher D. Manning and Hinrich Schütze},
  title     = {Foundations of Statistical Natural Language Processing},
  year      = {1999},
  publisher = {MIT Press}
}
```

It is recommended to name the file `references.bib` and place it in the root of the Quarto project folder.

Tip: Tools like Zotero (with the Better BibTeX plugin) can help manage and export `.bib` files easily.

2.8.2 Step 2: Link the `.bib` file in the YAML header

Add the following line to the YAML metadata block at the top of the `.qmd` file:

```
bibliography: references.bib
```

We can also include multiple `.bib` files if needed:

```
bibliography: [references.bib, extra.bib]
```

This tells Quarto where to find the citation data.

2.8.3 Step 3: Add inline citations

To cite a source in the text, use `@key`, where `key` is the BibTeX ID from your `.bib` file.

There are two common ways to format inline citations:

- `[@manning1999]` → renders as: (Manning and Schütze 1999)
  Used when the author name should appear in the citation.
- `[-@manning1999]` → renders as: (1999)
  Used when the author name is already mentioned in the sentence.

For example:

See [@manning1999]. Published in [-@manning1999].

2.8.4 Step 4: Generate the bibliography section

At the end of document, include a section where the bibliography will be rendered:

```
## References
```

Quarto will automatically generate this list using the sources you've cited. If no references section is added manually, it will appear at the end of the document by default.

2.8.5 Step 5: Change the citation style (Optional)

Citation styles can be changed using the `csl:` option in the YAML block. For example:

```
csl: apa.csl
```

CSL (Citation Style Language) files define formatting rules (e.g., APA, MLA, Chicago). Thousands of styles can be downloaded from the Zotero Style Repository.

2.8.6 Further Reading and Resources

- Quarto Documentation – Citations
- Zotero Style Repository

```
# Git and Collaborative Workflows

## Branching and Merge Requests
<!-- TODOs: branch creation, MR checklists, communication -->

## GitHub Desktop
<!-- TODOs: clone, commit, push, sync conflicts -->


## Best Practices
<!-- TODOs: shared tips, helpful links -->

---

# Project Organization

## Project Structure
```

~~~
Use the following basic template for your project

```md
project_name/ # <1>
  data/ # <2>
  R/ # <3>
  outputs/ # <4>
     images/
     tables/
     .../
  .gitignore # <5>
  _quarto.yml # <6>
  report.qmd # <7>
  worklog.md # <8>
  README.md # <9>
  project_name.Rproj # <10>
~~~

① Project name
② Original data used by your project
③ R scripts for functions and utilities, as well as setup code
④ All outputs are stored here, with nested folders for images, data, etc. as needed
⑤ Git ignore list (configured to skip temporary files)
⑥ Quarto configuration file — defines output format, theme, and other global settings
⑦ The main project report (Quarto Markdown)
⑧ Worklog tracking project member contributions
⑨ Project description and instructions
⑩ RStudio Project file (if using)

2.9 General Guidelines

- Use RStudio Projects (or similar tool) to organize your project
- Use GitHub issues, branches, and structured communication channels to organize your work
  - Establish project roles and responsibilities
  - Maintain the worklog to track your contributions
  - Have one `main` branch to represent the latest "clean" work you did
  - Use a single meta-issue to track project progress
- Use descriptive, consistent file names
  - Use `snake_case`, avoid spaces or other special characters in file names
  - Use numerical prefixes (e.g. `01_load_data.R`, `02_clean_data.R`) if scripts need to be executed in particular order
  - Use `.gitignore` to exclude temporary files that do not need to be versioned
  - Add `.Rhistory`, `.Rproj.user`, and large intermediate results (e.g. model caches) to `.gitignore`
- Organize your code
  - Prefer short, focused chunks with clear purpose

    - Put large analyses and common utilities into separate scripts (to be stored in `R/`)
- Never modify raw data files directly
    - Use spatial separation for generated files (i.e. `outputs/`)
- Document everything
    - Create a detailed README with project overview, dependencies, and instructions
    - Comment your code thoroughly
    - Document data sources (version, license, how the data was fetched, etc.)
    - Include a data dictionary (brief structured explanation of all data you use)
    - Document decisions and assumptions
- Aim for reproducibility
    - Document the packages you use (and how you use them)
    - Always use project-relative file paths (e.g. `data/data.csv` instead of `/Users/taras/Projects/my_project/data`)
    - Make sure that the code runs on all your machines

---

3 Coding Style Guide

Good coding style makes your code easier to read, share, and revisit later. The goal is clarity and consistency. This section outlines some simple style rules to follow in our group projects.

3.1 Coding Style

Structure

- Use **two-space indentation** — this keeps things compact and readable
- Use the **pipe operator (`|>`)** to break operations into clear steps
- Every **dplyr predicate** (like `filter()`, `mutate()`) should start on a **new line**.
- For long function calls, put each argument on a separate line, with indentation.

Variables

- All names should be self-explanatory, be as verbose as necessary (e.g. `penguins_stats_summary` instead of `pgs`)
- Use **snake_case** for all variable and function names (e.g., `penguin_data`, `filter_species()`)
- Use different variable names for different data

Code Chunks

- Chunks should be compact and focused — ideally under 10–15 lines
- Avoid "dead code" — don't include code that isn't used or explained
- Large, reusable functions or repeated logic should be outsourced to a shared script (e.g., `R/helpers.R`)

- Every chunk should fulfill a clear purpose, and its role should be obvious from the context or accompanying comments

Comments

- Use comments to explain why something is being done — not just what
- Avoid repeating the code in your comment
- Place comments above or beside logical steps to improve clarity
- Use comment markers (e.g. `# TODO:`, `# CHECK:`) to call attention to things that need to be done

3.2 Example

```
#| echo: true
#| eval: false
# survey Adelie penguins across the islands                                ①
# note: we need to make sure that we remove all empty (NA) entries
penguin_adelie_stats <- penguins |>                                        ②
  filter(                                                                  ③
    !is.na(bill_length_mm),
    !is.na(body_mass_g),
    species == "Adelie"
  ) |>                                                                     ④
  summarize(
    avg_bill_length = mean(bill_length_mm),
    # centroid_mass() calculates the centroid mass (see discussion),  ⑤
    # and is defined in R/centroids.R
    centroid_body_mass = centroid_mass(body_mass_g),
    .by = island
  )
```

① Clear comment, explains the purpose of the code
② Verbose, self-documenting variable name
③ Each function call on a separate line, grouped by consitent indentation
④ Use pipe operator to chain transformations
⑤ Custom function defined in an external script, with a guiding comment

---

4 Plotting Best Practices

---

5 AI Usage Guidelines

5.1 General Principles

- **Transparency**: Always disclose AI usage in your work
- **Responsibility**: You are responsible for all content, even if AI-assisted
- **Learning Focus**: Use AI as a tool to enhance learning, not replace it

5.2 Suggested Best Practices

- Use AI for brainstorming
- Ask AI to explain concepts you don't understand
- Have AI review your work for clarity and structure
- Avoid using AI to generate complete solutions without understanding

5.3 Further Resources

- UZH AI Guidelines

---

6 Multiformat Documents

You can generate output of the same document in all formats that are specified under format in the authoring section of the Quarto document. Upon rendering, an output file is generated in the first specified format, which is mostly html.

The Quarto documentation states this:

Clicking the Render button (or using the keyboard shortcut K) in RStudio will render the document to the first format listed in the YAML.

This, however, does not work in my installation of RStudio!

If you want files to be generated for several formats, you must do this on the RStudio command line. First, you need to install the quarto package. Use this command:

install.packages("quarto")

Then, after you have saved the Quarto qmd file, issue:

quarto render –to , <fmt2,…,

For example, to generate output for this document, enter:

quarto render living_document.qmd –to html,pdf,docx

The nice thing is that the rendered html file contains links to the other formats.

quarto::quarto_render("format_test.qmd", output_format = "all")

Or if you prefer using the terminal (or if you are a non R-useR) you can run the following command in the terminal,

quarto render format_test.qmd –to html,gfm

---

7 Extra Tips and Tricks

7.1 VS Code Setup for Quarto

- Install VS Code, R and Quarto.

- Open VS Code, go to Extensions and install **Quarto**.
- Install the required packages: knitr and rmarkdown. To do this, run the following commands:

  ```
  install.packages('knitr')
  install.packages('rmarkdown')
  ```

  You can run these commands on RStudio (installation instructions). Alternatively, you can set up an R terminal on VS Code. To do this, you will need another **extension**: **R**. Once you've installed R on VS Code, you can run an R terminal and install packages there.
- To preview your document, press the preview button (top right corner).
- To view recent changes, refresh the preview (refresh button in Quarto Preview tab).

---

8 Further Resources

Here are some useful official resources to explore:

- Quarto Documentation
- GitHub Docs: Branches
- GitHub Desktop Documentation
- Happy Git with R

Feel free to add more helpful links as we go!