

# Setting up a project

Best Practices for Writing Reproducible Code // part 1

# Research compendium

A research compendium is a collection of *all digital parts of a research project* including data, code, texts (...). The collection is created in such a way that reproducing all results is straightforward.

Source: *The Turing Way*



compendium

(Artwork by Scriberia for *The Turing Way*, CC-BY)

# Getting started

- Contain your project in a single recognizable folder
- Distinguish folder *types*, name them accordingly:
  - **Read-only**: data, metadata
  - **Human-generated**: code, paper, documentation
  - **Project-generated**: clean data, figures, models...
- Initialize a **README** file, document your project
- Choose a **license**
- Publish your project.

-

```
•
|-- CITATION
|-- README
|-- LICENSE
|-- requirements.txt
|-- data
|   |-- birds_count_table.csv
|-- doc
|   |-- notebook.md
|   |-- manuscript.md
|   |-- changelog.txt
|-- results
|   |-- summarized_results.csv
|-- src
|   |-- sightings_analysis.py
|   |-- runall.py
```

Wilson *et al.* (2017)

# Cookiecutter

You can set up a project template using a nifty tool called cookiecutter.

First, ensure you have cookiecutter installed:

```
pip install cookiecutter
```

or with conda:

```
conda install cookiecutter
```

Trouble with conda or pip? Check the instructions [here](#), or try the following alternatives:

- MacOS X with Homebrew:

```
brew install cookiecutter
```

- Debian/Ubuntu:

```
sudo apt-get install cookiecutter
```

# Cookiecutter



A command-line utility that creates projects from cookiecutters (project templates). e.g. creating a Python package project from a Python package project template.

*[cookiecutter.readthedocs.io](https://cookiecutter.readthedocs.io)*

There are MANY templates available for your purposes. **Take a look!**

We have designed a template based on **this paper by Wilson *et al.*** (2017). Go ahead and install it!

```
cookiecutter gh:bvreeede/good-enough-project
```

Answer the questions cookiecutter asks you, and browse the resulting project to see where your answers ended up.

# Backup option

Did cookiecutter not work for you? We have a backup option:

- Navigate to the location where you want to create your project.
- Then clone this template repository:

```
git clone https://github.com/bvreeede/good-enough-project-template.git
```

- We will initialize a git repository here. But for now: please remove the `.git` folder.
  - Use your GUI explorer, not your terminal for this step!
  - Ensure hidden files/folders are shown (on mac: press `cmd` + `shift` + `.` (dot)).
  - Delete the entire `.git` folder in your project folder.

# Your turn

- Place your project files in the right folder.
- Adjust paths in your code, and be sure to use relative (see next slide) paths!
- It is fine to have the main script (e.g. `main.py`) in the home folder!

.	
├── .gitignore	
├── CITATION.md	
├── LICENSE.md	
├── README.md	
├── requirements.txt	
├── bin	← Compiled and external code, ignored by git (PG)
│   └── external	← Any external source code, ignored by git (RO)
├── config	← Configuration files (HW)
├── data	← All project data, ignored by git
│   ├── processed	← The final, canonical data sets for modeling. (PG)
│   ├── raw	← The original, immutable data dump. (RO)
│   └── temp	← Intermediate data that has been transformed. (PG)
├── docs	← Documentation notebook for users (HW)
│   ├── manuscript	← Manuscript source, e.g., LaTeX, Markdown, etc. (HW)
│   └── reports	← Other project reports and notebooks (e.g. Jupyter, .Rmd) (HW)
├── results	
│   ├── figures	← Figures for the manuscript or reports (PG)
│   └── output	← Other output for the manuscript or reports (PG)
└── src	← Source code for this project (HW)

# A note on paths


- Your project should be transportable between computers.
- For this reason, you should use **relative paths** only: compare
  - `/Users/barbara/Dropbox/proteindomains/data/zincfinger.json`
  - `./data/zincfinger.json`
- `./` means: in this folder
- `../` means: one folder up



# Choosing a license

- Copyright is implicit; others cannot use your code without your permission.
- Licensing gives that permission, and its boundaries and conditions.
- Choosing a license early on means being aware of your license as the project proceeds (and not creating conflicts).
- There are over **80 OSI-approved licenses** (and **many, many** others) to choose from.

This is one I like to use:

 <p>bvreede/good-enough-project is licensed under the <b>MIT License</b></p> <p>A short and simple permissive license with conditions only requiring preservation of copyright and license notices. Licensed works, modifications, and larger works may be distributed under different terms and without source code.</p> <p>This is not legal advice. <a href="#">Learn more about repository licenses.</a></p>	<b>Permissions</b> <ul style="list-style-type: none"><li>✓ Commercial use</li><li>✓ Modification</li><li>✓ Distribution</li><li>✓ Private use</li></ul>	<b>Limitations</b> <ul style="list-style-type: none"><li>✗ Liability</li><li>✗ Warranty</li></ul>	<b>Conditions</b> <ul style="list-style-type: none"><li>ⓘ License and copyright notice</li></ul>
---	---	---	--

What is important to you? What does your lab use? **Choose your own license!**

# Publishing your project

Uh... Isn't 'publication' the thing you do... *at the end?*

No! Publishing your project at an early stage

- forces you to consider readability throughout
- minimizes the mess you have to deal with when you (finally) decide to publish
- allows collaboration and support
- facilitates sharing and re-use.

*But what if someone scoops my code! I'm a revolutionary, they will steal my ideas!*

If you are super paranoid, you can always opt for a private repository. It is your work & up to you. But consider the advantages!

# Publishing unpublished data

- If you have sensitive data...
  - Don't include your data in your software repository (that's not what they are for anyway).
  - Consider generating simulated data so your code can run regardless.
- And for all data:
  - Your data should be separate from your code!
  - If your code references your data, consider a config or metadata file for these references.

# Where do I publish?

## Living project: github

(or other social coding platform):

- synergistic with version control software git
- makes history public and accessible (*EEK!*)
- allows publication of different releases
- provides a platform for interaction and collaboration



## Archiving a release: zenodo

(or other stable repository, like the [OSF](#))

- **direct archiving supported** from github to zenodo
- this gives you a **doi** (digital object identifier): your code is citeable!

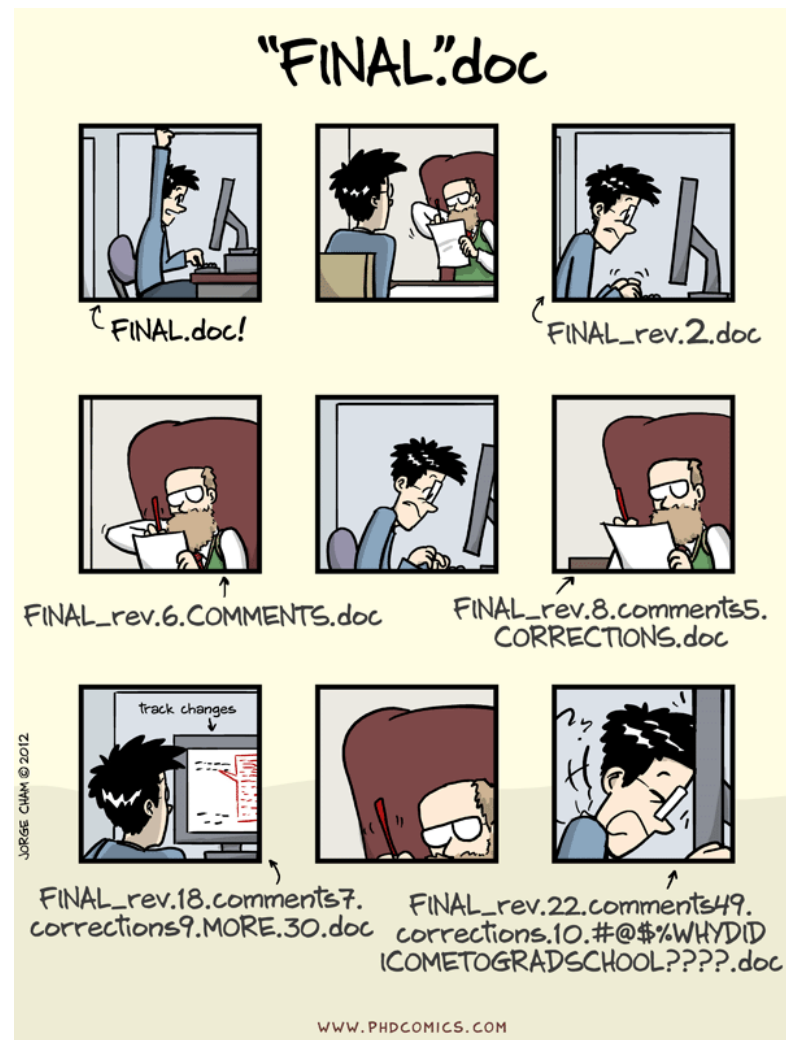


# Why do you need version control?

- It will help you manage ~~your code~~ most of your files (it is like track changes on steroids: it applies to all files in a folder).
- It allows you to trace back your steps: if something breaks, you can figure out what happened.
- NO MORE  
thesis\_final\_final\_SERIOUSLYFINAL.Rmd

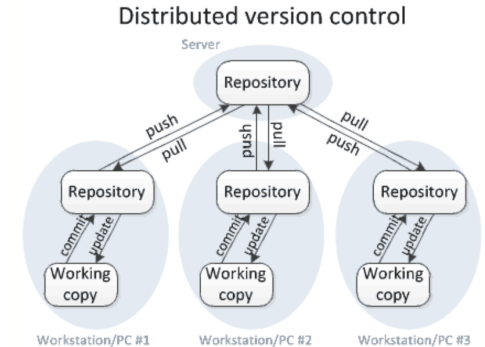
even better:

- a good version control system allows you to collaborate and share!
- a good version control system facilitates experimentation!



# What is git?

- Distributed Version Control system written by Linus Torvalds (of Linux fame)
- Allows you to log updates, branch your work (so you can experiment without losing the original!), and keep all backups, while efficiently using your storage
- Gives user a lot of control on what to track, and adds a narrative to changes ('commit comments')
- Current standard for code
- Open Source software written for the command line...
- ... but many GUI-clients exist nowadays, and most coding IDEs have built-in git.



# Your turn: starting with git

1. (You should have installed git by now! If you have not: [git-scm.com](https://git-scm.com).)
2. Navigate to your project folder in a terminal.

```
cd [path/to/project_folder]
```

3. Initiate a git repository in this location:

```
git init
```

4. Add all your files to the staging area:

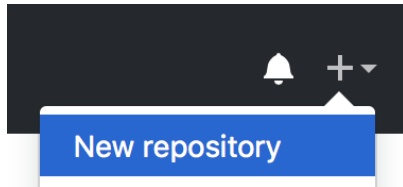
```
git add *
```

5. Commit all the files in the staging area to your repository:

```
git commit -m "First commit"
```

# Your turn: connecting to github

1. Go to your github account and add a new repository (click '+', then 'New repository'):



2. Fill out the information for your new repository. DO NOT initialize the repository yet!

Skip this step if you're importing an existing repository.

☐ **Initialize this repository with a README**

This will let you immediately clone the repository to your computer.

Add .gitignore: **None** ▼

Add a license: **None** ▼



3. Set the origin of your local repository to the URL of your repo on github:

```
git remote add origin git@github.com:[youraccount]/[yourrepo].git
```



# Your turn: pushing to github

You can now push the content of your local repository to the one on github:

```
git push -u origin master
```

Congrats, your local repository now has an online representation!



\\_\_\_ THANK YOU!

Take a look at your online repository. Who is the author of your commits? If it is not you, you can configure git to use your identity (make sure github knows this email address):

```
git config --global user.name "Your Name"  
git config --global user.email "your@email.com"
```

# Your workflow

1. Add the changed file(s) to the staging area, and commit the changes:

```
git add src/filename.py anotherfile.txt  
git add config/configfile.json  
git commit -m "My commit message"
```

OR

2. commit files directly, without staging:

```
git commit src/filename.py anotherfile.txt -m "My commit message"
```

NB: don't forget your commit message (try what happens if you do!).

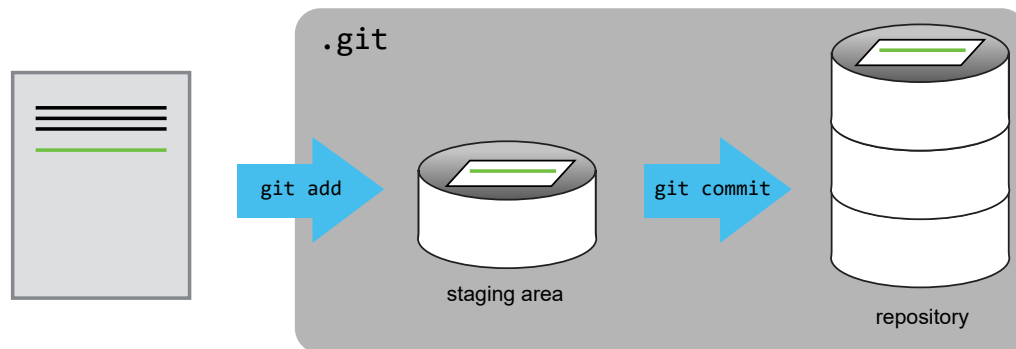


Image credit:  
Software Carpentries

# What else can I do with git?

Check the status of your repository:

```
git status
```

Check the log of your previous commits:

```
git log  
git log --oneline
```

What differences are there between your last commit and your workspace? (Or between two previous commits...)

```
git diff  
git diff HEAD~3
```

SO much more!

```
git --help
```

# .gitignore

The .gitignore file in your template contains files that **by definition** will not be tracked by git.

For example, if you do not want to track a file .DS\_Store (always present on my mac), you enter a line like this in your .gitignore file:

```
.DS_Store
```

Similarly, you can ensure all output in a folder will not be tracked:

```
results/
```

Or all files with a certain extension:

```
*.dat
```

*NB: There is a .gitkeep file in your template -- this does not do the opposite to .gitignore, but is instead used as a placeholder for folders: git does not track empty folders...*

# Your turn!

1. Continue moving your files into the file template.
2. Add, commit, and push all files you want to track! (Do you want to move a tracked file within a git repository? `git mv path/to/file.svg newpath/file.svg` and don't forget to commit!)
3. Are there (temporary) files you do not wish to track? Add them to the .gitignore file. Consider a .gitignore template for your language: [examples on this github repo](#).
4. Continue editing your code, and add/commit/push your changes. Can you do it from your IDE?
5. Experiment with editing and committing on github itself. You can then 'download' your code to your local repository using `git pull`.
6. What happens if you edit the same file online and locally, and try to push/pull?

# Enjoy, and git responsibly!

- Commits should be atomic: comprehensive 'units' of changes.
  - **DO**: edit/add an .svg and add it to your .Rmd presentation in the same commit
  - **DON'T**: edit for a full day and put this in a single commit (or worse: forget to...)
- Commits should have informative messages so you (and others) can trace your steps



	COMMENT	DATE
○	CREATED MAIN LOOP & TIMING CONTROL	14 HOURS AGO
○	ENABLED CONFIG FILE PARSING	9 HOURS AGO
○	MISC BUGFIXES	5 HOURS AGO
○	CODE ADDITIONS/EDITS	4 HOURS AGO
○	MORE CODE	4 HOURS AGO
○	HERE HAVE CODE	4 HOURS AGO
○	AAAAA	3 HOURS AGO
○	ADKFJSLKDFJSDKLFJ	3 HOURS AGO
○	MY HANDS ARE TYPING WORDS	2 HOURS AGO
○	HAAAAAANDS	2 HOURS AGO

AS A PROJECT DRAGS ON, MY GIT COMMIT  
MESSAGES GET LESS AND LESS INFORMATIVE.

- Track most files; .gitignore those files you don't.
- Explore new ideas with branches, keep a stable version on `master`

# Do you want to learn more?

- A [Software Carpentry course on git](#)
- A [version control + git tutorial](#) on Atlassian
- A [git cheatsheet](#) from Atlassian
- A book with [all the ins and outs of git](#) from the git website

