WORKING TOGETHER AND USING GIT

JOSH MURRAY

NOVEMBER 3RD, 2020

OBJECTIVES

- Beginning a new project
  - File/Folder structure
  - File naming
  - File paths
- Introduction to Git
- Using git with Github

Teamwork!

Teamwork!

NEW PROJECT SETUP

- README.md
- .gitignore
- project.Rproj
- data/
  - raw_data/
  - clean_data/
  - data_references
- code/
- figures/
  - exploratory_figures/
  - explanatory_figures/
- products/
  - writing/

README.MD

This is a very important file. The README file displays as the homepage on any git hosting platform (e.g. github). You use it to:

- Introduce the reader to the project
- Give instructions on how to set up the project.
- e.g. clone repo, run `renv::restore()`, etc…
- Give instructions on which files to fun in which order
- Run `clean_code.R` ingest raw data and do basic cleaning
- Run `prepocess_data.R` to prepare the data for modeling
- `utils.R` contains helper functions required for the analysis
- `descriptive-report.Rmd` creates the html report with our exploratory analysis
- Give instructions on how to contribute to the repo

.GITIGNORE

This important file tells you which files to ignore when pushing to Github to ignore when you are pushing to the web:

- Usually raw/sensitive data
- Large files
- Any file that might contains passwords, tokens, etc…

.GITIGNORE FORMAT

A gitignore file specifies intentionally untracked files that Git should ignore. It is a plain text file where:

- blank lines are used to separate entries
- a # can be used for comments
- “/” denotes a directory separator
- An asterisk "*" matches anything except a slash. The character “?” matches any one character except “/”. The range notation, e.g. [a-zA-Z], can be used to match one of the characters in a range. see here and here for an R or Python gitignore files.

```
# no .csv files
*.csv
#no R data files
*.Rda
```

DATA FOLDER

This folder should contain:

- `Raw data/`
- `Clean data/`
- Description files which contain a data dictionary (if it exists), along with instructions on how to get from raw data to clean data

CODE FOLDER

When you first get your hands on data, you may be tempted to write a bunch of quick and dirty code to explore the data. This is okay! Store it in the `code/raw_code` folder. This folder is for you!

In the `code/final_code` folder, we store the code which is

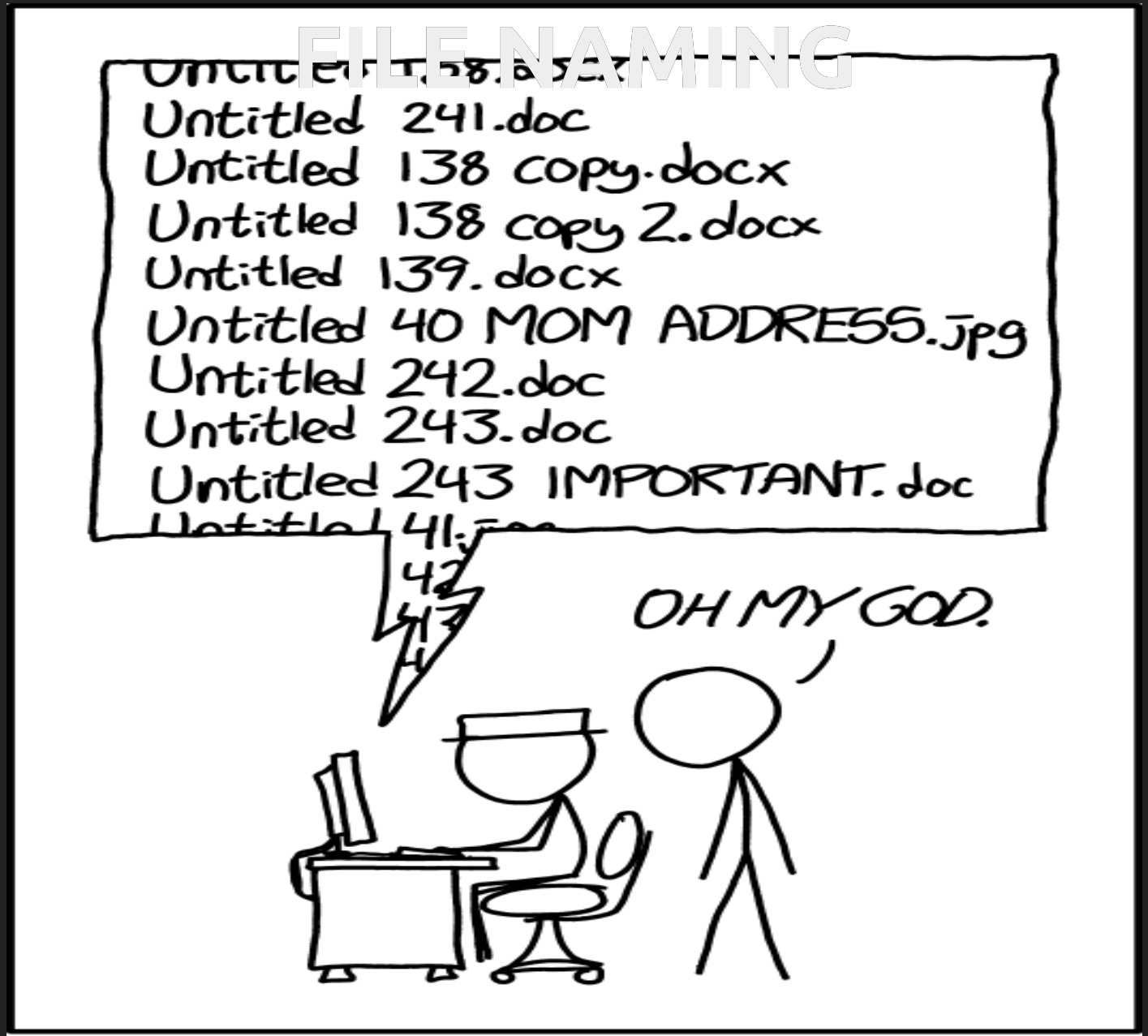
- Organized
- Easy to follow
- Reproducible

FIGURES

- `figures/exploratory_figures`: This are quick and dirty plots to explore the data, understand its structure, and get a sense of relationships between variables
- `figures/explanatory_figures`: Polished final figures that you will share with others. These are generally ordered and help tell the story of your final data analysis.

PRODUCTS FOLDER

This folder is used for storing your manuscripts (`/writing`) or any apps, dashboards you build. I would only include simple apps, dashboards that supplement the analysis. Full app/dashboard products should go in their own project and folder structure.



PROTIP: NEVER LOOK IN SOMEONE ELSE'S DOCUMENTS FOLDER.

FILE NAMING

I take my philosophy from Jenny Bryan, who says that file names should be

1. Human readable
2. Machine readable
3. Plays well with default ordering

MACHINE READABLE

- Regular Expression friendly
  - No spaces,
  - No punctuation
  - No accented characters
  - Case sensitive

```
> list.files(path, pattern = "smh_midnight") %>% head()
[1] "smh_midnight_report_2020_0405_180358.csv" "smh_midnight_report_2020_0406_000246.csv"
[3] "smh_midnight_report_2020_0406_180316.csv" "smh_midnight_report_2020_0407_000257.csv"
[5] "smh_midnight_report_2020_0407_100253.csv" "smh_midnight_report_2020_0408_000329.csv"
>
```

HUMAN READABLE

- The file name contains info on its content:
- Should be able to figure out what something is based on its name
- Err on the side of long names

bad_names	**good_names**
Josh's Lecture Notes.html	josh_lecture_notes.html
JoshMurrayReport.Rmd	josh-murray-report.Rmd
Final Report!.Rmd	final-report.Rmd
FinalReportGIMprojectv2.Rmd	final_report_gim_project_v2.Rmd

RELATIVE PATHS

When working on a project together, you are going to have to share files/data. When sharing files (e.g. through github), the files are no longer on your computer. Therefore, paths like:

```
C:/Users/Josh/Projects/GIM-risk-score/code/01_clean
```

Won't exist on a collaborators computer. This is where the here package can help!

HERE PACKAGE

- The here package looks for a `.Rproj` file and points to that directory
- You can use the here function to define paths in your code to code or data files
  - e.g. `here('code', 'raw_code', '01-explore-raw-dat`

CODING STYLE

Having a clean and consistent coding style helps collaborators understand your work more easily. Collaborating is much easier when there are consistent rules for naming variables, functions, etc..

There are many approaches to styling code

- Googles R style guide
- Tidyverse R style guide
- bioconductor style guide
- Google's Python style guide
- PEP 8 style guide for Python

REPRODUCIBILITY

DEV

Development

1. Dependencies
2. Secrets
3. Environment Variables
4. Selecting config's in code

ENVIRONMENTS

Development – Dependencies

Important that we're consistent with which versions of different R packages (or python libraries) we're using to avoid surprises when we go to deploy.

R – {renv} package and renv.lock

```
{
  "R": {
    "Version": "3.6.2",
    "Repositories": [
      {
        "Name": "CRAN",
        "URL": "https://cloud.r-project.org"
      }
    ]
  },
  "Packages": {
    "BH": {
      "Package": "BH",
      "Version": "1.72.0-3",
      "Source": "Repository",
      "Repository": "CRAN",
      "Hash": "8f9ce74c6417d61f0782cbae5fd2b7b0"
    },
    "DBI": {
```

Python – virtualenv and requirements.txt

```
backcall==0.1.0
certifi==2019.3.9
chardet==3.0.4
decorator==4.3.0
dj-database-url==0.5.0
Django==2.2
django-cors-headers==2.4.0
djangorestframework==3.9.1
djangorestframework-jwt==1.11.0
gunicorn==19.9.0
idna==2.8
ipython==6.4.0
ipython-genutils==0.2.0
jedi==0.12.1
numpy==1.16.2
pandas==0.24.2
```

RENV

The general workflow when working with renv is:

1. Call `renv::init()` to initialize a new project-local environment with a private R library,
2. Work in the project as normal, installing and removing new R packages as they are needed in the project,
3. Call `renv::snapshot()` to save the state of the project library to the lockfile (called renv.lock),
4. Continue working on your project, installing and updating R packages as needed.
5. Call `renv::snapshot()` again to save the state of your project library if your attempts to update R packages were successful, or call `renv::restore()` to revert to the previous state as encoded in the lockfile if your attempts to update packages introduced some new problems.

CONFIG FILES/ENVIRONMENT VARIABLES

Development – R {config} package

R code:

```
109
110
111
112 config <- config::get(file = here::here("inst","golem-config.yml"))
113
114 db_connection <- connect_to_db(
115   db_host = config$db_host,
116   db_name = config$db_name,
117   db_port = config$db_port,
118   username = keyring::key_get("db", username = "db_username"),
119   password = keyring::key_get("db", username = "db_password")
120 )
121
122
123
```

./inst/golem-config.yml

```
1  default:
2    golem_version: 0.0.0.9000
3    db_host: 172.27.21.215
4    db_name: edrndb_dev
5    db_port: 5432
6  staging:
7    golem_version: 0.0.0.9000
8    db_host: 172.27.21.215
9    db_name: edrndb_dev
10   db_port: 5432
11 production:
12   golem_version: 0.0.0.9000
13   db_host: 172.27.21.215
14   db_name: edrndb_prod
15   db_port: 5432
16
```

- Much easier/faster than specifying individual environment variables for each of these!
- Python users, see pyconfig pip library for similar

VERSION CONTROL WITH GIT AND GITHUB

"FINAL".doc

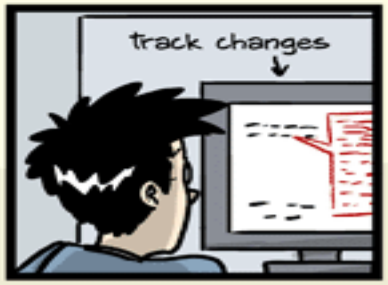
VERSION CONTROL

FINAL.doc!

FINAL_rev.2.doc

FINAL_rev.6.COMMENTS.doc

FINAL_rev.8.comments5.
CORRECTIONS.doc



FINAL_rev.18.comments7.
corrections9.MORE.30.doc

FINAL_rev.22.comments49.
corrections.10.#@$%WHYDID
ICOMETOGRADSCHOOL????.doc

WHY GIT

Why is Git/GitHub Important?

- **Version control:** easy to revert to previous changes
- **Transparency:** knowing exactly what code is "live" in staging and production environments
- **Collaboration**: multiple contributors adding incremental changes to the same project

WHY GIT AND GITHUB?

- Git is a version control system for tracking changes in a set of files called a *repository* in a structured way
- We can use git to manage all of the files in the folder structure we have specified here: code, figures, manuscripts, etc…
- New users to Git will struggle with the new workflow that accompanies it.
- The greatest benefits to using Git come from collaborating in a team setting
- *Github* is a hosting service that creates an online home for your repository
  - It allows others to see your work, and maybe even make changes if you allow

WHAT ARE THE BENEFITS?

- A way to display your work to potential employers
- If you need to work together on a project (hint hint). Individual team memebers work independently and then reconcile changes and share results with github
- To learn from others! Github is a great way to see the nuts and bolts of the packages you use, or the cool code someone created to analyse a dataset you care about.

REGISTERING A GITHUB ACCOUNT

- You can register an account for free at github.com
- Select a username. Some advice from Jenny Bryan:
  - Incorporate your name somehow
  - reuse your username from other public accounts (e.g. twitter)
  - Pick a name that you won't feel embarrassed saying out loud to colleagues
  - make it timeless. Don't highlight your current position/school
  - It's highly recommended that you use all lowercase letters
  - Don't pick usernames that are common programming words

CHECKING IF GIT IS ALREADY INSTALLED

Check to see if it is already installed, run `where git` (windows), `which git` (linux) to see where the executable is located. You can run terminal commands in R using `system()`

```
# Where is git installed
print(system("where git"))
```

```
## [1] 0
```

```
# which version of git is installed
print(system('git --version'))
```

```
## [1] 0
```

If these return anything, fantastic news. You already have git successfuly installed.

INSTALLING GIT

- Installing on Windows: install Git for windows. This will install “Git Bash” a useful command line tool.
- On Ubuntu or Debian Linux:
  `sudo apt-get install git`
- On Redhat Linux: `sudo yum install git`
- On Mac: `xcode-select --install`

GETTING STARTED WITH GIT

You can set your github username and email at the terminal with the following:

```
# set username
git config --global user.name 'Jane Doe'
# set email address
git config --global user.email 'jane@example.com'
# check display
git config --global --list
```

Or with R using the `usethis` package

```
usethis::use_git_config(user.name = "Jane Doe", user.email =
        "jane@example.org")
```

GIT WORKFLOW BASICS (SOLO)

Using git isn’t all that different from your usual workflows.

- **Pull** any changes that were made to the code.
- **Add** your changes (to the launching pad).
- **Commit** your changes (from the launching pad to the Moon).
- **Push your** changes (from the Moon to Mars).

GIT WORKFLOW (WHEN WORKING WITH OTHERS)

- **Pull** any changes
- Create a new branch (recreate the space station in a parallel universe)
- **Add** your changes (to the launching pad of your parallel universe space station).
- **Commit** your changes (to a Moon in a parallel universe).
- **Push** your changes (from the parallel universe Moon to a parallel universe Mars)
- Create a **pull request** (or PR) (from parallel universe Mars to real verse Mars)

GIT PULL

You can pull the most recent changes to the repo with `git pull`.

```
# pull the most recent changes
git pull
```

GIT LOG

You can see the commit history with `git log`.

```
# pull the most recent changes

git log
```

GIT STATUS

After working on your code and making changes, you can examine which files have changed with 1git status`. This allows you to check in on the status of the repo.

```
# check for any changes

git status
```

ADDING CHANGES TO THE STAGING AREA

When you have saved changes in your working directory, you can add them to the staging area with `git add`

```
git add -A
```

The `-A` is an option which says to add all changes

COMMIT MESSAGES

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

WRITING GOOD COMMIT MESSAGES

from https://chris.beams.io/posts/git-commit/

1. Separate subject from body with a blank line
2. Limit the subject line to 50 characters
3. Capitalize the subject line
4. Do not end the subject line with a period
5. Use the imperative mood in the subject line
6. Wrap the body at 72 characters
7. Use the body to explain what and why vs how

MAKING A COMMIT

After you have staged some changes with `git add -A`, you can commit these changes with `git commit`

```
git commit -m "initial commit"
```

the `-m` option specifies that you are adding a commit message

PUSHING CHANGES TO THE REMOTE REPO

You can push changes in your local repo to the remote repo with `git push`

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

In the above origin is the name of the remote repository and master is the name of the branch you are pushing.

DEMOS

We will walk through the following demos

- Creating a local git repo from the command line
- Working with github
- Branching and Pull requests
  - Code review
- Fixing common mistakes