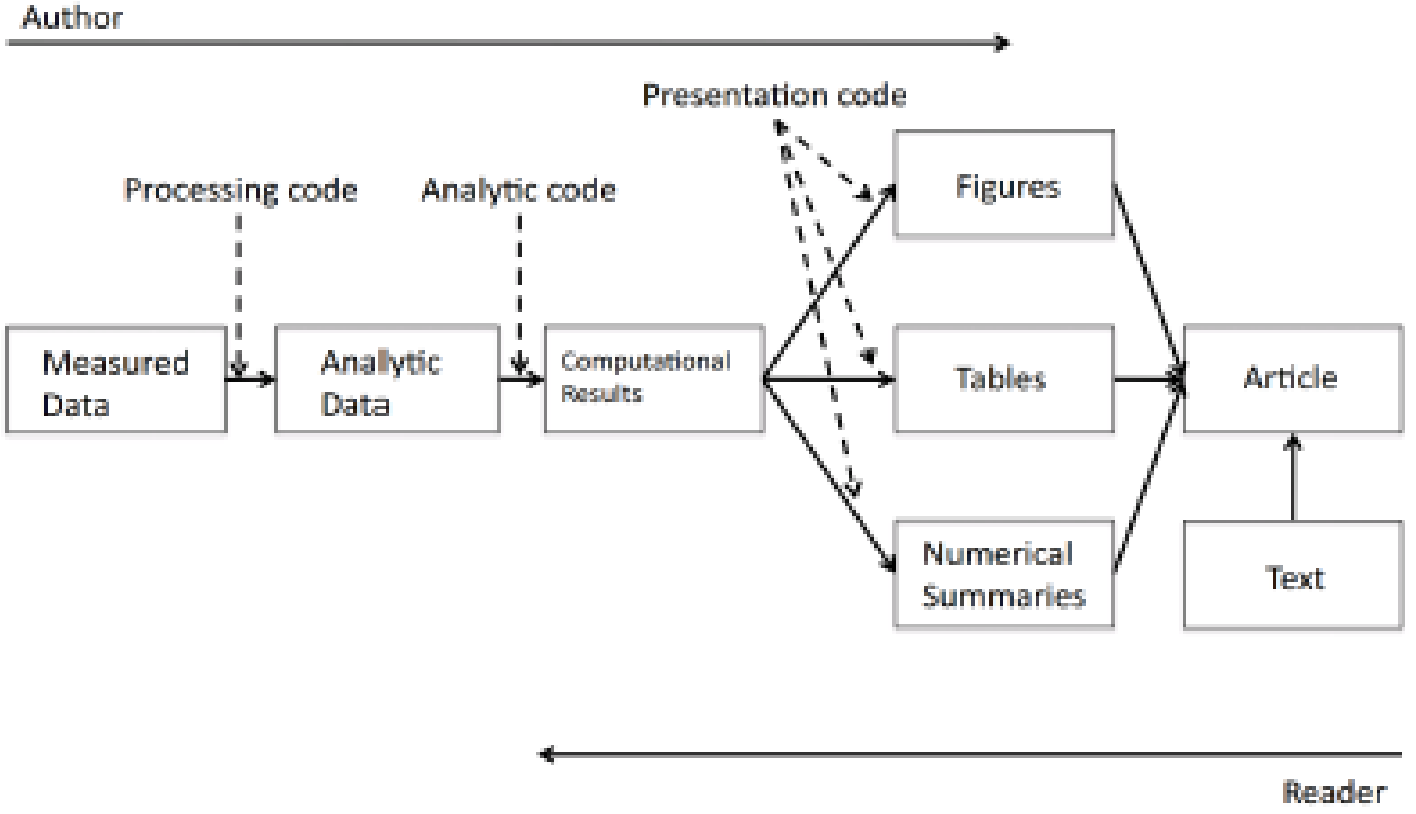
Tools for reproducible research

January 2017

Boriana Pratt
Princeton University

Research Pipeline



Reproducible Research

"Reproducible research is the idea that data analyses, and more generally, scientific claims, are published with their data and software code so that others may verify the findings and build upon them."

(From the description of "*Reproducible Research " course available on Coursera from John Hopkins University)*

"By reproducible research, we mean research papers with accompanying software tools that allow the reader to directly reproduce the results and employ the methods that are presented in the research paper."

(From the abstract of Gentleman, Robert and Temple Lang, Duncan, "Statistical Analyses and Reproducible Research" (May 2004). *Bioconductor Project Working Papers*)

Reproducible Research

Considerations:

- being able to reproduce own results at a later date
- manage changes to data, analysis and results
- satisfy new journal requirements

A reproducible report should be self-contained.

Guidelines for reproducible research

- Everything done with a script
  - download data via script
  - Don't hand edit files
  - Analysis should be in scripts, same for data cleaning
  - Set and save the seed (in R also run sessionInfo())

- Organize data, code, and dependencies
  - Encapsulate everything
  - Separate raw data from derived data
  - Separate data from code
  - Use relative paths
  - Write readme files (document everything!)

- Automate the process
  - Using make (makefile)

- Turn scripts into reproducible reports
  - Include documentation (why you did this)

- Use version control

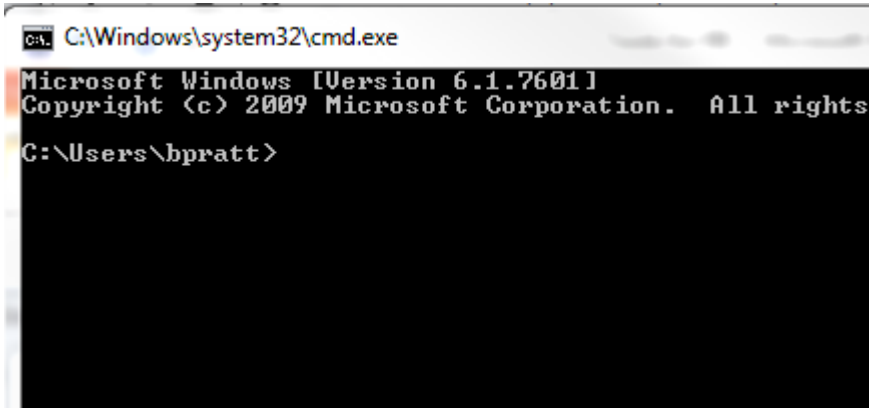
From: http://kbroman.org/steps2rr/ See also https://rpubs.com/marschmi/105639

Tools

- File dependency management – make, R packrat
- Version control - Git / Github
- Tools for R:
    - Sweave (.Rnw)
    - Knitr
    - Rmarkdown
    - Notebook demo
- Tools for Stata
    - Weaver command
    - markdown command

cmd.exe

- click on the “Start” button
- click on the lower-left text input box (“Search programs and files” or “Ask me anything”)
- type “cmd” and press the enter key



A few commands:

- cd (pwd) – get current directory
- cd (cd) : change directory
- dir (ls): list files (and subfolders)
- copy (cp): copy a file
- del (rm): delete a file
-mkdir (mkdir): make new directory
-rmdir (rmdir): delete a directory

cmd.exe

Exercise:

1. Find out what directory you are in
2. See what's in it
3. Change directory
4. Make a new directory and go to it

Managing File Dependencies

Make

- Tool to create a roadmap of the programs and outputs produced.
- Rules saved in makefile.mk

Here is what a simple rule looks like:

```
target:  dependencies ...
(TAB)    commands
         ...
```

Or

```
target: prerequisites
(TAB)    recipe
         ...
```

Documentation for GNU Make https://www.gnu.org/software/make/

Make example (to demonstrate)

I have two files in my directory:

prep_data.R – creates and saves a data frame as dt.csv

fig1.Rmd – loads the data frame and makes a plot

Workflow:

prep_data.R -> dt.csv -> fig1.Rmd -> fig1.pdf

File dependencies:

report.pdf depends on report.Rmd, which depends on dt.Rds, which depends on prep_data.R

The makefile would be:

```
fig1.pdf: fig1.Rmd dt.csv
        rscript –e "library(rmarkdown); \
        render('fig1.Rmd','pdf_document')"

dt.csv:  prep_data.R
        rscript prep_data.R

clean:
        del dt.csv
        del fig1.pdf
```

Then run it from the command (terminal) window: `make –f makefile`

Let's try it…

Question: Is the R file reproducible?

Make extras

Comment lines start with # (pound sign)

Variables can be defined like this:

`objects = fig1.pdf fig2.pdf` and then accessed with `$(objects)`

`VPATH = src:../RR` – to search other directories for file dependencies

Automatic variables:

`$@` - target filename

`$<` - name of the first prerequisite

`$^` - all prerequisites

For example could write:

```
fig1.pdf: fig1.R
        rscript  $<
```

% can be used as wildcat character

Could be used as:

```
%.pdf: %.R
        rscript  $<
```

This means the same as above if you have a file `fig1.R` in your current directory. If you have two files `fig1.R` and `fig2.R`, the above rule will produce two files: `fig1.pdf` and `fig2.pdf`.

Managing Environment/Package Dependencies

Results depend not only on code and data, but also on the computing environment.

R packrat – dependency management system for R packages (separate for each project).

Makes R projects:

- Isolate - updating a package for one project won't break other projects
- Portable - can easily transfer to another machine (or platform)
- Reproducible - records the version of packages the project depends on

Install and use packrat:

```
install.packages("packrat")
```

Before starting anything in a directory initialize it:

```
packrat::init("C:/RR")
```

Example:

I save and run a file (anal.R) in the initialized directory (C:/RR).

`packrat::snapshot()`

```
> packrat::snapshot()

Adding these packages to packrat:

                _
    BH          1.60.0-2
    DBI         0.5-1
    R6          2.2.0
    Rcpp        0.12.7
    assertthat  0.1
    dplyr       0.5.0
    lazyeval    0.2.0
    magrittr    1.5
    tibble      1.2

Fetching sources for BH (1.60.0-2) ... OK (CRAN archived)
Fetching sources for DBI (0.5-1) ... OK (CRAN current)
```

`packrat::status()`

```
> packrat::status()

The following packages are used in your code, tracked by packrat, but no longer present in your library:
                    from   to
    BH           1.60.0-2   NA
    DBI             0.5-1   NA
    R6              2.2.0   NA
    Rcpp           0.12.7   NA
    assertthat        0.1   NA
    dplyr           0.5.0   NA
    lazyeval        0.2.0   NA
    magrittr          1.5   NA
    tibble            1.2   NA

Use packrat::restore() to restore these libraries.
```

File Dependencies

Packrat will keep a list of R packages and their versions that are used in any of the R (.R, .Rnw, .Rmd, .md, etc.) program files in the folder.

Packrat commands:

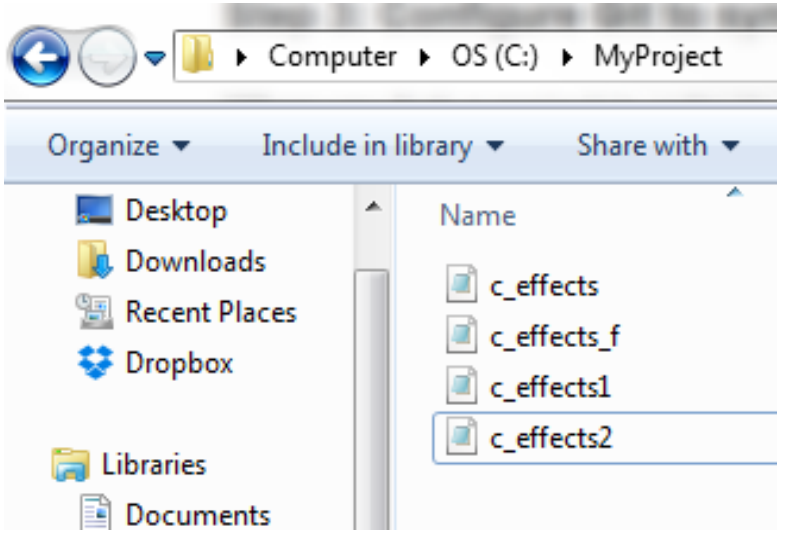
`packrat::status()`	Shows differences between the projects' packrat dependencies and its R scripts.
`packrat::snapshot()`	Saves the current state of your library.
`packrat::restore()`	Restores the library state saved in the most recent snapshot.
`packrat::clean()`	Removes unused packages from your library.

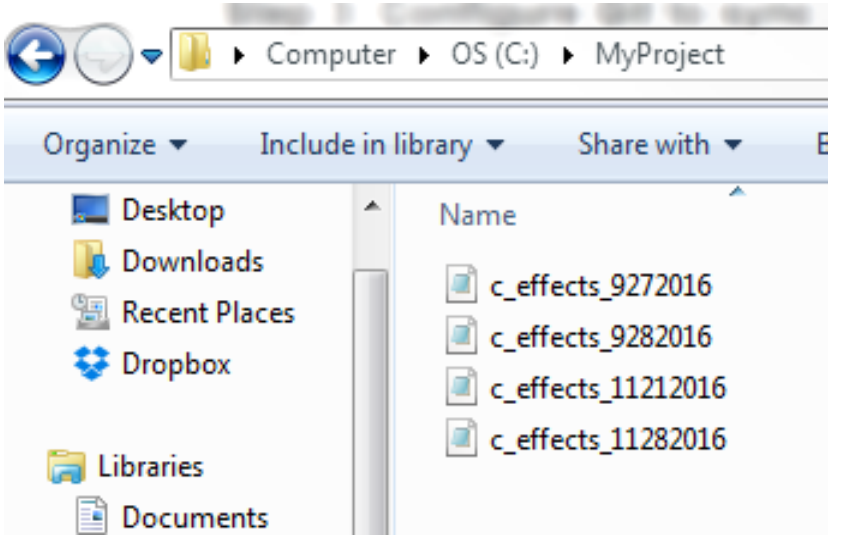
Bundle and share your projects with `packrat::bundle()` and `packrat::unbundle()`.

For more see: https://rstudio.github.io/packrat/commands.html

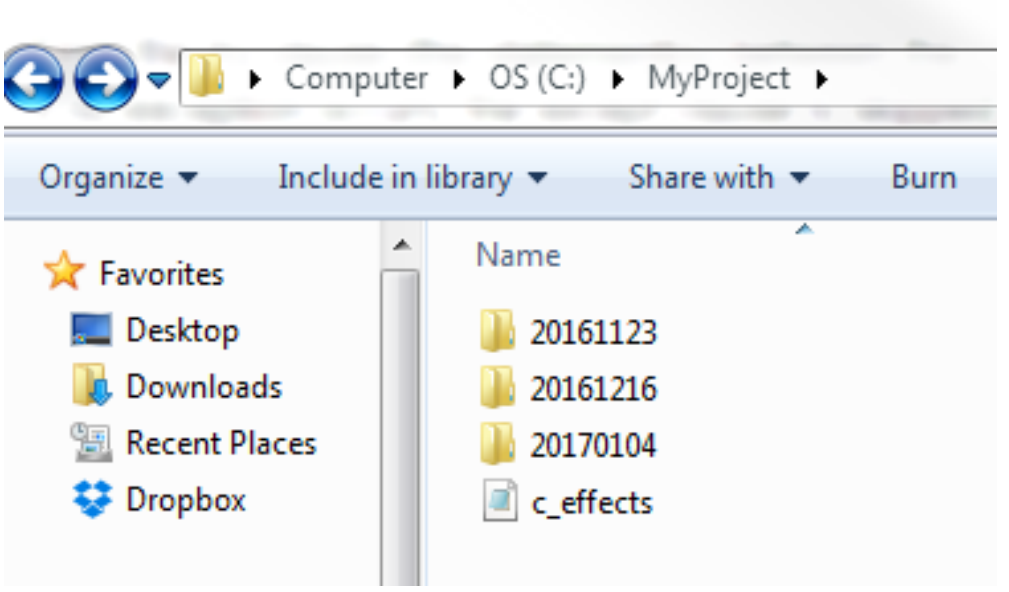
Any questions?

Version Control





Version Control



Or use Git…

Version Control

Git is:

- open source program for tracking changes in text files
- version control system, a tool to manage your source code history

Git was written by the author of the Linux operating system.

GitHub is:

- hosting service for Git repositories

At the heart of GitHub is an open source version control system (VCS) called *Git*. Git is responsible for everything GitHub-related that happens locally on your computer.

To download: https://desktop.github.com/ or https://git-scm.com/downloads

Git

How Git works: stores a snapshot of the code file – called ‘commit’.
Stores the file as well as metadata (who did the commit, date, notes).

- Git stores everything under the (hidden) .git directory
- Everything in Git is check-summed before it is stored (by 40-character SHA-1 tag)
- Git is a distributed system – everyone has their own local copy of the whole repository

For more check this website: https://git-scm.com/

Git

Setup

- open the cmd.exe window (terminal window)
- set name and address for all commits

```
>git config --global user.name "Me"
>git config --global user.email me@example.edu
```

Question:

What do you expect the command '`git config --global user.name`' to return?

Initial commit:

- Create a project directory
- Create a R markdown file (ex_mkdown.Rmd)

```
git init
git add ex_mkdown.Rmd
git commit -m "first commit"
git status
git log
```

Let's try it…

Git

To check repository status and history:

`>git status`	-shows if there are not committed changes to any files
`>git diff`	- shows differences between last snapshot and the current file
`>git log`	- see all commits to date

Before a file that is new or changed has a snapshot taken, it has to be staged (cashed).

To stage a file (files):

`>git add "file_name"`	- stage one file
`>git add css/`	- stage a directory (and subdirectories)
`>git add –all`	- stage all files in the current directory

To commit (take snapshot):

`>git commit –m "note"`	- commit what is in the staging area

Git

More commands

`>git init` - initialize a directory to be 'tracked'

`>git help`

`>git config` - set up

`>git config --help` – to see what options are available

Undoing

`>git commit –ammend`

If you commit too early and forget a file, for example:

`>git commit –m "first commit"`

`>git add` *`forgotten_file`*

`>git commit --ammend`

Unstaging a staged (git add) file:

`>git reset HEAD` *`file_to_unstage`*

Undo changes to a file (modified it and would like to revert to how it was at the last commit):

`>git checkout --` *`file_name`*

Git

Let’s make a change to the file - add a line.

`> git diff` – to see what has changed

Commit the change.

Now let’s add another file to the folder and commit it...

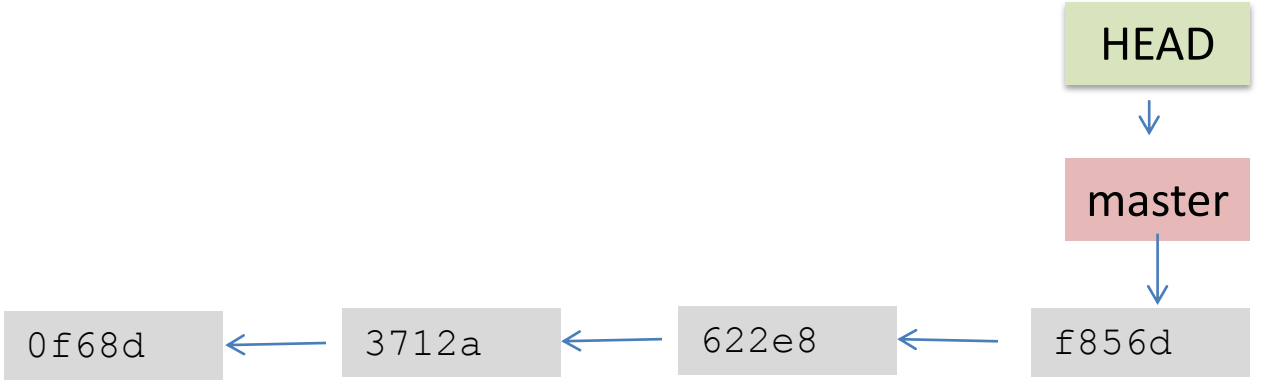
Let’s look at the history/log…

```
>git log --oneline --decorate
f856d83 (HEAD -> master) minor change
622e8ef third commit
3712a38 adding a line
0f68d35 first commit
```

Git

```
>git log --oneline --decorate
f856d83 (HEAD -> master) minor change
622e8ef third commit
3712a38 adding a line
0f68d35 first commit
```

How Git works:



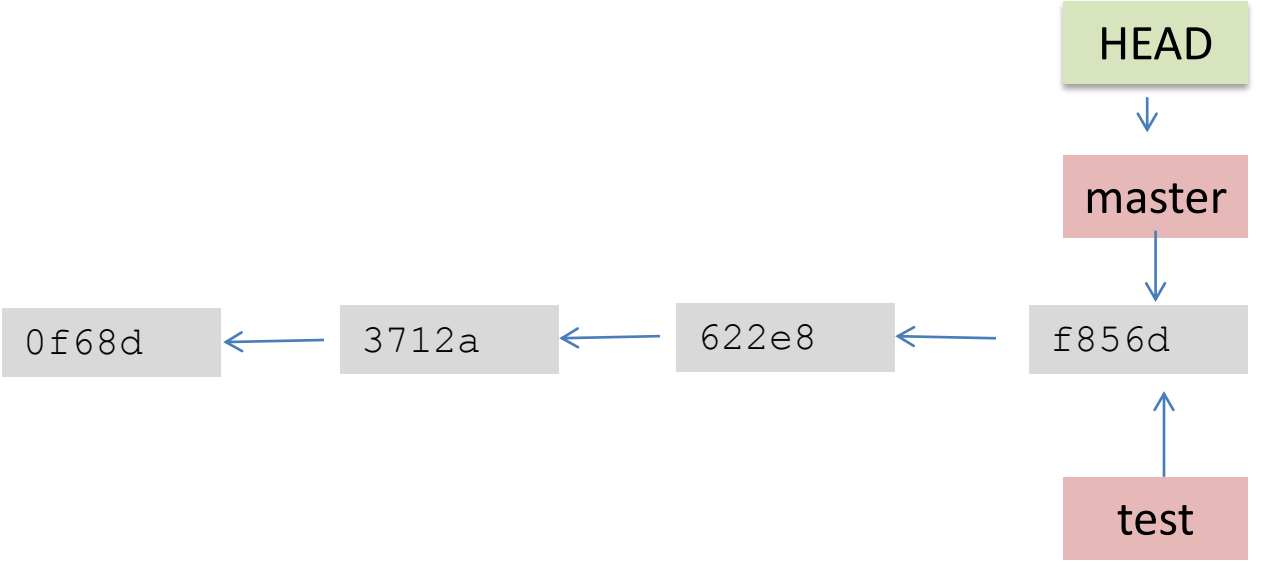
each commit points to its parent
HEAD is a special pointer (points to where you are currently in the tree of commits)

Git

Make a branch:

```
>git branch test
```

```
>git log --oneline --decorate
f856d83 (HEAD -> master, test) minor change
622e8ef third commit
3712a38 adding a line
0f68d35 first commit
```

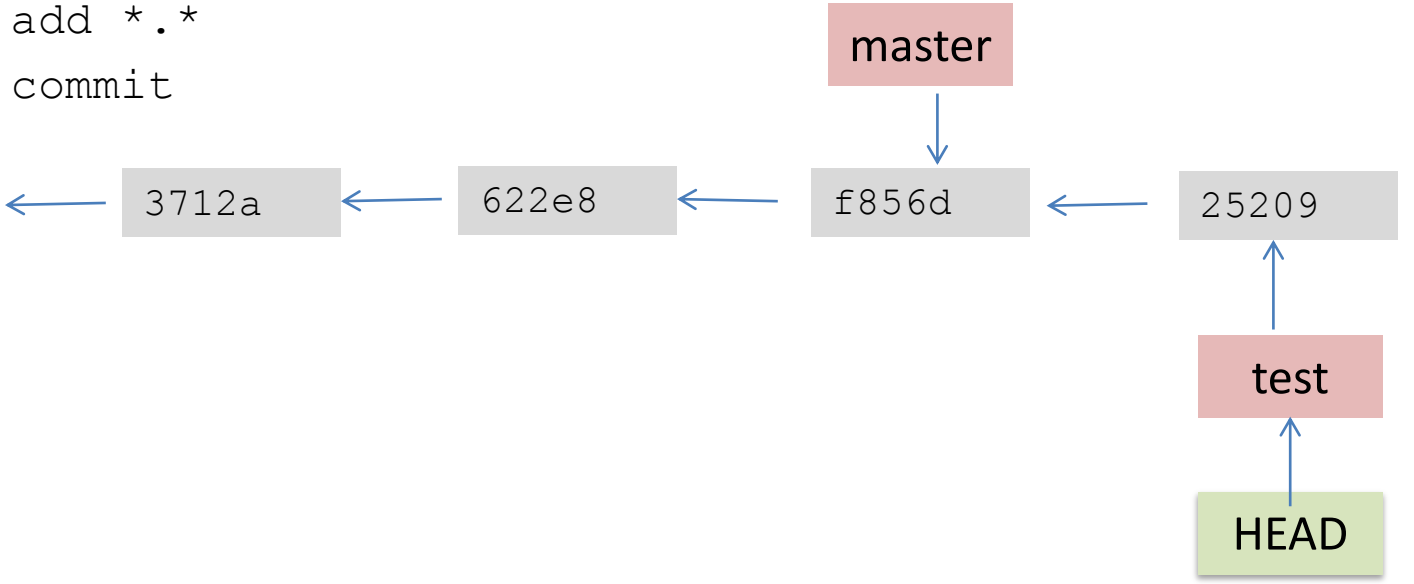


Checkout a branch:

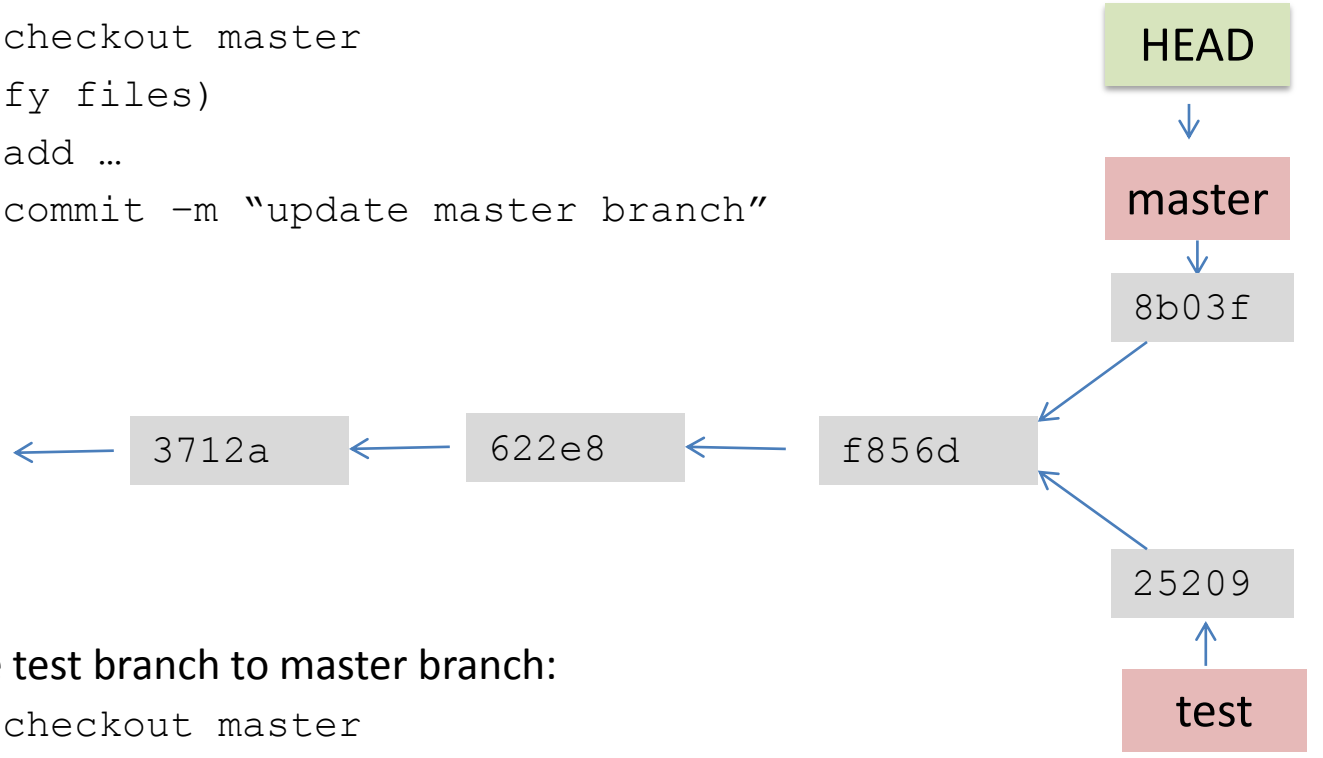
```
>git checkout test
```

Proceed with changes and commit:

```
>git add *.*
>git commit
```



```
>git checkout master
(modify files)
>git add …
>git commit –m “update master branch”
```

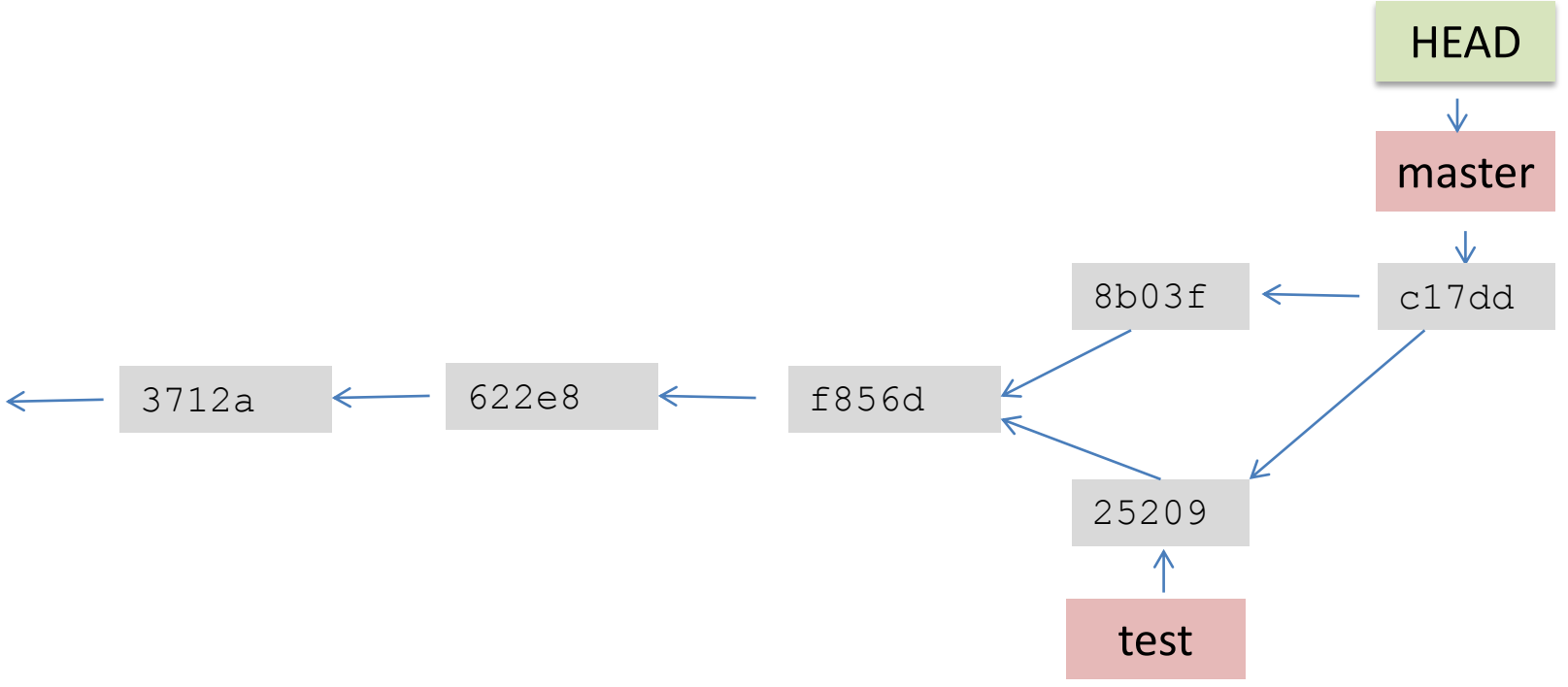


Merge test branch to master branch:

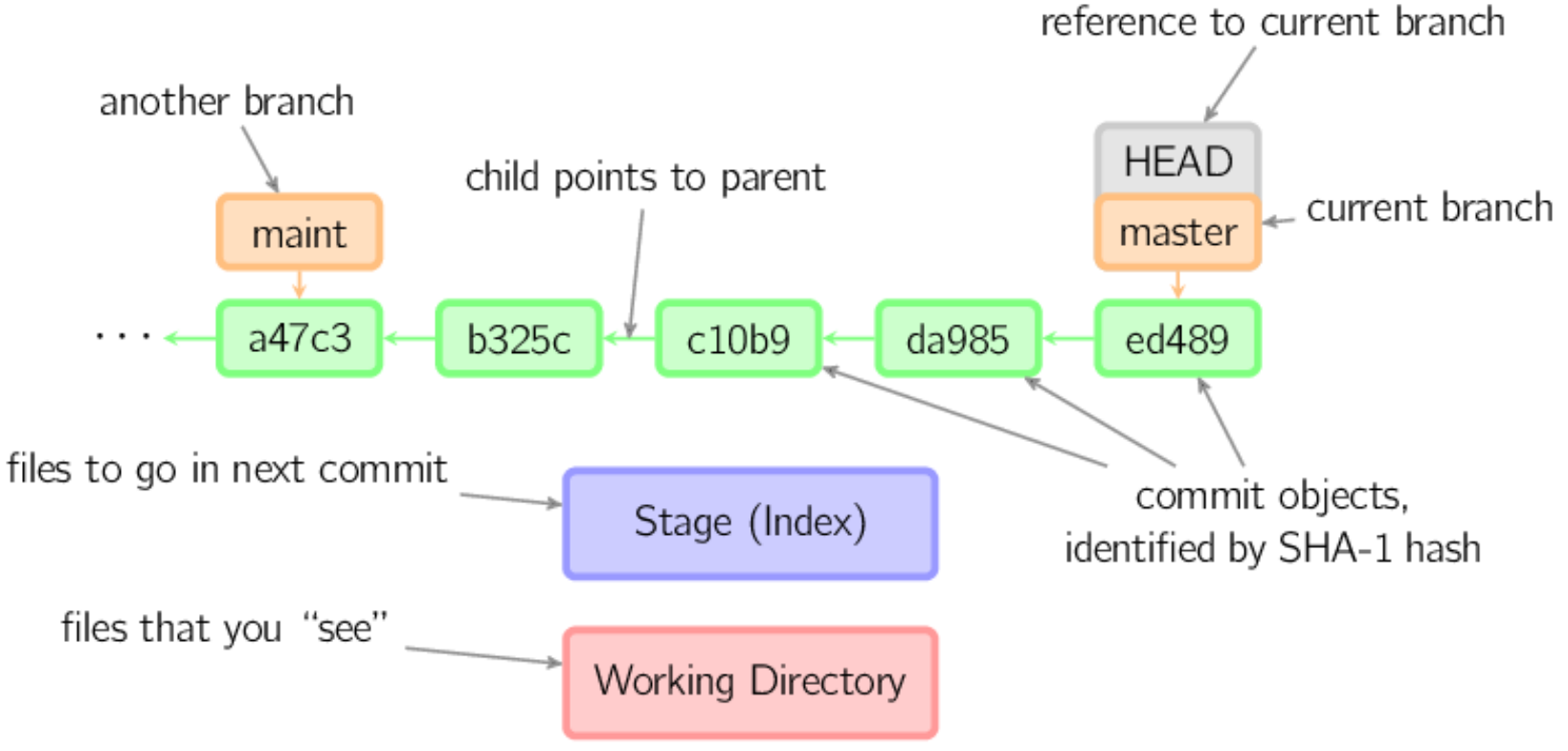
```
>git checkout master

>git merge test
Auto-merging ex_mkdown.Rmd
CONFLICT (content):  Merge conflict in ex_mkdown.Rmd
Automatic merge failed; fix conflicts and then commit the result.
```

When you get a “conflict” message you have to manually edit the conflicting parts, remove <<<<<<<, ======= and >>>>>>> , then stage the file and commit.



Git


reference to current branch
another branch
HEAD
child points to parent
maint
master
current branch
· · ·
a47c3
b325c
c10b9
da985
ed489
files to go in next commit
Stage (Index)
commit objects,
identified by SHA-1 hash
files that you "see"
Working Directory

GitHub

GitHub (https://github.com/about) is:

- a repository hosting service
- launched in 2008
- a place to share your code

Cloning

A public repository on GitHub can easily be cloned:

```
>git clone https://github.com/ASvyatkovskiy/PrincetonPy
```

This creates a copy of the repo on your local machine.

Github

Pushing to a GitHub repository

Requires a GitHub account, then you can create a repository.

To get your code to GitHub see here : https://help.github.com/articles/create-a-repo

```
>git remote add origin https://github.com/...
>git push –u origin master
```

Will ask for username and password…

Pull from a GitHub repository

```
>git pull
```

If pulling from your GitHub repository, `pull` does both `fetch` and `merge`. If you would like someone to pull code from your repository you have to create a pull request.

Github

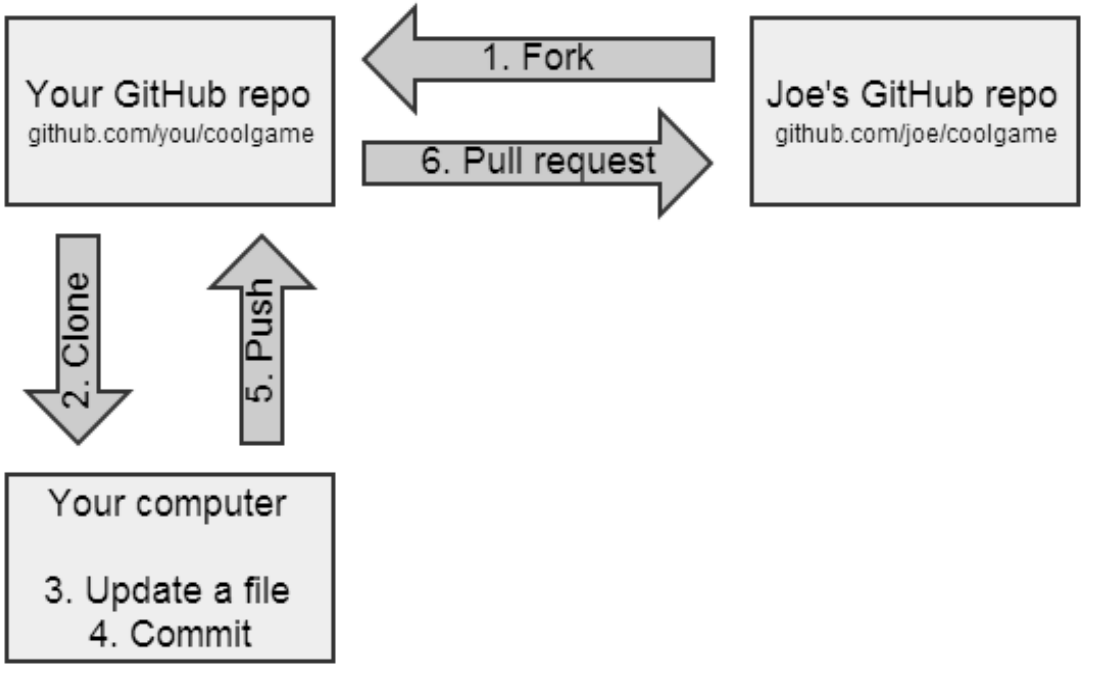
Working with repositories:

A ***fork*** is a copy of a repository (a clone on the server side). Forking is done through GitHub.

To create pull request :

1. Create a fork (clone the repo under your account).
2. Clone to your local computer.
3. Create a branch or update files and commit changes.
4. Push changes to own GitHub repo.
5. Send the original project account a PR.

Diagram from Kevin Markham's simple guide to forks



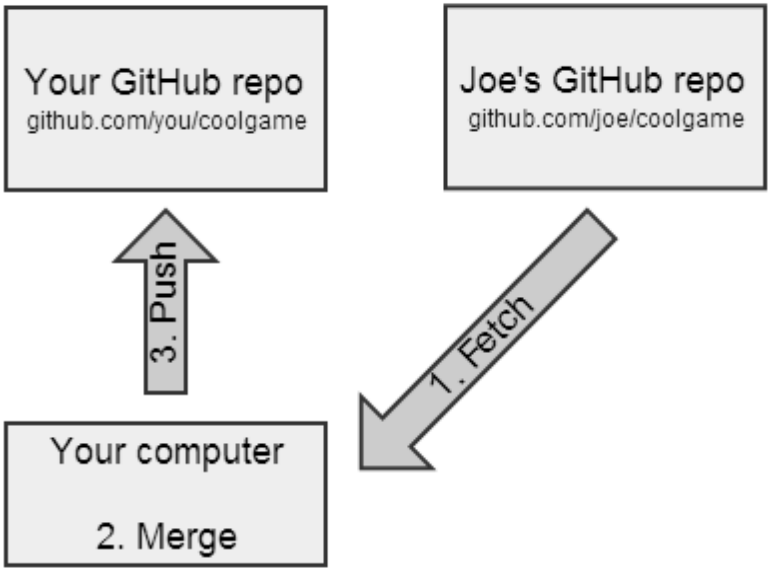
Github

To accept pull request:

1. Review code and documentation.
2. If it's OK, pull from the collaborator's GitHub repo onto local master.
3. After merge push back up to the project's GitHub repo.

Syncing a fork

1. Fetch changes
2. Merge to your local repo
3. Push the updates to your GitHub repo (optional)



Git/GitHub resources

- Scott Chacon and Ben Straub's (open source) book "Pro Git" https://git-scm.com/book/en/v2
- Kevin Markham's (Data School) youtube videos on "Version control with Git and GitHub" http://www.dataschool.io/git-and-github-videos-for-beginners/

  (or https://www.youtube.com/user/dataschool)
- Karl Broman's tutorial "git/github guide" http://kbroman.org/github_tutorial/
- Gregg Pollack "Try Git" https://www.codeschool.com/courses/try-git

Git documentation page https://git-scm.com/doc

GitHub guides https://guides.github.com/

Checklist

1. Was as much as possible done by the computer?
2. Was any file hand-edited, or any part of the analysis done by hand?
3. Is everything documented, including the software environment?
4. Was a version control system used?
5. Have we saved any output that we cannot reconstruct from original data and the code?
6. How far back in the analysis pipeline can we go before our results are no longer automatically reproducible?

Question:

It's a good idea to call sessionInfo() in your R code? (Y/N)

Any questions?

Recommendations for reproducible research (from https://rpubs.com/marschmi/105639)

1. Encapsulate the full project into one directory that is supported with version control.
2. Release your code and data.
3. Document everything and use code as documentation.
4. Make figures, tables, and statistics the results of scripts.
5. Write code that uses relative paths.
6. Always Set your seed.