

02. Toolkit

Data Science for Economists — Summer 2024

Julian Hinz

Bielefeld University

DOCKER + VSCODE WORKING?

Session Roadmap

- git
- Docker
- bash
- make
- R

GIT AND GITHUB

Git

- Git is a distributed version control system
 - “Dropbox and the ”Track changes” feature in MS Word have a baby: Git”
- Optimized for code (not data, actually)

GitHub

- Online hosting platform that provides services built on top of the Git system
→ Similar: Bitbucket and GitLab
- Makes Git *a lot* more user friendly
- Seamless integration into lots of other software: VSCode (and RStudio)

4 main Git operations

1. Stage (or “add”): Add changes to the repo history
→ file edits, additions, deletions, etc.
2. Commit: Yes, you are sure these changes should be part of the repo history
→ need to add a message (and optionally a description)
3. Pull: Download new changes made on the GitHub repo (i.e. the upstream remote)
→ either by your collaborators or you on another machine
4. Push: Upload any (committed) local changes to the GitHub repo

4 main Git operations

1. Stage (or “add”): Add changes to the repo history
→ file edits, additions, deletions, etc.
2. Commit: Yes, you are sure these changes should be part of the repo history
→ need to add a message (and optionally a description)
3. Pull: Download new changes made on the GitHub repo (i.e. the upstream remote)
→ either by your collaborators or you on another machine
4. Push: Upload any (committed) local changes to the GitHub repo

4 main Git operations

1. Stage (or “add”): Add changes to the repo history
→ file edits, additions, deletions, etc.
2. Commit: Yes, you are sure these changes should be part of the repo history
→ need to add a message (and optionally a description)
3. Pull: Download new changes made on the GitHub repo (i.e. the upstream remote)
→ either by your collaborators or you on another machine
4. Push: Upload any (committed) local changes to the GitHub repo

4 main Git operations

1. Stage (or “add”): Add changes to the repo history
→ file edits, additions, deletions, etc.
2. Commit: Yes, you are sure these changes should be part of the repo history
→ need to add a message (and optionally a description)
3. Pull: Download new changes made on the GitHub repo (i.e. the upstream remote)
→ either by your collaborators or you on another machine
4. Push: Upload any (committed) local changes to the GitHub repo

Merge conflicts

```
# README
Some text here.
<<<<<< HEAD
Text added by Partner 2.
=====
Text added by Partner 1.
>>>>>> 814e09178910383c128045ce67a58c9c1df3f558.
More text here.
```

- Delete lines that you don't want, then special Git merge conflict symbols
- Then: stage, commit, pull and push

Branches and forks

Branch

- Take snapshot of existing repo and try out a whole new idea without affecting your main branch
- If new idea works, merge back into main branch
 - fix bugs
 - implement new empirical strategies, robustness checks, ...
- If it doesn't work, just delete experimental branch

Branches and forks

Fork

- Forking a repo similar to branch, but creates a copy of entire repo
- Upstream pull request makes merge back into origin repo possible
 - Easy to do on Github

.gitignore

- Tells Git what to ignore
 - exclude whole folders or a class of files (e.g. based on size or type)
- Simply add names of files or folders that should be ignored

DOCKER

“Container” technology



- Docker containers are the software equivalent
 - physical goods <-> software
 - transport modes <-> operating systems

“Container” technology

- Standardized shape and form
- “If it runs on your machine, it will run on my machine.”
- Allows to always run code from a pristine, predictable state

How it works

- Stripped-down version of an operating system
 - Usually a Linux distro like Ubuntu
- Installs all of the programs and dependencies that are needed to run the code
 - + add any extra configurations that are needed/wanted
- Package everything up as a tarball (i.e. compressed file)
 - Containers are like mini, portable OS that contain everything needed to run some piece of software (but nothing more!)

Docker terminology

- Dockerfile: “The sheet music”
 - list of layers and instructions for building a Docker image
- Image: “The MP3 file”
 - tarball packages everything needed
- Container: “Song playing on my phone”
 - running instance of an image

Minimal working example

```
$ docker run --rm -it rocker/r-base
```

docker **run flags**

- **--rm** automatically removes the container once it exits (i.e. clean up)
- **-it** Launch with interactive (i) shell/terminal (t)

A bit more sophisticated working example

```
$ docker run -d -p 8787:8787 -e PASSWORD=verystrong rocker/tidyverse
```

docker run flags

- `-d` detach (i.e. run as background process)
- `-p 8787:8787` share a port with the host computer's browser
- `-e PASSWORD=pswd123` set password for logging on to RStudio Server
- `rocker/tidyverse:4.0.0` prepared `tidyverse` image built on top of R 4.0.0

SHELL

Shell

- Terminology: shell, terminal, tty, command prompt, etc.
→ Same same: command line interface (CLI)
- Many shell variants: focus on Bash (“Bourne again shell”)
- Included by default on Linux and MacOS
- Windows users need to install a Bash-compatible shell

Shell

- Powerful: executing commands and for fixing problems
→ some things you just can't do in an IDE or GUI
- Reproducibility: Scripting is reproducible, clicking is not
- Remote: Interacting with servers and super computers
- Automation: workflow and analysis pipelines, e.g. with Makefile

EXPLORER

DSIER [DEV CONTAINER: DSIER DOCKER]

- > .devcontainer
- > 01-getting-started
- > 02-toolkit
- ◆ .gitignore
- 📄 README.md

PROBLEMS OUTPUT **TERMINAL** PORTS 1 DEBUG CONSOLE

```
astronaut + /workspaces/DSIER (main) $
```

+ - ^ x

Dev Contai...

bash

> OUTLINE

> TIMELINE

SHELL

Basics

```
username@hostname:~$
```

- `username` denotes a specific user
- `hostname` denotes name of the computer
- `:~` denotes the directory path (where `~` signifies the user's home directory).
- `$` denotes the start of the command prompt (`#` for root)

Keyboard shortcuts

- Tab completion
- ↑ (and ↓) keys to scroll through previous commands
- Ctrl + → (and Ctrl + ←) to skip whole words at a time
- Ctrl + a moves the cursor to the beginning of the line
- Ctrl + e moves the cursor to the end of the line
- Ctrl + k deletes everything to the right of the cursor
- Ctrl + u deletes everything to the left of the cursor
- Ctrl + Shift + c to copy and Ctrl + Shift + v to paste

Syntax

- `command option(s) argument(s)`

```
astronaut → /workspaces/DSIER (main) $ ls -lh
total 4.0K
drwxr-xr-x 3 astronaut astronaut 96 Apr 26 19:03 01-getting-started
drwxr-xr-x 2 astronaut astronaut 64 Apr 26 19:03 02-toolkit
-rw-r--r-- 1 astronaut astronaut 135 Apr 19 15:43 README.md
```

→ start with a dash, usually one letter

→ multiple options can be chained under single dash, sometimes two

```
$ ls -lah 01-getting-started/
$ ls --group-directories-first --human-readable 01-getting-started/
```

- arguments usually on file or directory

man and cheat

- ```
man ls
```

- ```
$ cheat ls
## # Displays everything in the target directory
## ls path/to/the/target/directory
##
## # Displays everything including hidden files
## ls -a
##
## # Displays all files, along with the size (with unit suffixes) and timesta
## ls -lh
##
## # Display files, sorted by size
## ls -S
```

Navigation

- `pwd` to print working directory
- `cd` to change directory

```
astronaut → /workspaces/DSIER (main) $ pwd
/workspaces/DSIER
astronaut → /workspaces/DSIER (main) $ cd ../
astronaut → /workspaces $ pwd
/workspaces
astronaut → /workspaces $
```

Create files and directories

- `touch` and `mkdir`

```
$ mkdir testing
$ touch testing/test1.txt testing/test2.txt testing/test3.txt
$ ls testing
test1.txt  test2.txt  test3.txt
```


Removing files and directories

- `rm`

```
$ rm testing/test1.txt
$ ls testing
test2.txt  test3.txt
$ rm testing
rm: cannot remove 'testing': Is a directory
$ rm -rf testing
$ ls testing
ls: cannot access 'testing': No such file or directory
```

- “recursive” (-r or -R) and “force” (-f) options

Copying

- `cp object path/copyname`

→ keeps old name if not provided with new one

```
$ touch example.txt
$ mkdir testing
$ cp example.txt testing
$ ls testing
example.txt
```

Moving and renaming

- `mv object path/newobjectname`

```
$ mv example.txt testing/example2.txt
$ ls testing
example2.txt  example.txt
$ mv testing/example2.txt testing/example_new.txt
$ ls testing
example_new.txt  example.txt
```

Wildcards

- Wildcards: special characters as replacements for other characters
- Replace any number of characters with *

```
$ cp examples/*.sh examples/copies ## Copy any file with .sh extension  
$ rm examples/copies/* ## Delete everything in the "copies" directory
```

- Replace a single character with ?

```
$ ls examples/meals/??nday.csv  
$ ls examples/meals/?unday.csv  
## examples/meals/monday.csv  
## examples/meals/sunday.csv  
## examples/meals/monday.csv
```

MORE USEFUL COMMANDS

Working with text files

- Print whole file with `cat` (“concatenate”)

```
$ cat -n examples/sonnets.txt
```

- Print only first or last couple of lines with `head` and `tail`

```
$ head -n 3 examples/sonnets.txt ## First 3 rows  
$ tail -n 1 examples/sonnets.txt ## Last row
```

Working with text files

- Search within files: `grep` (“Global regular expression print”)

```
$ wc examples/sonnets.txt  
2633 17698 95662 examples/sonnets.txt
```

```
$ grep -n "Shall I compare thee" examples/sonnets.txt
```

Redirect

- Send output from the shell to a file using redirect operator `>`

```
$ echo "At first, I was afraid, I was petrified" > survive.txt
$ find survive.txt
survive.txt
```

- To append file, use `>>` (`>` overwrites)

```
$ echo "'Kept thinking I could never live without you by my side" >> survive.txt
$ cat survive.txt
At first, I was afraid, I was petrified
'Kept thinking I could never live without you by my side
```


Pipes

- Awesome feature: send (“pipe”) output to another command with `|`
→ chain together a sequence of simple operations

```
$ cat -n examples/sonnets.txt | head -n100 | tail -n10
```

Compress and decompress

- Compress data with `zip` and decompress with `unzip`

```
$ zip archive.zip examples/sonnets.txt
adding: examples/sonnets.txt (deflated 59%)
```

```
$ unzip -l archive.zip
Archive:  archive.zip
  Length      Date    Time    Name
-----
  95662  2022-04-26  20:18   examples/sonnets.txt
-----
  95662
          1 file
```

```
$ unzip archive.zip -d examples
Archive:  archive.zip
  inflating: examples/examples/sonnets.txt
```

LOOPS AND SCRIPTING

Loops

- Repeat operation over set: Loops

```
for i in LIST
do
  OPERATION $i
done
```

- Example: Combing csv files

```
$ touch examples/meals/mealplan.csv
## loop over the input files and append their contents to our new CSV
$ for i in $(ls examples/meals/*day.csv)
> do
>   cat $i >> examples/meals/mealplan.csv
> done
```

Scripting

- .sh file with code can be executed

```
#!/bin/sh
echo -e "\nHello World!\n"
```

- `#!/bin/sh` is a shebang, indicating which program to run the command with
→ `-e` flag tells bash that we want to evaluate an expression rather than a file

```
$ examples/hello.sh
Hello World!
```

- Not limited to running shell scripts in the shell
- Example: `Rscript`

```
$ Rscript -e 'cat("Hello World, from R!")'  
Hello World, from R!
```

MAKE

Build systems

- Sequence of operations to go from inputs to outputs
 - Define dependencies, targets, and rules
- Avoid unnecessary rule execution
- Many build systems, `make` is a common choice

Makefile Example

```
target ... : prerequisites ...
    recipe
    ...
paper.pdf: paper.tex plot-data.png
    pdflatex paper.tex

plot-%.png: %.csv plot.R
    ./plot.r -i $*.csv -o $@
```

- Targets, dependencies, and rules defined in Makefile
- % is a pattern, matching the same string on left and right
- wildcard * searches your filesystem for matching filenames
- \$@ is an automatic variable that contains the target name

Running make

```
$ make
make: *** No rule to make target 'paper.tex', ...
$ touch paper.tex
$ make
make: *** No rule to make target 'plot-data.png', ...
```

- make checks for rules and dependencies
→ complains if dependencies are missing

Building with make

```
$ cat paper.tex
$ cat plot.r
$ cat data.csv
$ make
./plot.r -i data.csv -o plot-data.png
pdflatex paper.tex
```

- Creates a PDF when all dependencies are satisfied
- Running make again shows it's up to date

make

- Build systems, like make, automate the build process
- Saves time and ensures consistency in complex projects
- Essential tool for managing dependencies and targets

WRAP UP

- So far: Shell, git and Make
- This afternoon: R

02. Toolkit

Data Science for Economists — Summer 2024

Julian Hinz

Bielefeld University