DIS 3.3 · DATA INTEGRATION

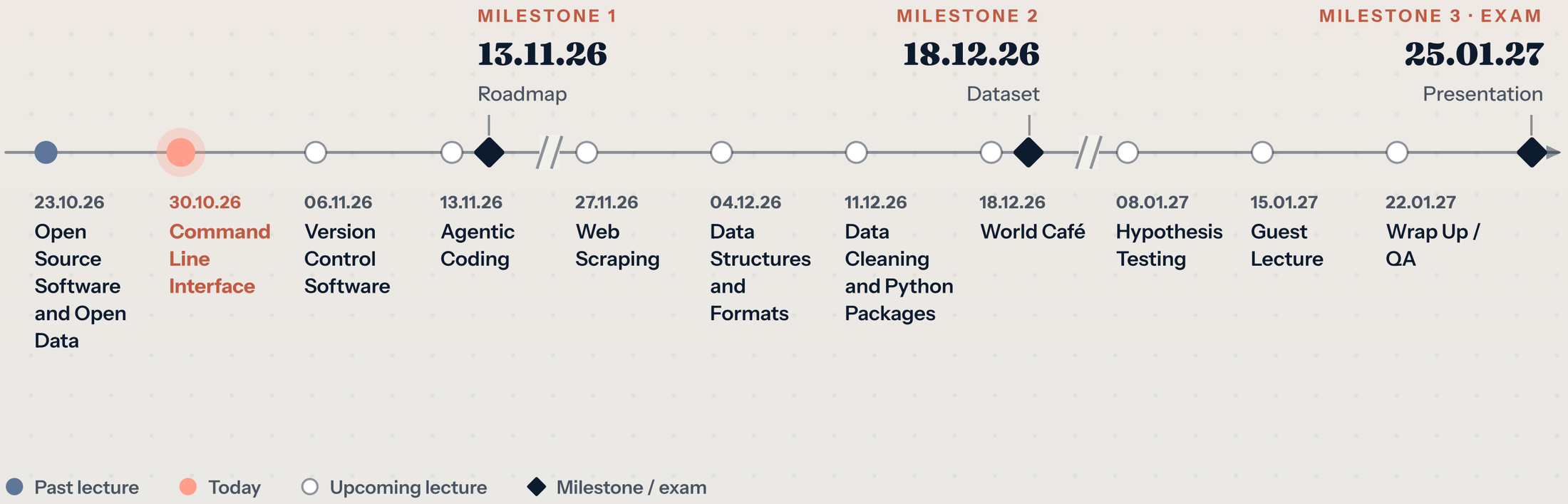
Command line.

The command line interface (CLI) and the shell

Winter 2026/27 · Timo Breuer · TH Köln

```
$ pwd
/Users/timo/dis/shell-lesson
$ wc -l *.tsv
   13712 2014-01-31_JA-africa.tsv
   27392 2014-01-31_JA-america.tsv
  507732 2014-01_JA.tsv
    5375 2014-02-02_JA-britain.tsv
  554211 total
$ wc -l *.tsv | sort -n | head -n 1
    5375 2014-02-02_JA-britain.tsv
```

Timeline


MILESTONE 1
13.11.26
Roadmap
MILESTONE 2
18.12.26
Dataset
MILESTONE 3 · EXAM
25.01.27
Presentation
23.10.26
Open Source Software and Open Data
30.10.26
Command Line Interface
06.11.26
Version Control Software
13.11.26
Agentic Coding
27.11.26
Web Scraping
04.12.26
Data Structures and Formats
11.12.26
Data Cleaning and Python Packages
18.12.26
World Café
08.01.27
Hypothesis Testing
15.01.27
Guest Lecture
22.01.27
Wrap Up / QA
Past lecture
Today
Upcoming lecture
Milestone / exam

Why should I use the command line?

01

Efficiency

Faster than a GUI for repetitive tasks: one command can process a thousand files.

02

Control

More control and flexibility, especially for complex tasks like scripting and automation.

03

Essential skill

Many development and system administration tasks are done on the command line.

04

Foundation

CLI skills are the basis for learning programming languages and tools.

> **In data projects, commands are reproducible: you can write them down, share them and run them again. A click cannot.**

Tools you will meet in this course

`git` `python` `pip` `curl` `ssh` `coding agents`

Scope of today's lecture and roadmap

01

What is the CLI?

02

Command syntax

03

Navigating the file system

04

Files & directories

05

Working with text files

06

Basic scripting

07

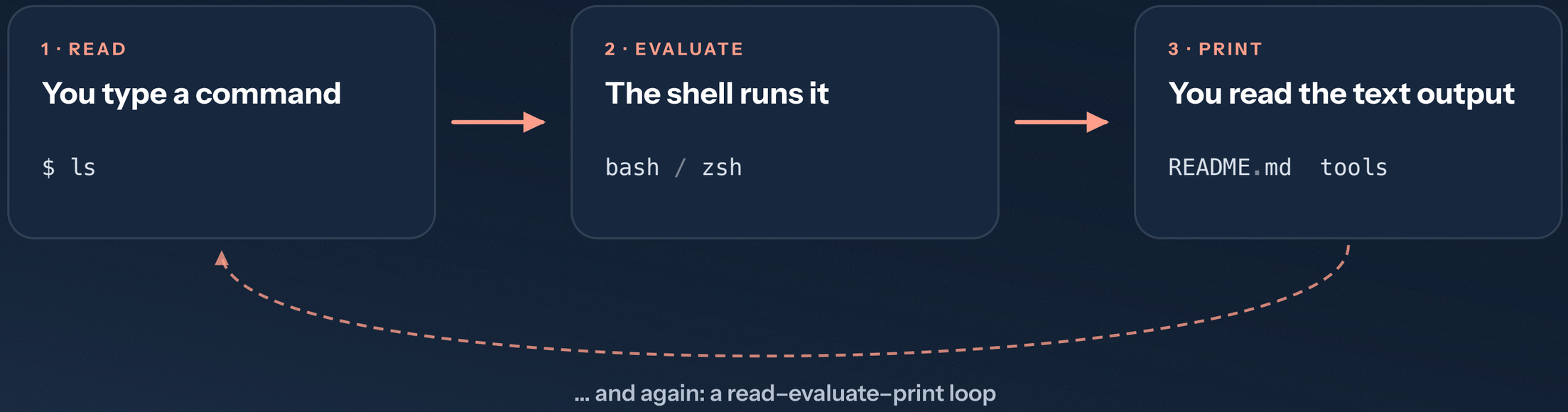
System information

08

Networking basics

What is the command line interface?

> **A text-based user interface: you type a command, the computer runs it and answers with text.**

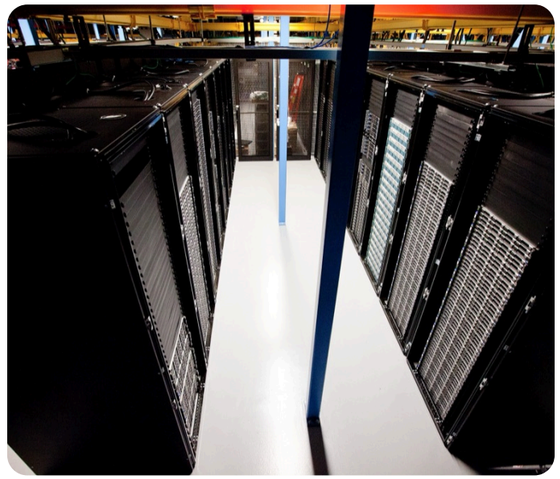


A brief history



Teleprinters (TTY): commands typed in, output printed on paper

- Originated in the early **1960s** with mainframes and minicomputers.
- Early interfaces were **teletypes**: you typed a command and the answer was printed.
- **UNIX** (1970s) made the CLI popular in academia and industry.
- Still essential for system administration, programming and software development.



Today: servers are administered on the command line, often remotely via SSH

1960s — Teletypes at mainframes

1971 — First Unix shell

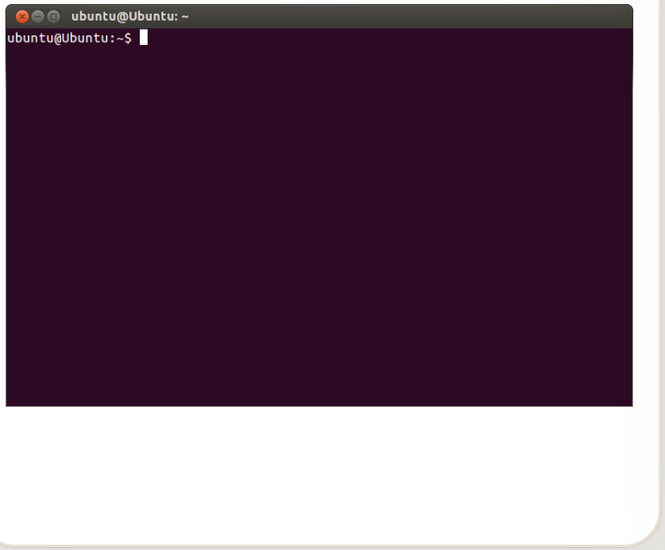
1979 — Bourne shell `sh`

1989 — Bash, the “Bourne-again shell”

2019 — zsh becomes default on macOS

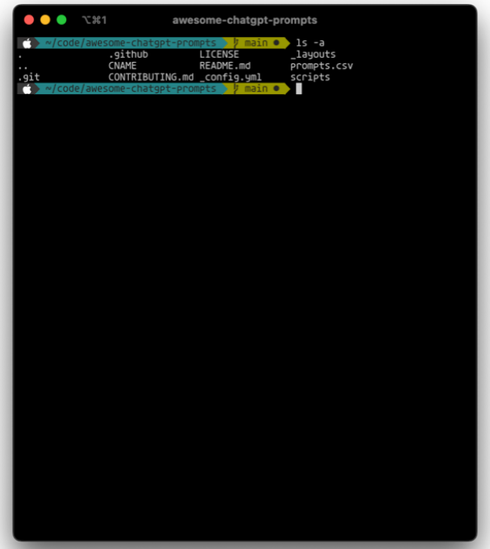
Image sources: en.wikipedia.org/wiki/Teleprinter · en.wikipedia.org/wiki/Web_server

Different flavours, same commands



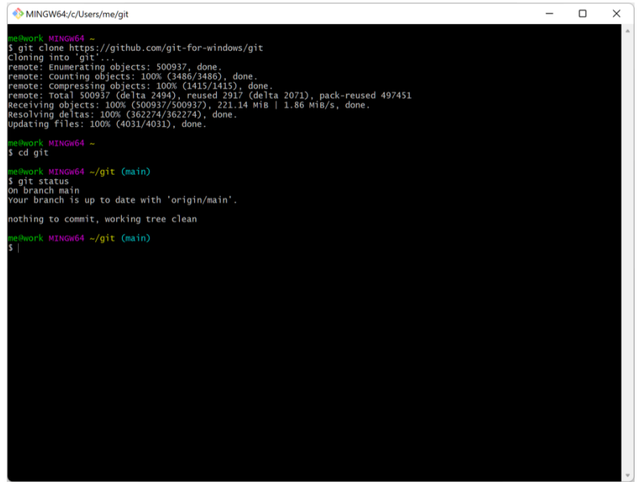
Terminal

Ubuntu Linux · bash



iTerm2

macOS · zsh



Git Bash

Windows · bash

What is a terminal, what is a shell?



The window around it all is the **terminal**.

- **Terminal**
  The window for text input and output: Terminal, iTerm2, Windows Terminal. Once a physical device.
- **Shell**
  The command line interpreter that reads and runs your commands: bash, zsh, PowerShell.
- **Prompt**
  Shows that the shell is ready, often with your location: `~/dis $`
- **Command line**
  The space right of the prompt where you type.
- **Console**
  Historically the physical terminal of a computer; today mostly a synonym for terminal.

After: askubuntu.com/questions/506510 — difference between terminal, console, shell and command line

Which shell on which system?

macOS

APP
Terminal or iTerm2

SHELL
zsh (default since 2019); bash is installed, too

✓ Ready to go

Linux

APP
Terminal

SHELL
bash on most distributions

✓ Ready to go

Windows

APP
Windows Terminal

SHELL
`cmd.exe` does not understand Bash commands; PowerShell has its own

→ Install Git Bash or use WSL

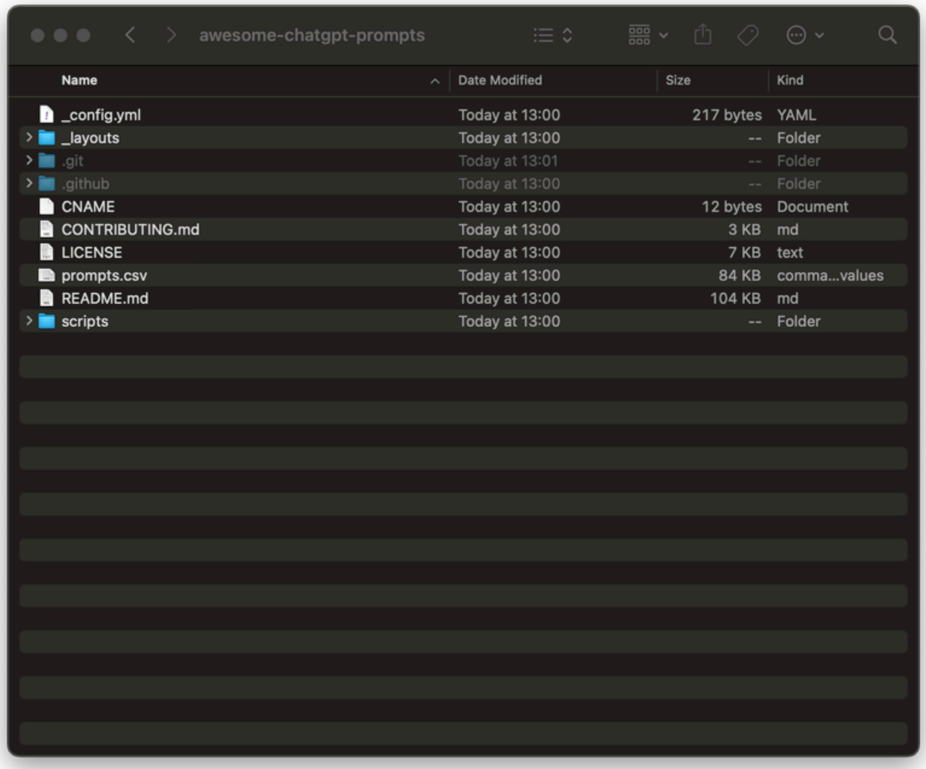
We use Bash commands in this course. On Windows, install **Git Bash** from **gitforwindows.org** — you need Git next week anyway. Alternatives: **WSL** (Windows Subsystem for Linux), Cygwin or a Linux virtual machine.

Source: github.com/irgroup-classrooms/awesome-data-modeling — How to use BASH on Windows?

Comparison of the CLI with the GUI

awesome-chatgpt-prompts

```
~/code/awesome-chatgpt-prompts $ ls -a
.               CNAME           _config.yml
..              CONTRIBUTING.md _layouts
.git            LICENSE         prompts.csv
.github         README.md       scripts
~/code/awesome-chatgpt-prompts $
```



Same folder, two interfaces.

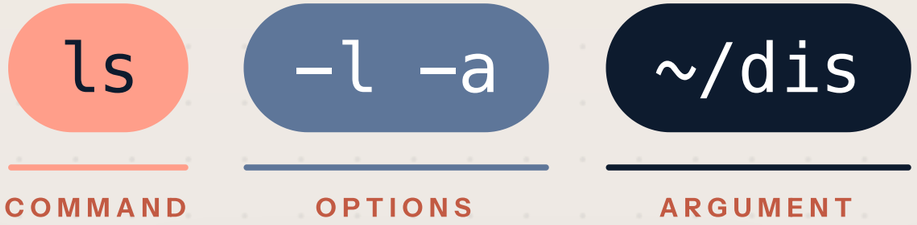
- Files starting with a dot are **hidden**: `ls -a` shows them, Finder only with ⌘ ⇧ .
- The terminal shows exactly what is there — and its output is text you can process further.

Comparison of the CLI with the GUI

	Command line interface (CLI)	Graphical user interface (GUI)
Interaction	Text-based: type commands	Visual: click buttons, icons and menus
Efficiency	Fast for repetitive and complex tasks	Intuitive for beginners and occasional tasks
Learning curve	Steeper: you need to know commands and syntax	Gentle: menus show what is possible
Control	Fine-grained control over system and programs	Limited to what the interface offers
Resources	Minimal — also works on a remote server via SSH	Higher: needs graphics, can be slower
Automation	Scriptable and reproducible: save commands, rerun them	Clicks are hard to record and repeat
Feedback	Text output	Visual feedback, easy switching between apps

Not either/or: explore with the GUI, automate with the CLI.

Anatomy of a command



Command

The program to run.

- `ls`, `cd`, `wc`, `grep`
- Case-sensitive: `ls` works, `LS` does not.

Options (flags)

Change how the command behaves.

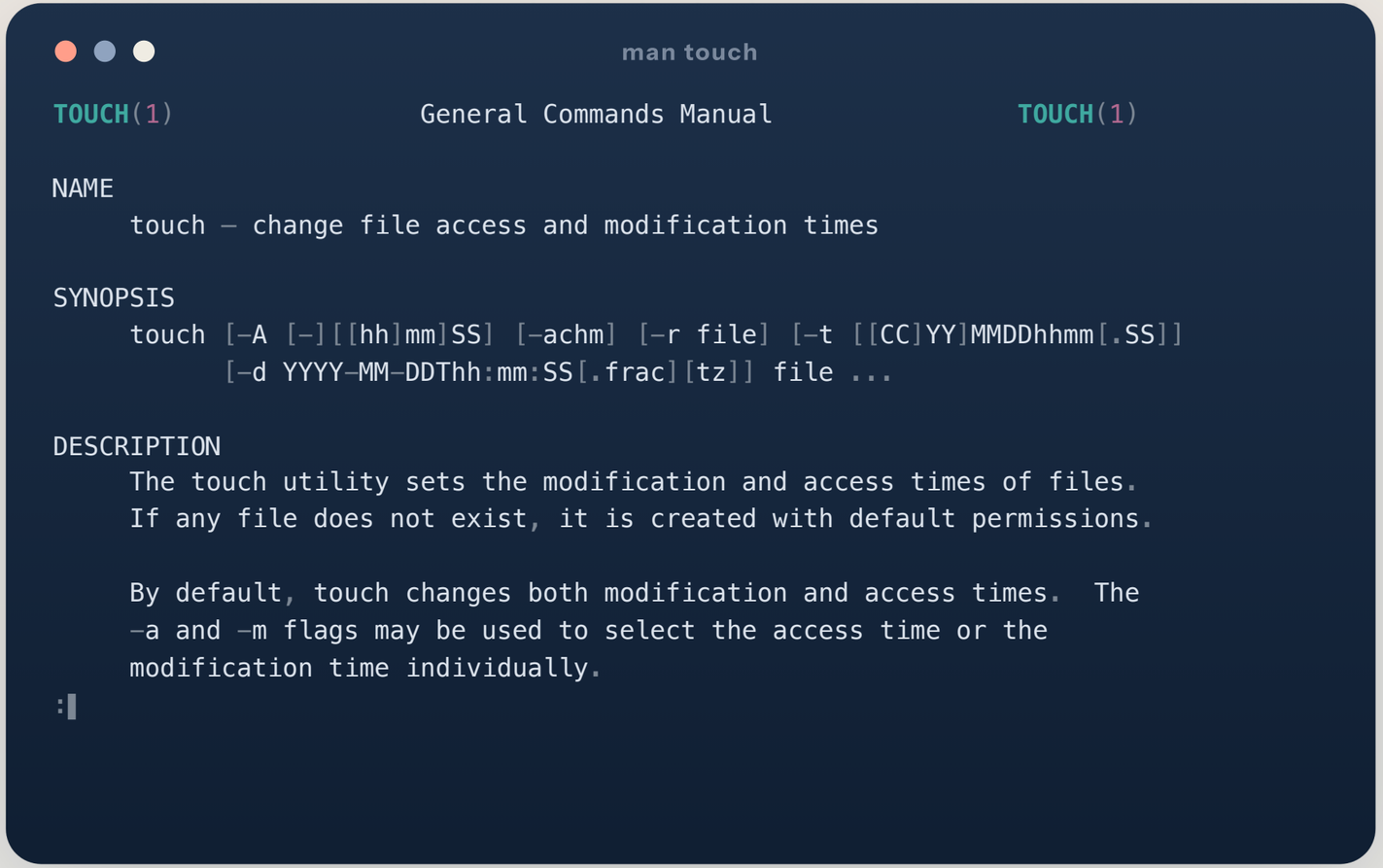
- Short: `-l`, combined: `-la` = `-l -a`
- Long: `--numeric-sort` (not every tool has them)
- With a value: `head -n 5`

Arguments

What the command works on: files, folders, text.

- One: `cd data`
- Two: `cp a.txt b.txt`
- Spaces in names need quotes: `cd "my folder"`

Getting help: man pages



`man touch`
opens the manual page of a command

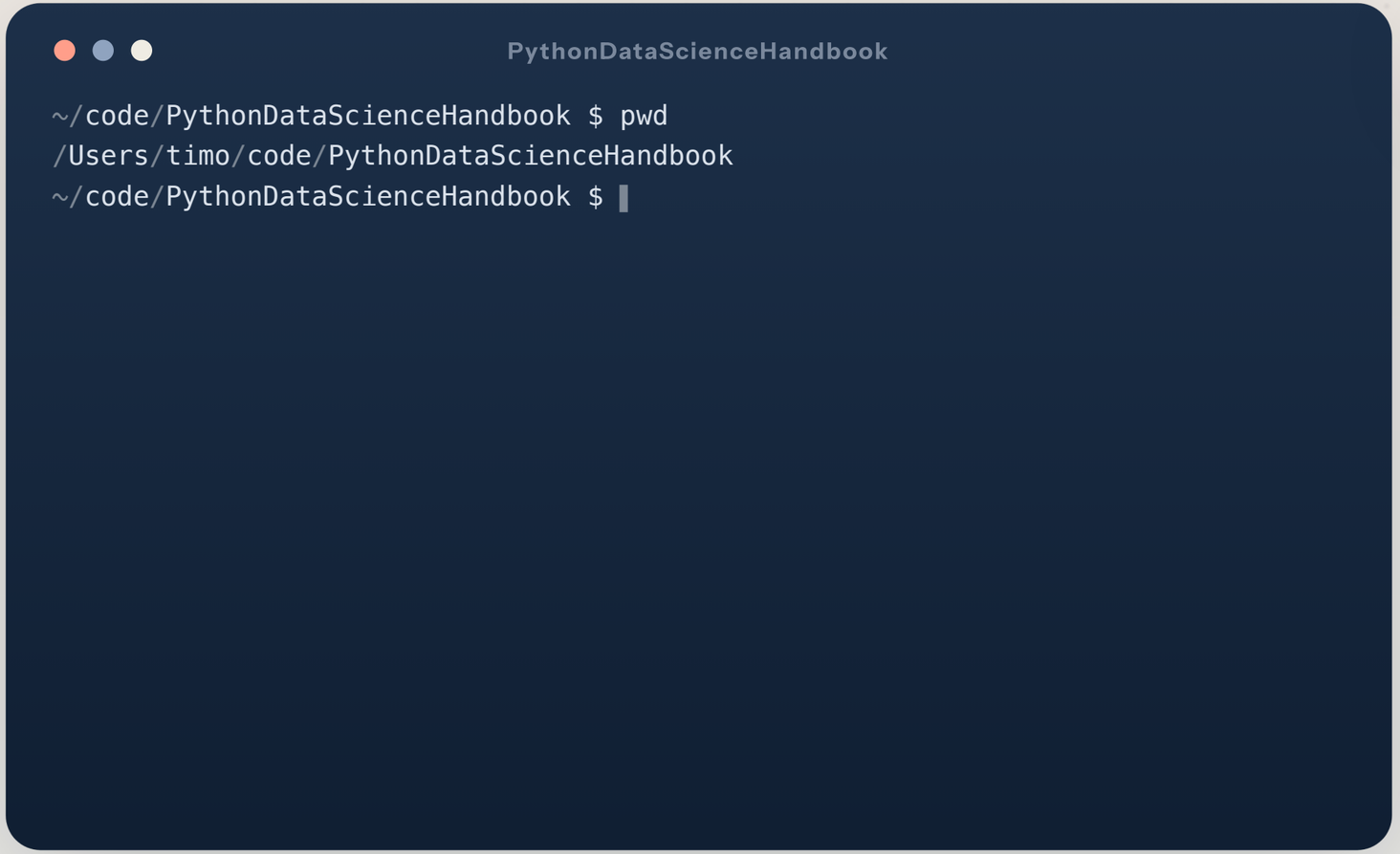
Navigate
↑ ↓ or space to scroll, `/word` to search, `q` to quit

`--help`
Linux and Git Bash: `ls --help` prints a short summary. Git Bash has no `man`.

tldr.sh
community cheat sheets with practical examples

explainshell.com
paste a command, see what each part means

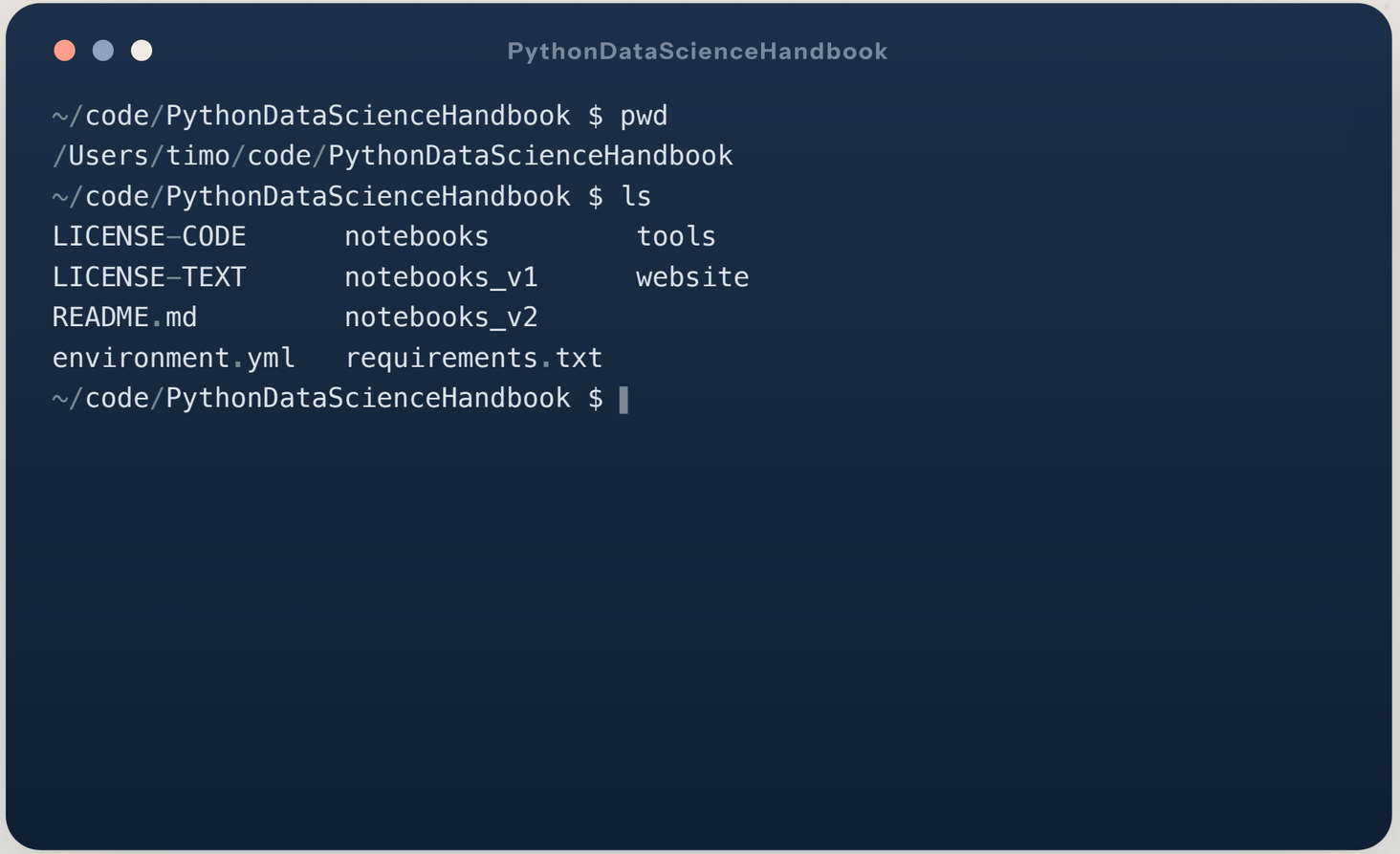
Basic commands for navigation




```
01 pwd
02 ls
03 cd tools
04 ls -l
05 cd ..
```

Print **w**orking **d**irectory: where am I? Knowing where you are is key on the command line!

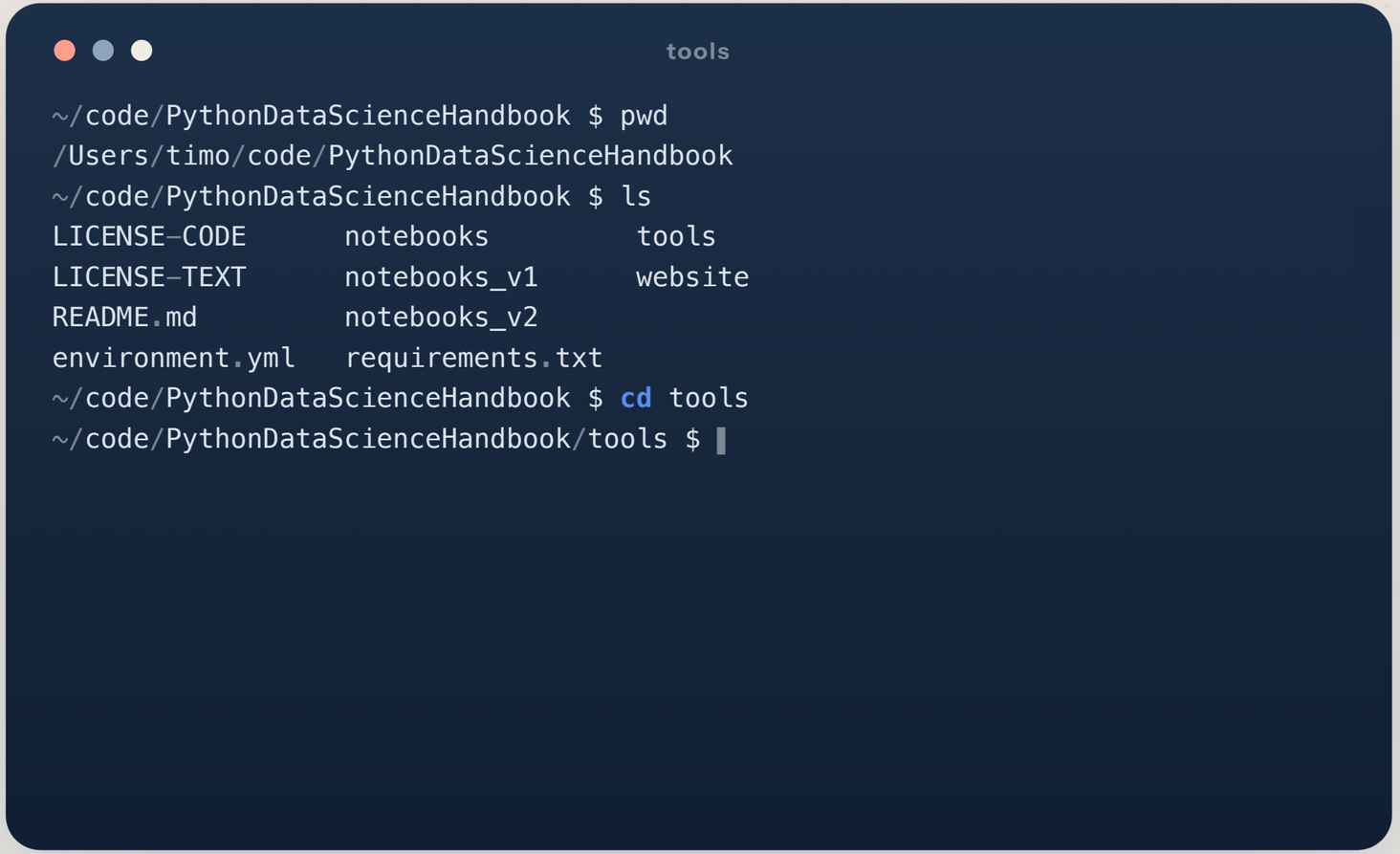
Basic commands for navigation




```
01 pwd
02 ls
03 cd tools
04 ls -l
05 cd ..
```

List the files and directories in the current directory.

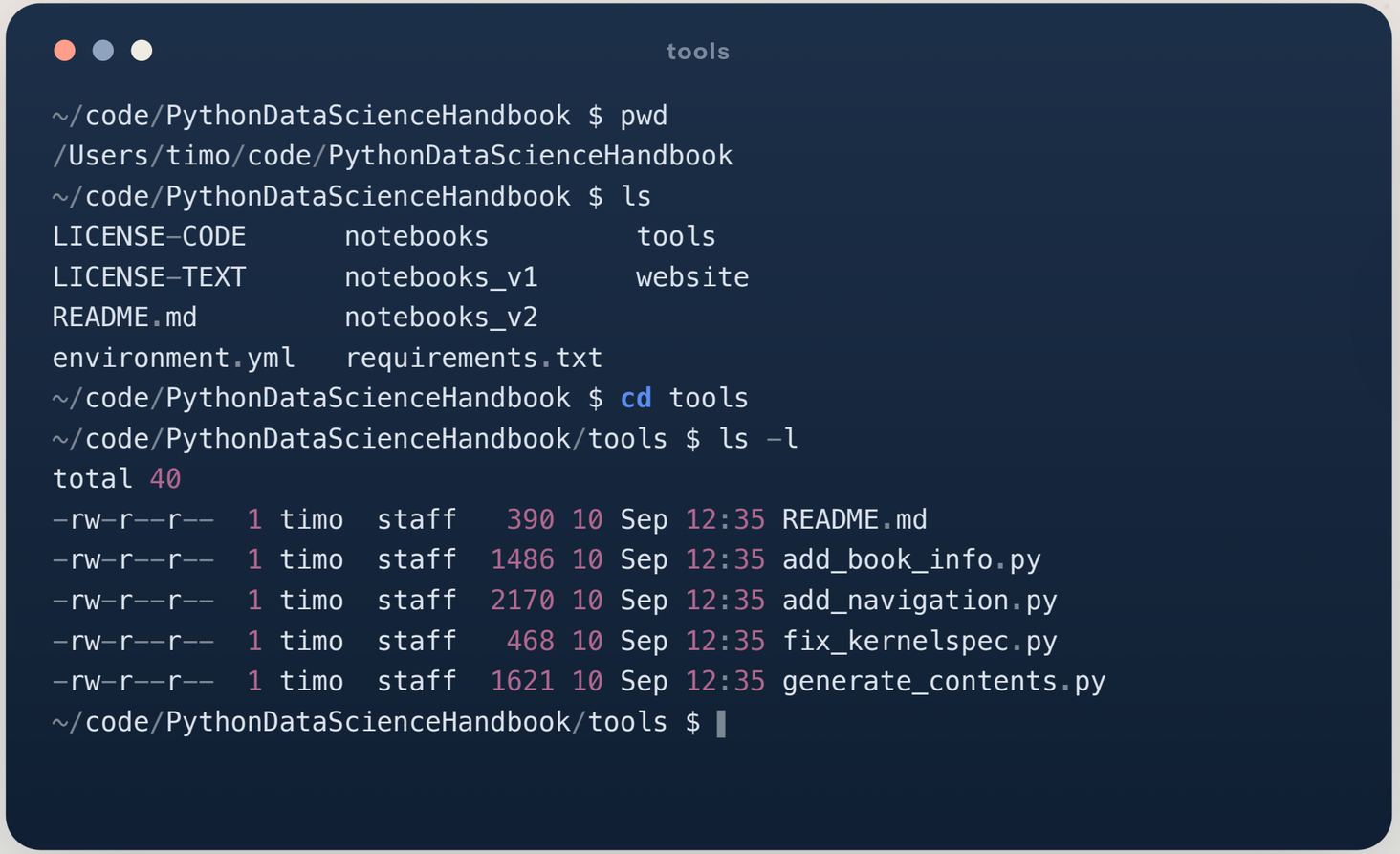
Basic commands for navigation




```
01 pwd
02 ls
03 cd tools
04 ls -l
05 cd ..
```

Change directory into `tools`. The prompt shows the new location.

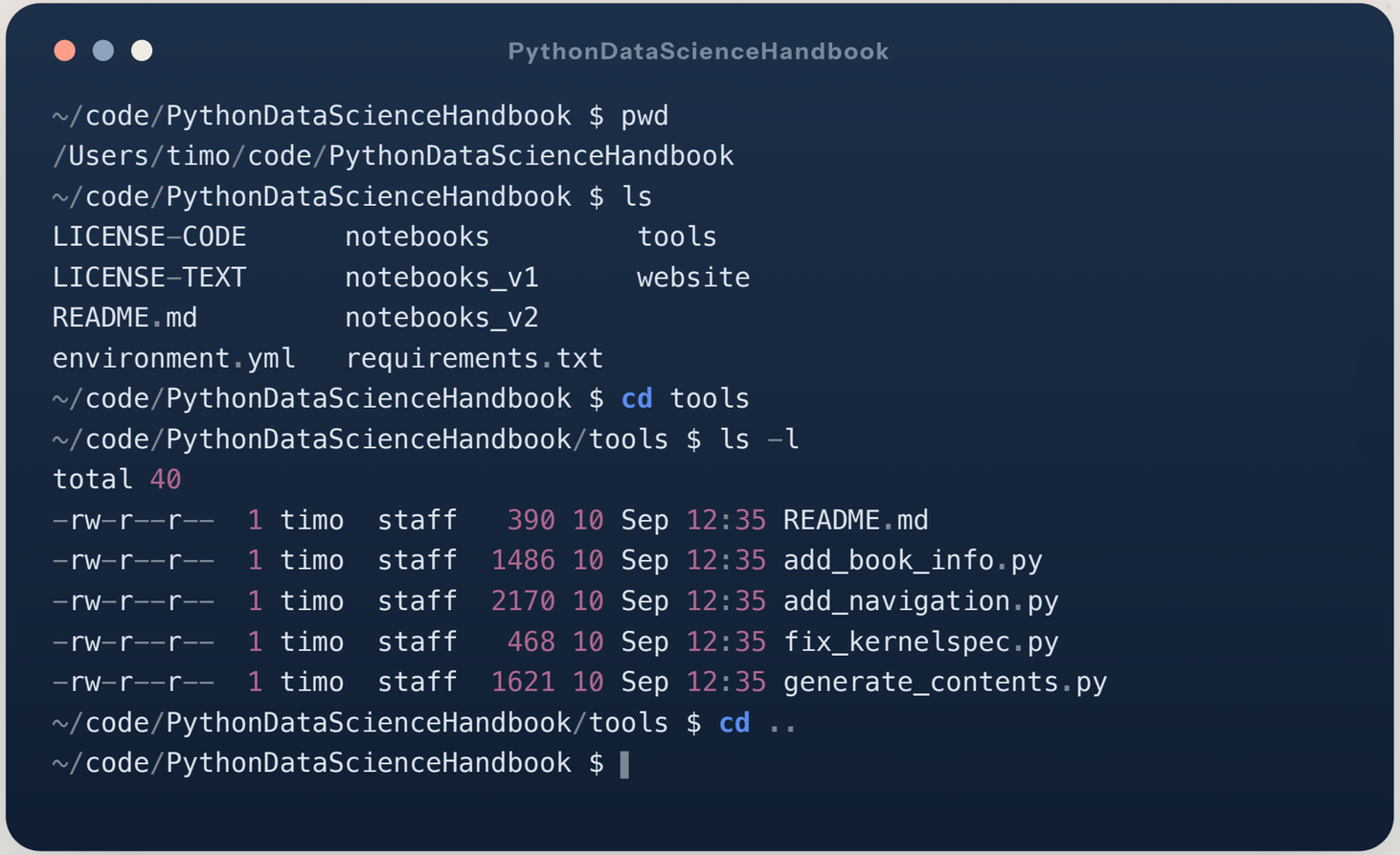
Basic commands for navigation




```
01 pwd
02 ls
03 cd tools
04 ls -l
05 cd ..
```

List in long format: permissions, owner, size and date of the last change.

Basic commands for navigation




```
01 pwd
02 ls
03 cd tools
04 ls -l
05 cd ..
```

`..` is the parent directory: go one level up. `cd` alone takes you home.

Reading the output of ls -l

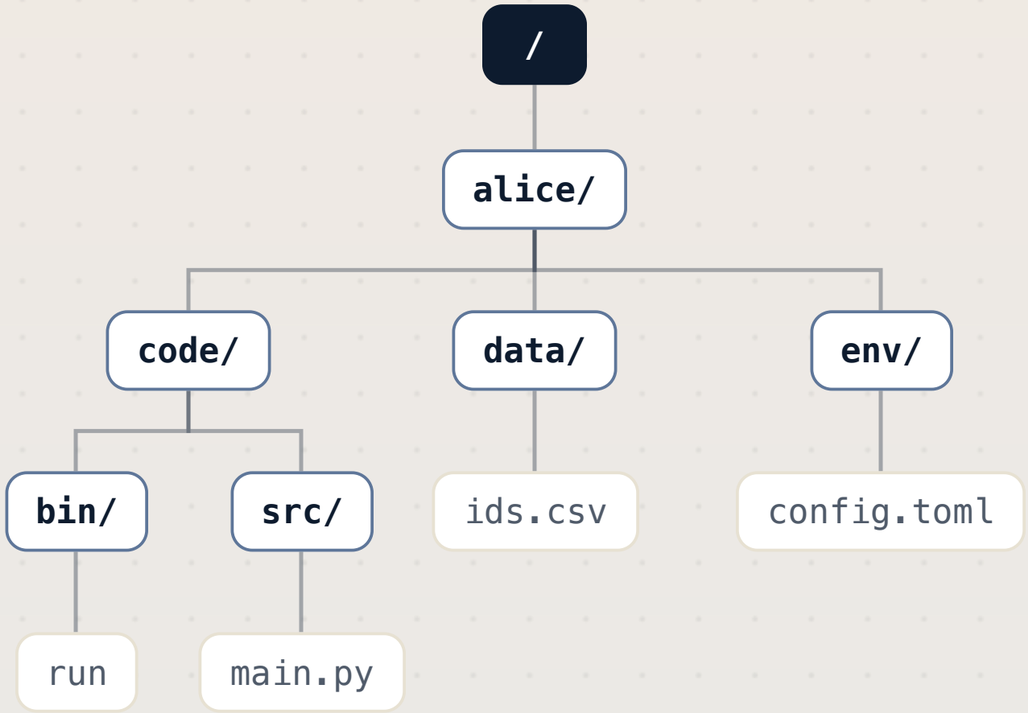


PERMISSIONS: WHO MAY DO WHAT?

rw-	r--	r--
owner	**group**	**others**
read + write	read only	read only

- `r` read · `w` write · `x` execute (run a program, enter a directory)
- First character: `-` file, `d` directory — e.g. `drwxr-xr-x`
- `ls -lh` shows sizes in K, M, G instead of bytes
- Scripts need the `x` permission — we add it later with `chmod +x`

Absolute and relative paths



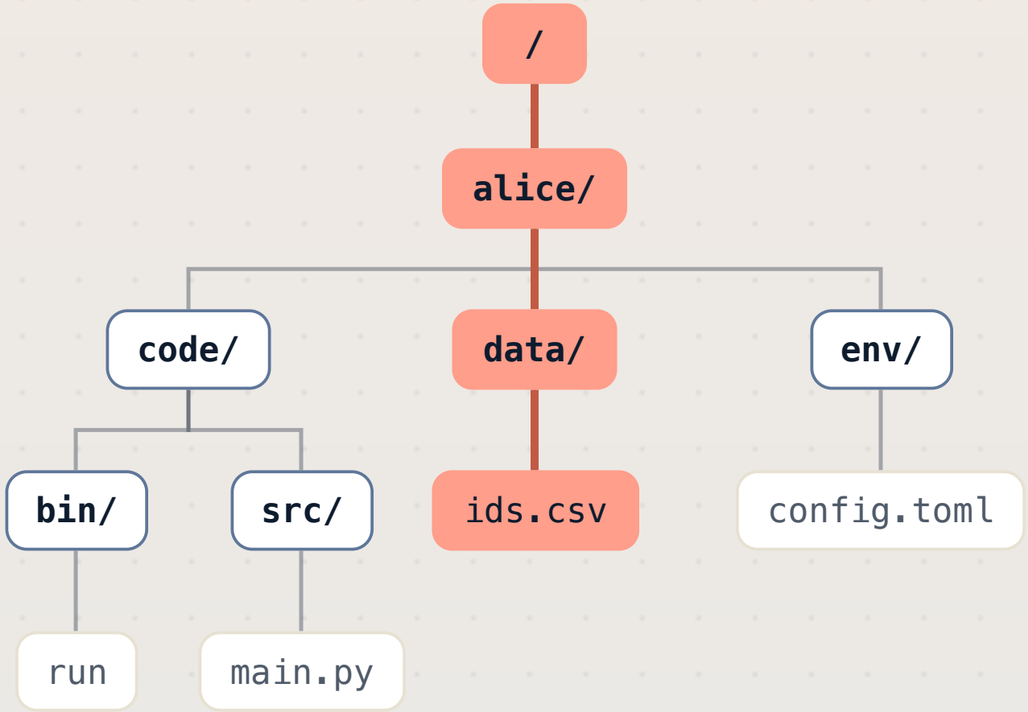
QUESTION

What is the path to `ids.csv`?

You are in `/alice/data`.

Symbol	Meaning
`/`	The root directory: top of the tree — and the separator between names
`~`	Your home directory, e.g. `/Users/timo`
`.`	The current directory
`..`	The parent directory, one level up

Absolute and relative paths

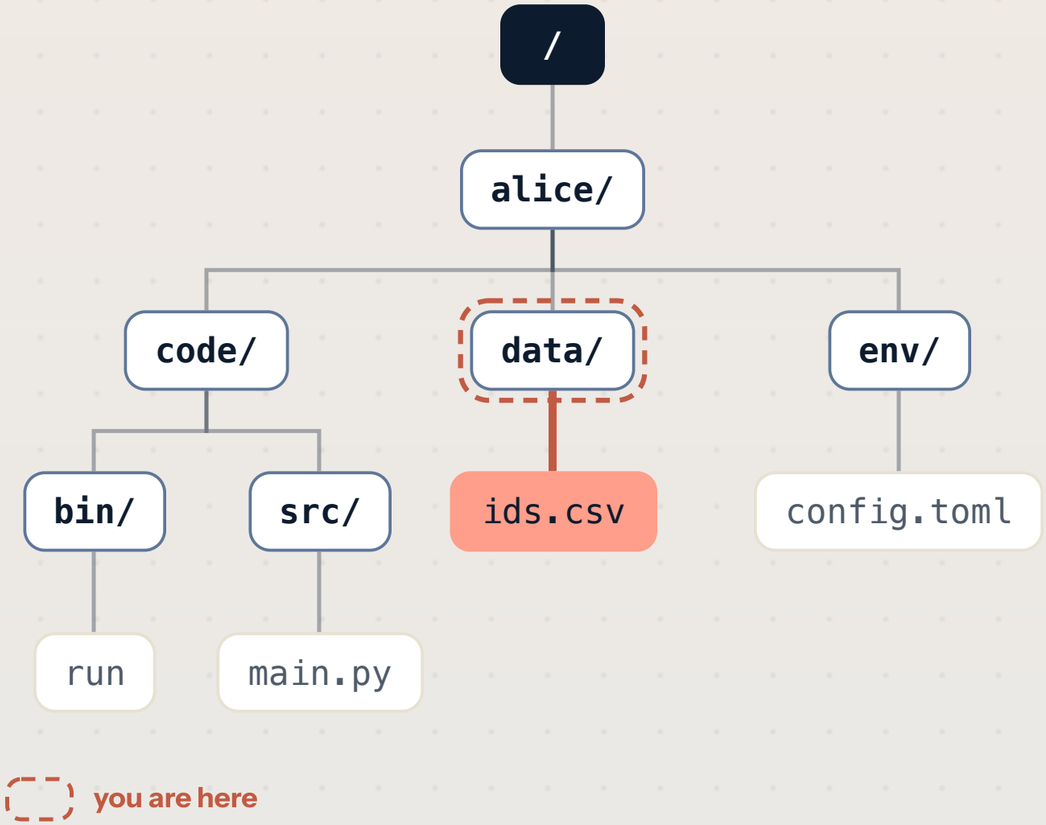


ABSOLUTE PATH

`/alice/data/ids.csv`

- Starts at the root `/`.
- Always points to the same file — no matter where you are.
- Windows: `C:\Users\timo` is `/c/Users/timo` in Git Bash.

Absolute and relative paths

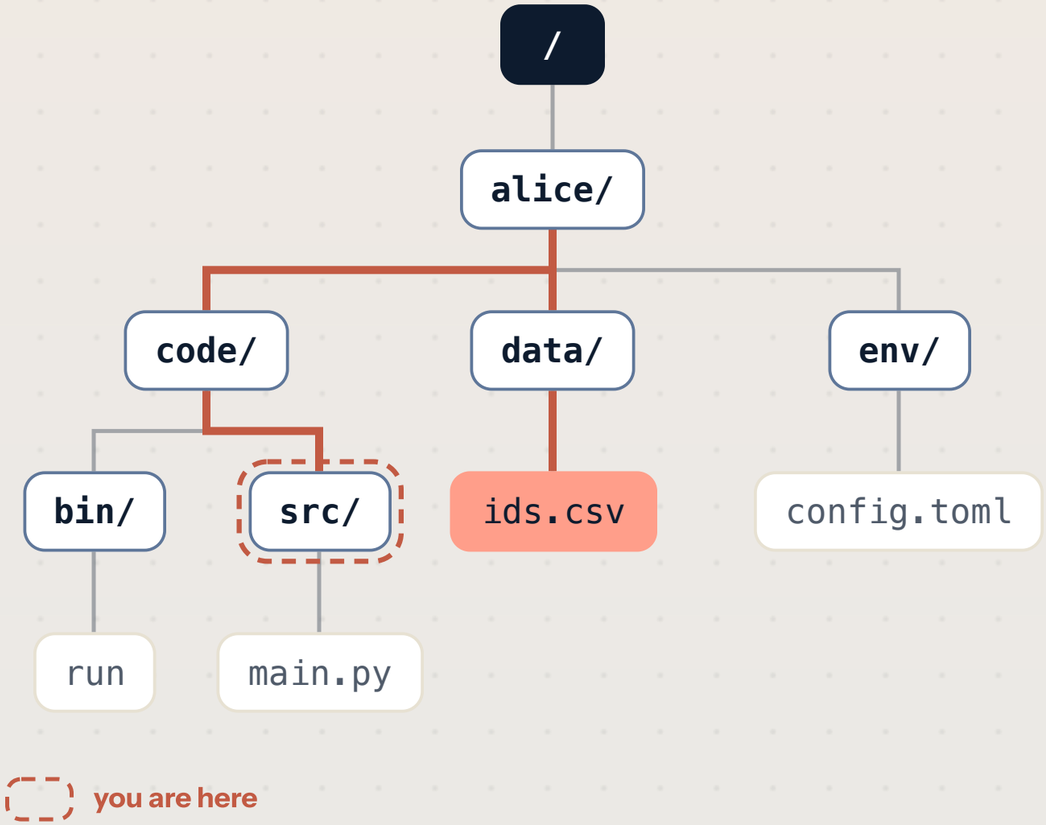


RELATIVE PATH · FROM /alice/data

```
ids.csv
./ids.csv
../data/ids.csv
../../alice/data/ids.csv
```

- All four point to the same file.
- Relative paths start from the current directory: shorter, but they depend on where you are.

Absolute and relative paths



RELATIVE PATH · FROM /alice/code/src

../../data/ids.csv

`..`	up to `code/`
`..`	up to `alice/`
`data/`	down into `data/`
`ids.csv`	the file

Try it: `cd ../../data` and check with `pwd`.

Working with files and directories




```
01  mkdir dis
02  cd dis
03  touch notes.txt
04  echo "hello world" > notes.txt
05  ls > overview.txt
06  rm notes.txt
07  cp overview.txt backup.txt
08  mv backup.txt archive.txt
09  rm -r dis
```

Make a new **dir**ectory called `dis`.

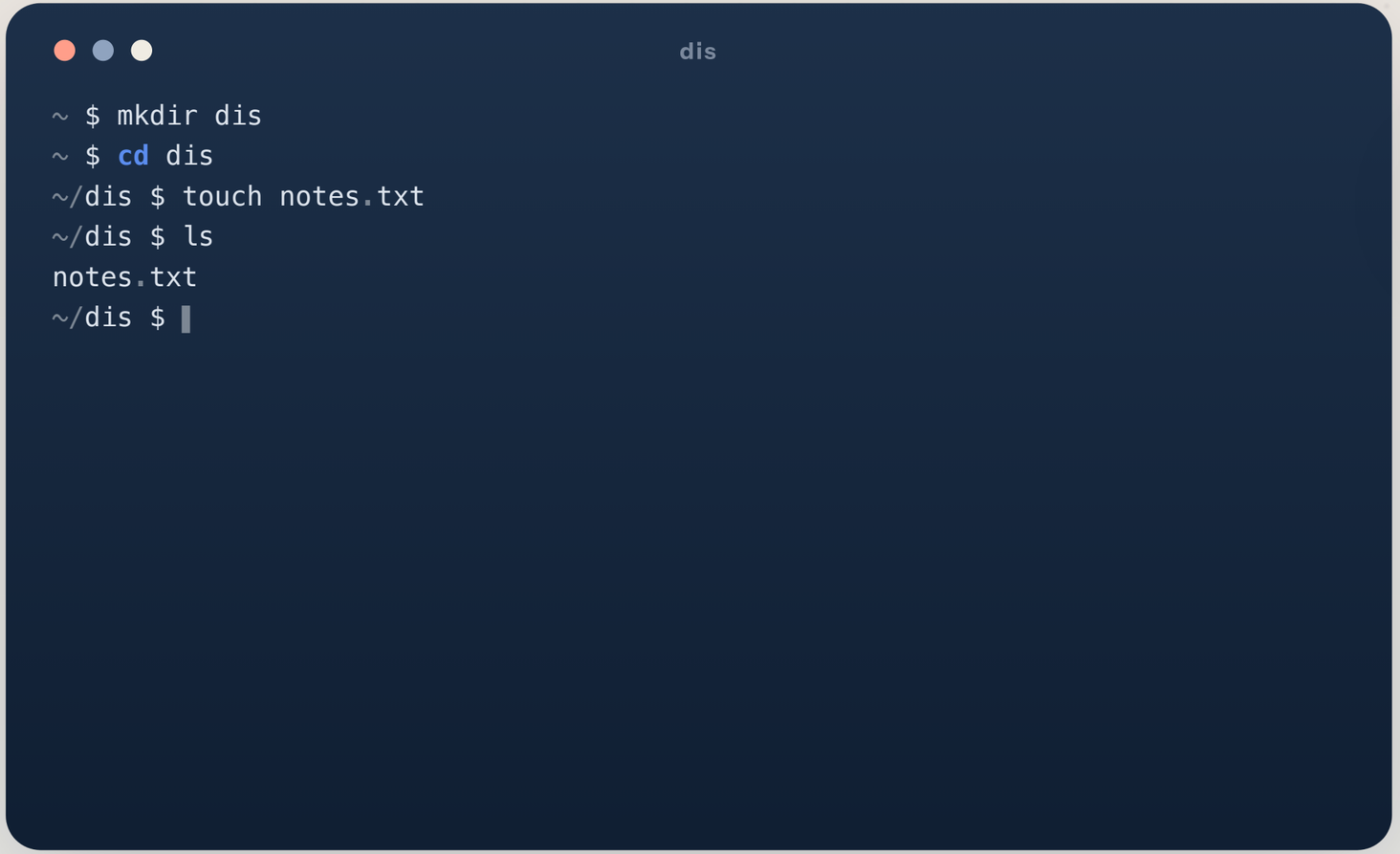
Working with files and directories




```
01  mkdir dis
02  cd dis
03  touch notes.txt
04  echo "hello world" > notes.txt
05  ls > overview.txt
06  rm notes.txt
07  cp overview.txt backup.txt
08  mv backup.txt archive.txt
09  rm -r dis
```

Change into the new directory.

Working with files and directories




```
01  mkdir dis
02  cd dis
03  touch notes.txt
04  echo "hello world" > notes.txt
05  ls > overview.txt
06  rm notes.txt
07  cp overview.txt backup.txt
08  mv backup.txt archive.txt
09  rm -r dis
```

Create an empty file and check with `ls`.

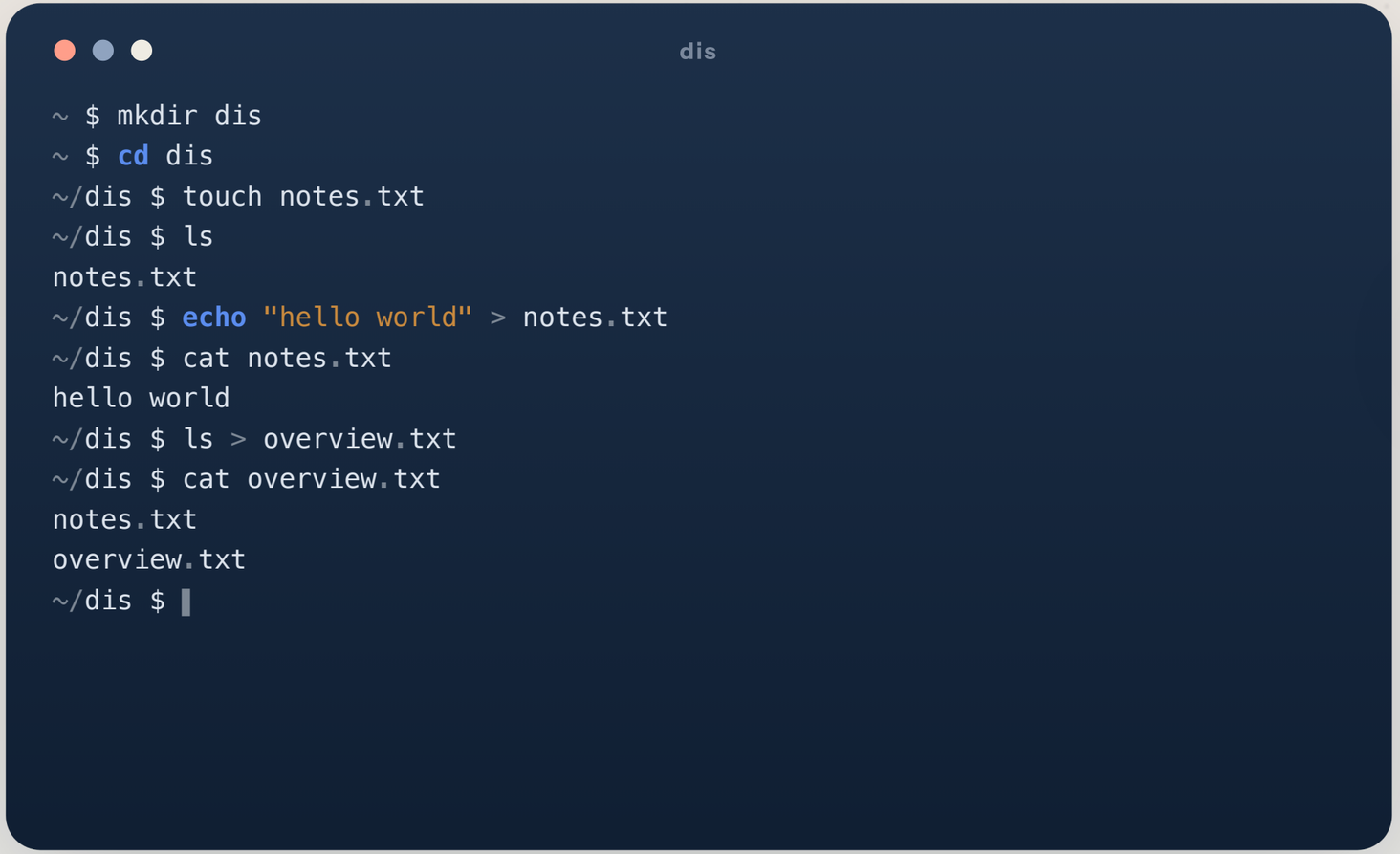
Working with files and directories




```
01  mkdir dis
02  cd dis
03  touch notes.txt
04  echo "hello world" > notes.txt
05  ls > overview.txt
06  rm notes.txt
07  cp overview.txt backup.txt
08  mv backup.txt archive.txt
09  rm -r dis
```

Write a phrase into the file: `>` redirects the output of `echo`. `cat` prints the file.

Working with files and directories




```
01  mkdir dis
02  cd dis
03  touch notes.txt
04  echo "hello world" > notes.txt
05  ls > overview.txt
06  rm notes.txt
07  cp overview.txt backup.txt
08  mv backup.txt archive.txt
09  rm -r dis
```

Save the listing in a file. It lists itself: the shell creates the file before `ls` runs.

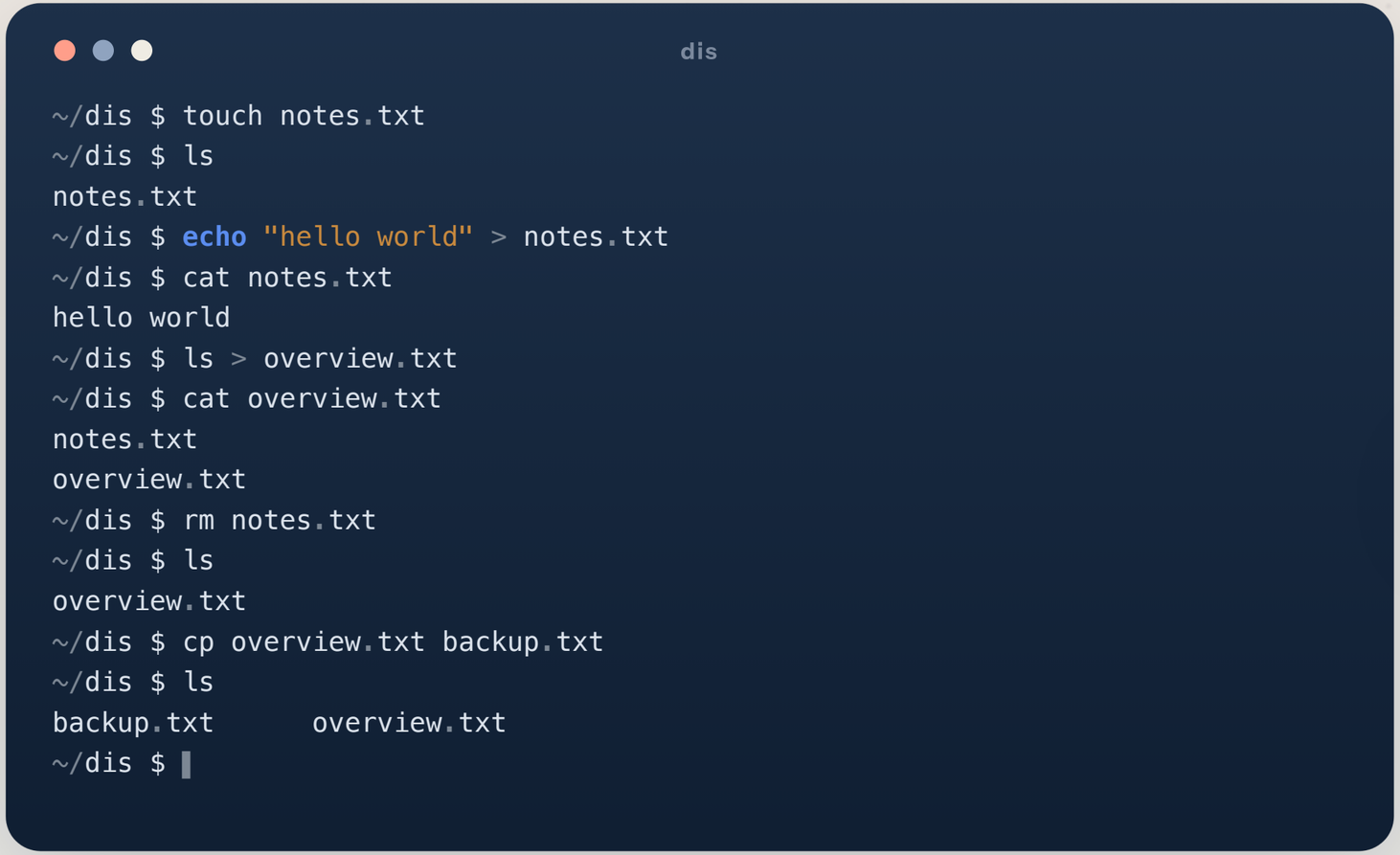
Working with files and directories




```
01  mkdir dis
02  cd dis
03  touch notes.txt
04  echo "hello world" > notes.txt
05  ls > overview.txt
06  rm notes.txt
07  cp overview.txt backup.txt
08  mv backup.txt archive.txt
09  rm -r dis
```

Remove a file — and verify with `ls`.

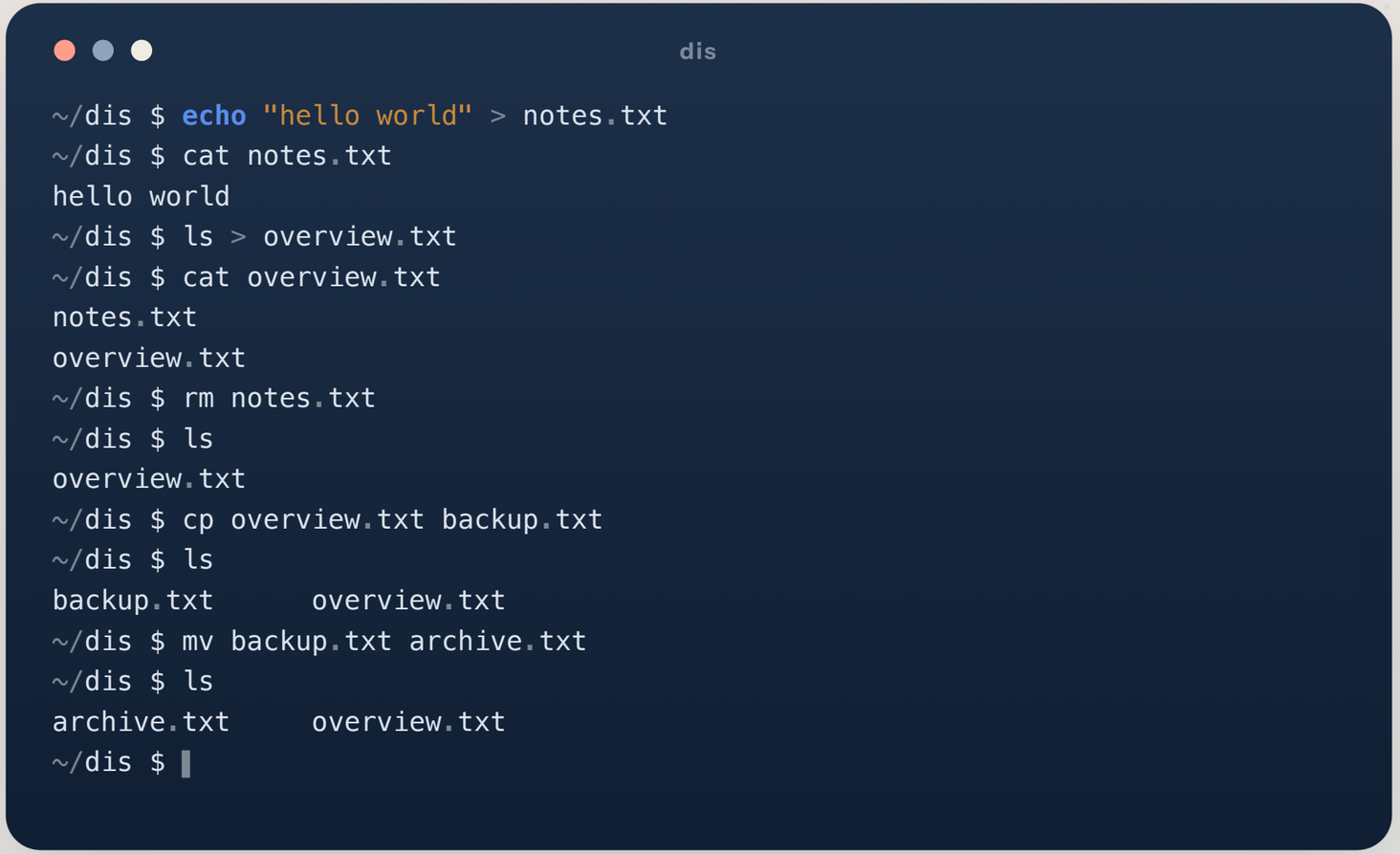
Working with files and directories




```
01  mkdir dis
02  cd dis
03  touch notes.txt
04  echo "hello world" > notes.txt
05  ls > overview.txt
06  rm notes.txt
07  cp overview.txt backup.txt
08  mv backup.txt archive.txt
09  rm -r dis
```

Copy a file.

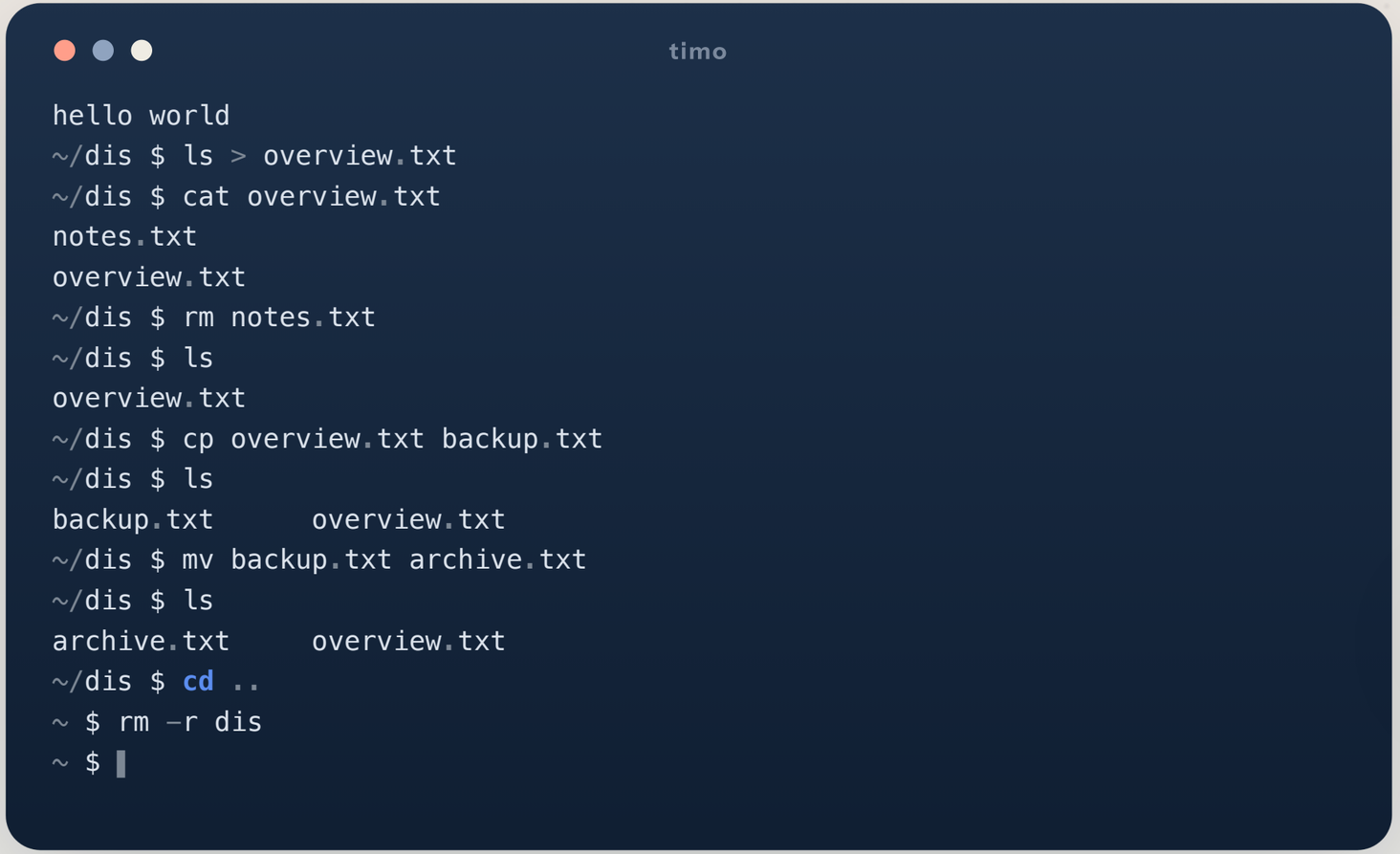
Working with files and directories




```
01  mkdir dis
02  cd dis
03  touch notes.txt
04  echo "hello world" > notes.txt
05  ls > overview.txt
06  rm notes.txt
07  cp overview.txt backup.txt
08  mv backup.txt archive.txt
09  rm -r dis
```

Move or rename a file.

Working with files and directories




```
01  mkdir dis
02  cd dis
03  touch notes.txt
04  echo "hello world" > notes.txt
05  ls > overview.txt
06  rm notes.txt
07  cp overview.txt backup.txt
08  mv backup.txt archive.txt
09  rm -r dis
```

Go up with `cd ..` and remove the whole directory (`-r` = recursive).

Careful: there is no trash can

- `rm` deletes immediately and for good — no undo, no trash.
- `rm -r` deletes a directory with everything inside. `rmdir` only removes empty directories.
- `cp` and `mv` silently overwrite existing files. Add `-i` to be asked first: `rm -i`, `cp -i`, `mv -i`.
- Before you delete: check where you are with `pwd` and what is there with `ls`.



Never ever run

```
sudo rm -rf /
```

… unless you want trouble!

`sudo`	as administrator
`rm -r`	delete recursively
`-f`	force: no questions
`/`	everything, from the root

Cheat sheet: files and directories

Command	What it does	Example
`pwd`	print working directory	`pwd`
`ls`	list files and directories	`ls -la`
`cd`	change directory	`cd data`, `cd ..`, `cd ~`
`mkdir`	make a new directory	`mkdir results`
`touch`	create a new empty file	`touch notes.txt`
`cp`	copy a file (a directory with `-r`)	`cp a.txt b.txt`
`mv`	move or rename a file or directory	`mv a.txt data/`
`rm`	remove a file (a directory with `-r`)	`rm notes.txt`
`rmdir`	remove an empty directory	`rmdir results`
`echo`	print text	`echo "hi" > notes.txt`
`cat`	print the contents of a file	`cat notes.txt`

Getting the sample data

shell-lesson

```
~/dis $ mkdir shell-lesson && cd shell-lesson
~/dis/shell-lesson $ curl -L -O https://librarycarpentry.org/lc-shell/data/shell-lesson.zip
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 34.2M  100 34.2M    0     0  26.1M      0  0:00:01  0:00:01 --:--:-- 32.0M
~/dis/shell-lesson $ unzip -q shell-lesson.zip
~/dis/shell-lesson $ rm -r shell-lesson.zip __MACOSX
~/dis/shell-lesson $ ls
2014-01-31_JA-africa.tsv   2014-02-02_JA-britain.tsv  829-0.txt
2014-01-31_JA-america.tsv  201403160_01_text.json     diary.html
2014-01_JA.tsv             33504-0.txt                pg514.txt
~/dis/shell-lesson $
```

DOWNLOAD

`curl -L -O` saves the file: `-O` keeps its name, `-L` follows the redirect of this link.

UNPACK

`unzip -q` extracts quietly. Then remove the zip and the macOS helper folder `__MACOSX`.

THE DATA

4 TSV files with journal article metadata, 3 books from Project Gutenberg, a JSON and an HTML file.

Data: Library Carpentry, The Unix Shell — librarycarpentry.org/lc-shell

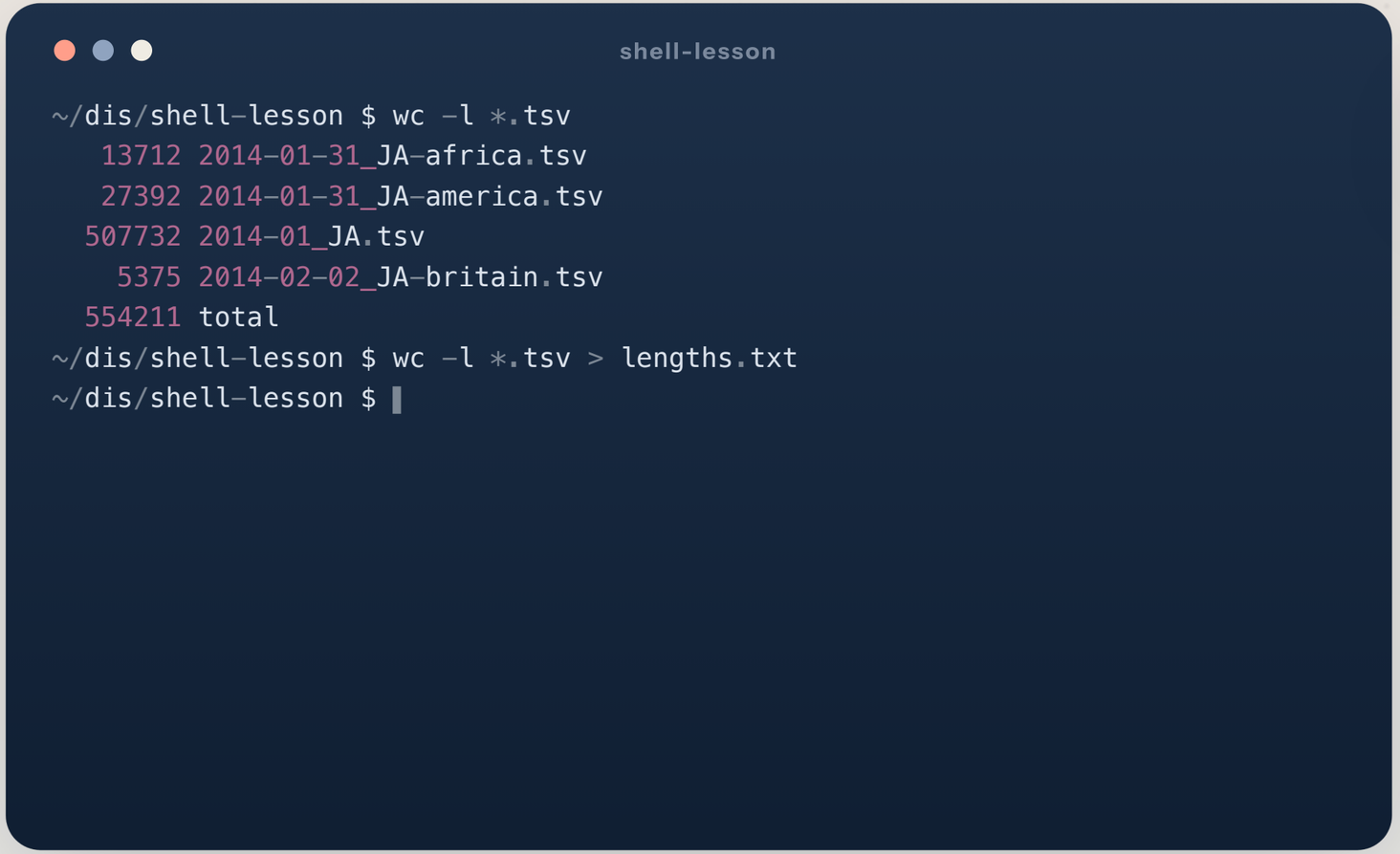
Which file is the shortest?




```
01  wc -l *.tsv
02  wc -l *.tsv > lengths.txt
03  sort -n lengths.txt
04  head -n 1 sorted.txt
05  wc -l *.tsv | sort -n | head -n 1
```

Count the lines (`-l`) of all `.tsv` files. `*` is a wildcard: it matches any characters.

Which file is the shortest?




```
01  wc -l *.tsv
02  wc -l *.tsv > lengths.txt
03  sort -n lengths.txt
04  head -n 1 sorted.txt
05  wc -l *.tsv | sort -n | head -n 1
```

Redirect the output into a file. Nothing is printed — it went into `lengths.txt`.

Which file is the shortest?

shell-lesson

```
~/dis/shell-lesson $ wc -l *.tsv
   13712 2014-01-31_JA-africa.tsv
   27392 2014-01-31_JA-america.tsv
  507732 2014-01_JA.tsv
    5375 2014-02-02_JA-britain.tsv
  554211 total
~/dis/shell-lesson $ wc -l *.tsv > lengths.txt
~/dis/shell-lesson $ sort -n lengths.txt
    5375 2014-02-02_JA-britain.tsv
   13712 2014-01-31_JA-africa.tsv
   27392 2014-01-31_JA-america.tsv
  507732 2014-01_JA.tsv
  554211 total
~/dis/shell-lesson $
```

```
01 wc -l *.tsv
02 wc -l *.tsv > lengths.txt
03 sort -n lengths.txt
04 head -n 1 sorted.txt
05 wc -l *.tsv | sort -n | head -n 1
```

Sort the lines **n**umerically. How do we find the shortest file?

Which file is the shortest?

shell-lesson

```
~/dis/shell-lesson $ wc -l *.tsv
   13712 2014-01-31_JA-africa.tsv
   27392 2014-01-31_JA-america.tsv
  507732 2014-01_JA.tsv
    5375 2014-02-02_JA-britain.tsv
  554211 total
~/dis/shell-lesson $ wc -l *.tsv > lengths.txt
~/dis/shell-lesson $ sort -n lengths.txt
    5375 2014-02-02_JA-britain.tsv
   13712 2014-01-31_JA-africa.tsv
   27392 2014-01-31_JA-america.tsv
  507732 2014-01_JA.tsv
  554211 total
~/dis/shell-lesson $ sort -n lengths.txt > sorted.txt
~/dis/shell-lesson $ head -n 1 sorted.txt
    5375 2014-02-02_JA-britain.tsv
~/dis/shell-lesson $
```

```
01  wc -l *.tsv
02  wc -l *.tsv > lengths.txt
03  sort -n lengths.txt
04  head -n 1 sorted.txt
05  wc -l *.tsv | sort -n | head -n 1
```

Save the sorted list, print its first line: the shortest file. It works — but leaves two helper files behind.

Which file is the shortest?

shell-lesson

```
   27392 2014-01-31_JA-america.tsv
  507732 2014-01_JA.tsv
    5375 2014-02-02_JA-britain.tsv
  554211 total
~/dis/shell-lesson $ wc -l *.tsv > lengths.txt
~/dis/shell-lesson $ sort -n lengths.txt
    5375 2014-02-02_JA-britain.tsv
   13712 2014-01-31_JA-africa.tsv
   27392 2014-01-31_JA-america.tsv
  507732 2014-01_JA.tsv
  554211 total
~/dis/shell-lesson $ sort -n lengths.txt > sorted.txt
~/dis/shell-lesson $ head -n 1 sorted.txt
    5375 2014-02-02_JA-britain.tsv
~/dis/shell-lesson $ wc -l *.tsv | sort -n | head -n 1
    5375 2014-02-02_JA-britain.tsv
~/dis/shell-lesson $
```

```
01 wc -l *.tsv
02 wc -l *.tsv > lengths.txt
03 sort -n lengths.txt
04 head -n 1 sorted.txt
05 wc -l *.tsv | sort -n | head -n 1
```

Can we do better? Yes, with a **pipe**: `|` sends the output of one command straight into the next.

Redirects and pipes

```
$ wc -l *.tsv
```




```
$ wc -l *.tsv > lengths.txt
```




```
$ wc -l *.tsv | sort -n | head -n 1
```

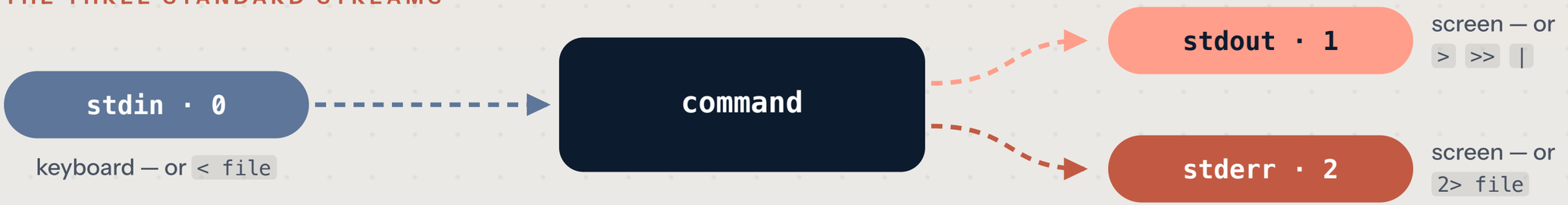


After: librarycarpentry.org/lc-shell/05-counting-mining.html

Redirection: where input and output go

Operator	Meaning	Example
`>`	write the output to a file — **overwrites** it	`ls > files.txt`
`>>`	append the output to the end of a file	`echo "done" >> log.txt`
`<`	read the input from a file	`wc -l < notes.txt`
`2>`	write error messages to a file	`ls xyz 2> errors.txt`
`|`	pipe the output into the next command	`ls | wc -l`

THE THREE STANDARD STREAMS



Wildcards: many files at once

shell-lesson

```
~/dis/shell-lesson $ ls *.tsv
2014-01-31_JA-africa.tsv   2014-01_JA.tsv
2014-01-31_JA-america.tsv  2014-02-02_JA-britain.tsv
~/dis/shell-lesson $ ls 829-?.txt
829-0.txt
~/dis/shell-lesson $ ls [0-9]*.txt
33504-0.txt  829-0.txt
~/dis/shell-lesson $ ls 2014-01*
2014-01-31_JA-africa.tsv   2014-01-31_JA-america.tsv  2014-01_JA.tsv
~/dis/shell-lesson $ echo *.txt
33504-0.txt 829-0.txt lengths.txt pg514.txt sorted.txt
~/dis/shell-lesson $
```

Pattern	Matches
`*`	any number of characters — also none
`?`	exactly one character
`[0-9]`	one character from the set or range
`2014-01*`	names that start with `2014-01`

The **shell** expands wildcards **before** the command runs. `echo *.txt` shows what a command receives: a plain list of names — `ls` never sees the `*`.

Searching text with grep

shell-lesson

```
~/dis/shell-lesson $ grep -c revolution *.tsv
2014-01-31_JA-africa.tsv:20
2014-01-31_JA-america.tsv:34
2014-01_JA.tsv:867
2014-02-02_JA-britain.tsv:9
~/dis/shell-lesson $ grep -ci revolution *.tsv
2014-01-31_JA-africa.tsv:118
2014-01-31_JA-america.tsv:1018
2014-01_JA.tsv:9327
2014-02-02_JA-britain.tsv:122
~/dis/shell-lesson $ grep -ci "french revolution" *.tsv
2014-01-31_JA-africa.tsv:0
2014-01-31_JA-america.tsv:13
2014-01_JA.tsv:530
2014-02-02_JA-britain.tsv:4
~/dis/shell-lesson $
```

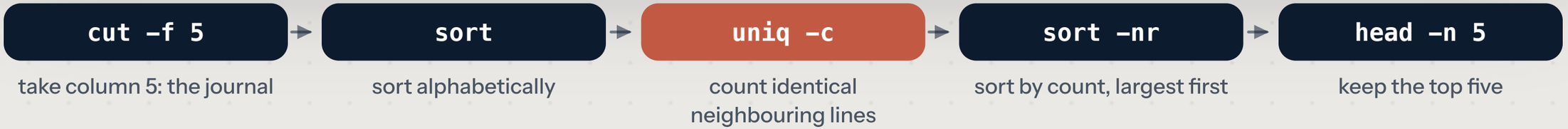
Command	Prints
`grep word file`	lines that contain *word*
`-c`	the number of matching lines
`-i`	ignore upper and lower case
`-w`	whole words only — not *revolutionary*
`-v`	lines that do **not** match
`-n`	line numbers, too

Put phrases with spaces in quotes:
`"french revolution"`

Counting values with cut, sort, uniq

shell-lesson

```
~/dis/shell-lesson $ head -n 1 2014-02-02_JA-britain.tsv | tr "\t" "\n" | head -n 5
File
Creator
Issue
Volume
Journal
~/dis/shell-lesson $ cut -f 5 2014-02-02_JA-britain.tsv | sort | uniq -c | sort -nr | head -n 5
 409 ALBION -ILLINOIS-
 401 ENGLISH HISTORICAL REVIEW
 357 AMERICAN HISTORICAL REVIEW
 234 JOURNAL OF BRITISH STUDIES
 160 CONTEMPORARY BRITISH HISTORY
~/dis/shell-lesson $
```



Cheat sheet: working with text files

Command	What it does	Example
`cat`	concatenate and print text files	`cat notes.txt`
`less`	view a file page by page (`q` quits)	`less pg514.txt`
`more`	older pager: one screen at a time	`more pg514.txt`
`head`	first lines of a file	`head -n 5 data.tsv`
`tail`	last lines of a file	`tail -n 5 data.tsv`
`wc`	count lines, words and characters	`wc -l *.tsv`
`grep`	search for patterns	`grep -i revolution *.tsv`
`cut`	extract columns	`cut -f 5 data.tsv`
`sort`	sort lines (`-n` numerically, `-r` reverse)	`sort -n lengths.txt`
`uniq`	merge repeated neighbouring lines (`-c` counts)	`sort | uniq -c`

Work faster: history and shortcuts

shell-lesson

```
~/dis/shell-lesson $ history
 1068  cd dis
 1069  cd shell-lesson
 1070  pwd
 1071  ls
 1072  wc -l *.tsv
 1073  wc -l *.tsv > lengths.txt
 1074  sort -n lengths.txt
 1075  sort -n lengths.txt > sorted.txt
 1076  head -n 1 sorted.txt
 1077  wc -l *.tsv | sort -n | head -n 1
 1078  grep -c revolution *.tsv
~/dis/shell-lesson $
```

Key	What it does
Tab	complete commands, file and directory names — press twice to see all options
↑ / ↓	browse your previous commands
Ctrl + R	search your command history
Ctrl + C	cancel a running command
Ctrl + L	clear the screen (or type `clear`)
Ctrl + A / E	jump to the start / end of the line

What is a shell script?

WHAT

- A text file with commands that the shell executes one after another.
- Useful to automate file management, running programs and text processing — everything from today.

WHY

- **Automates** repetitive tasks: less typing, less work.
- **Simplifies** complex operations: readability.
- **Increases efficiency**: write once, reuse often.

> **A script is a recipe: same steps, same result — a first step towards reproducible data pipelines.**

Writing a simple shell script

greet.sh

```
#!/bin/bash                                                    1
# greet.sh: greets the user and shows the date and time

# Greet the user
echo "Hello, welcome to the command line!"                     2

# Ask for the user's name
echo -n "Please enter your name: "                             3
read name

# Greet the user by their name
echo "Nice to meet you, $name!"

# Display the current date and time
echo "Today is $(date +"%Y-%m-%d")."
echo "The current time is $(date +"%H:%M:%S")."
echo "Hope you have a great day ahead!"
```

1. **`#!/bin/bash`**

   The “shebang”: which interpreter runs the script — here Bash.

2. **`echo`**

   Prints text to the terminal. `echo -n` prints without a line break.

3. **`read name`**

   Waits for input and stores it in the variable `name`.

Writing a simple shell script

greet.sh

```
#!/bin/bash
# greet.sh: greets the user and shows the date and time

# Greet the user
echo "Hello, welcome to the command line!"

# Ask for the user's name
echo -n "Please enter your name: "
read name

# Greet the user by their name
echo "Nice to meet you, $name!"

# Display the current date and time
echo "Today is $(date +"%Y-%m-%d")."
echo "The current time is $(date +"%H:%M:%S")."
echo "Hope you have a great day ahead!"
```

1. **`$name`**

   Use a variable by putting `$` in front of its name.

2. **`$(date +"%Y-%m-%d")`**

   Command substitution: runs `date` and inserts its output.

3. **`# comment`**

   Lines starting with `#` are ignored by the shell. They explain the code to humans.

Running the script

dis

```
~/dis $ ls -l greet.sh
-rw-r--r--  1 timo  staff  427 30 Oct 10:12 greet.sh
~/dis $ ./greet.sh
bash: ./greet.sh: Permission denied
~/dis $ chmod +x greet.sh
~/dis $ ls -l greet.sh
-rwxr-xr-x  1 timo  staff  427 30 Oct 10:12 greet.sh
~/dis $ ./greet.sh
Hello, welcome to the command line!
Please enter your name: Ada
Nice to meet you, Ada!
Today is 2026-10-30.
The current time is 10:15:42.
Hope you have a great day ahead!
~/dis $
```

`chmod +x`

adds the execute permission — compare the two `ls -l` lines.

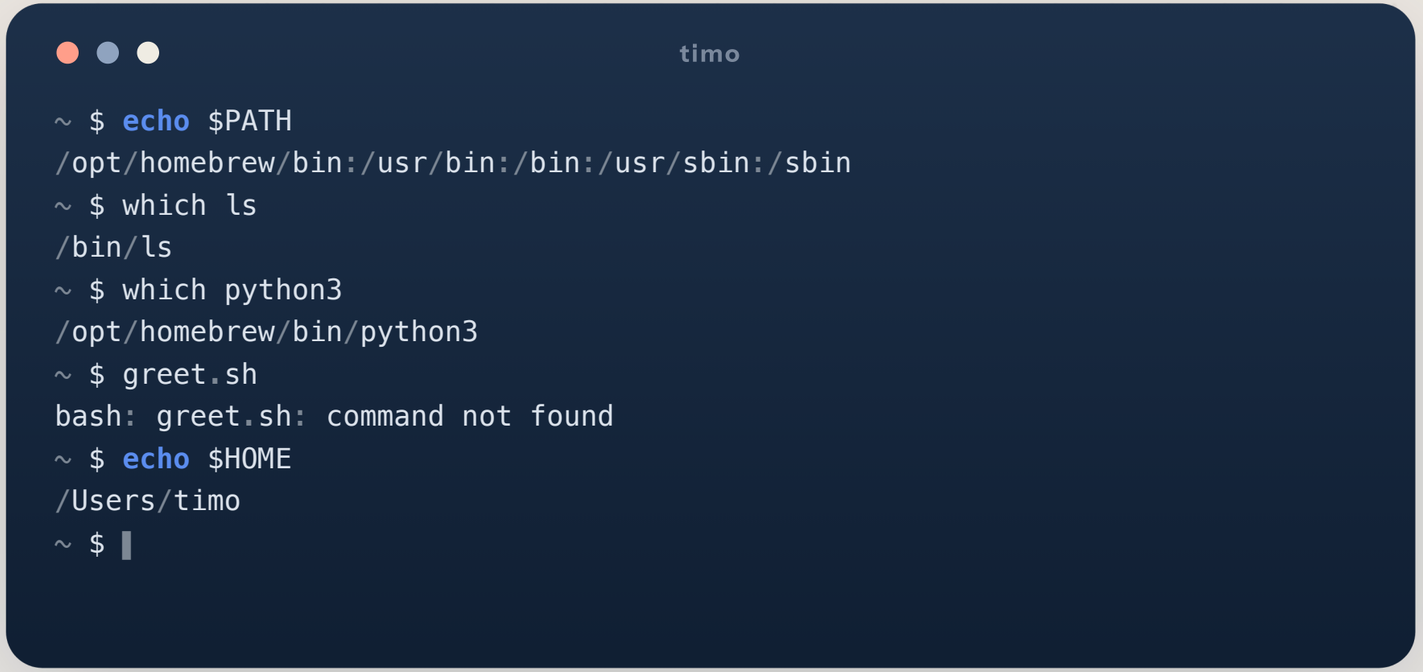
`./greet.sh`

runs the file in the current directory. Without `./` the shell does not find it (next slide).

`bash greet.sh`

also works without `chmod`: you hand the file to Bash directly.

Where do commands come from?



Other useful variables: `$HOME`, `$USER`, `$PWD`. `env` lists them all.

`PATH`

An environment variable: a list of directories, separated by `:`.

Lookup

For `ls`, the shell searches these directories in order and runs the first match.

`which`

shows which program will actually run.

command not found?

The program is not installed — or its directory is not in `PATH`.

Going further: arguments and loops

backup.sh

```bash
#!/bin/bash
# backup.sh: copy all .tsv files of a folder
# into a dated backup folder.
# Usage: ./backup.sh <folder>

src=$1                          # 1st argument
dest="backup_$(date +%Y-%m-%d)"

mkdir -p "$dest"
for file in "$src"/*.tsv; do    # loop
    cp "$file" "$dest"
    echo "copied $file"
done
```

dis

```
~/dis $ chmod +x backup.sh
~/dis $ ./backup.sh shell-lesson
copied shell-lesson/2014-01-31_JA-africa.tsv
copied shell-lesson/2014-01-31_JA-america.tsv
copied shell-lesson/2014-01_JA.tsv
copied shell-lesson/2014-02-02_JA-britain.tsv
~/dis $ ls backup_2026-10-30
2014-01-31_JA-africa.tsv   2014-01_JA.tsv
2014-01-31_JA-america.tsv  2014-02-02_JA-britain.tsv
~/dis $
```

`$1`, `$2`, …

the command-line arguments of the script

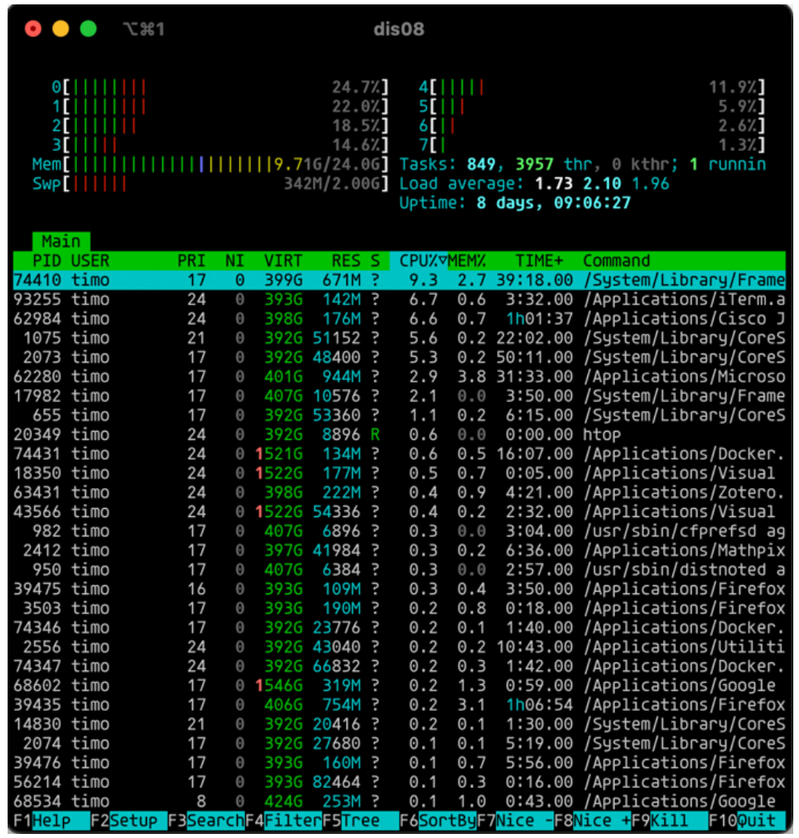
`for … do … done`

repeats the commands for every matching file

Voluntary homework

write both scripts yourself — then automate a small task of your own

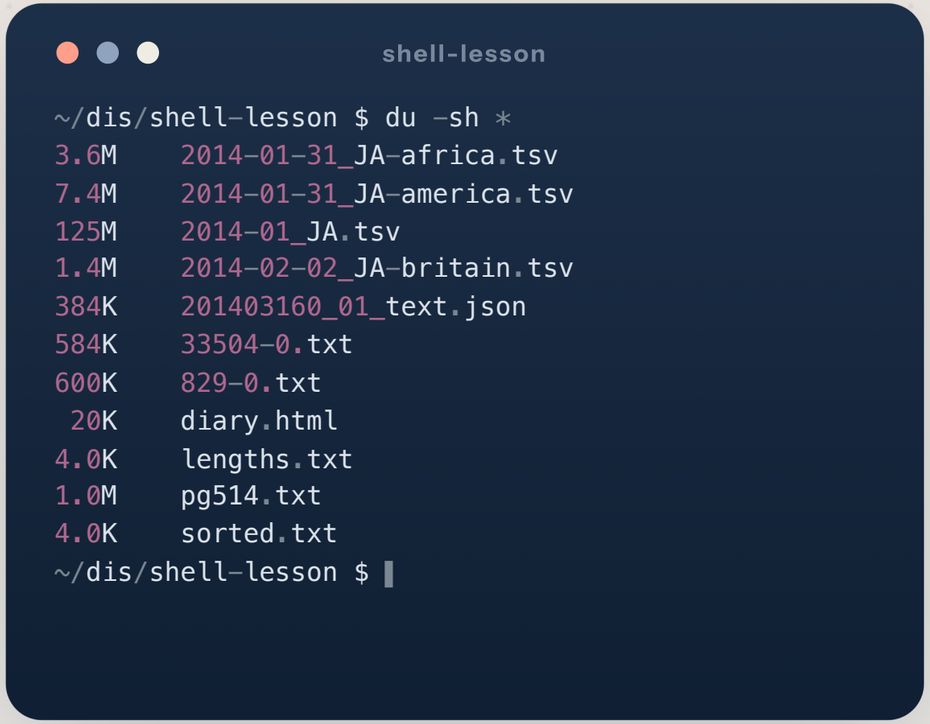
Checking system processes



- `htop`: a real-time, interactive view of processes, CPU and memory usage.
- Sort, filter (F4) and kill processes (F9) right in the interface; quit with `q`.
- Not preinstalled — macOS: `brew install htop`, Ubuntu: `sudo apt install htop`. `top` works everywhere.
- `ps` shows a snapshot: process ID (PID), user, CPU and memory usage, command.
- `ps aux | grep python` finds your Python processes; `kill <PID>` stops one.

More system information

Command	Tells you
`whoami`	your user name
`hostname`	the name of the computer
`uname -a`	operating system and kernel version
`date`	current date and time
`uptime`	how long the system has been running, and its load
`df -h`	free disk space, human-readable
`du -sh *`	size of each file and folder here
`top`, `htop`	running processes, live
`ps aux`	all running processes, as a snapshot



Before you open a dataset in Python, check how big it is.

Basic networking commands

`ping` Test connectivity

```
ping -c 4 google.com
```

Is a host reachable, and how fast does it answer? Without `-c` it runs until Ctrl + C.

`curl` Transfer data from or to a server

```
curl https://example.com
```

Prints the response; `-O` saves it as a file, `-L` follows redirects. Great for testing web APIs.

`wget` Download files

```
wget https://example.com/file.zip
```

Batch downloads and mirroring websites. Not always preinstalled (macOS: `brew install wget`).

`ssh` Secure remote access

```
ssh username@hostname
```

Run commands on a remote server as if you were sitting in front of it. `exit` leaves.

The CLI in the rest of this course

06.11.26

Version Control Software

```
git init
```

Git is a command line tool first. Every Git command runs in the terminal.

13.11.26

Agentic Coding

```
ls, grep, python …
```

Coding agents run shell commands for you — you must read them before you approve them.

27.11.26

Web Scraping

```
curl -L https://…
```

Look at the raw HTML and API responses before you write a scraper.

11.12.26

Data Science Packages

```
pip install pandas
```

Install and run Python packages from the terminal.

Install Git before next week — on Windows it brings Git Bash along: **gitforwindows.org**.

Today’s learning outcomes

✓ Understand the role of the CLI

Explain what it is, its advantages, and when to use it instead of a GUI.

✓ Navigate the file system

Move around with `pwd`, `ls`, `cd`; use absolute and relative paths.

✓ Manipulate files and directories

Create, copy, move and delete them — carefully.

✓ Execute and chain commands

Use options and arguments, wildcards, redirects and pipes.

✓ Write and run basic scripts

Create, execute and debug simple shell scripts that automate tasks.

✓ Access system information

Monitor processes and resources; use basic network tools.

Best practices: history, Tab completion, man pages — and `pwd` before `rm`.

Check your understanding

Show answers →

01

You are in `/alice/code/bin`. What is the relative path to `config.toml`?

02

What is the difference between `>` and `>>`?

03

What does this pipeline print?

```
wc -l *.tsv | sort -n | head -n 1
```

04

`./greet.sh` fails with “Permission denied”. What now?

Questions?

Next lecture: Version Control Software on 06.11.26.
Please install Git before then.

RESOURCES FROM TODAY

- librarycarpentry.org/lc-shell — the shell lesson with today's data
- swcarpentry.github.io/shell-novice — Software Carpentry: The Unix Shell
- missing.csail.mit.edu — The Missing Semester of Your CS Education
- explainshell.com — paste a command, get it explained
- tldr.sh — short, example-based man pages
- gitforwindows.org — Git Bash for Windows