Lecture 06-2: Command Line Skills for Data Detectives

DATA 503: Fundamentals of Data Engineering

Lucas P. Cordova, Ph.D.

2026-02-25

This supplemental lecture reviews core command-line navigation (cd, ls, paths) with visual diagrams, then builds hands-on proficiency with the text-processing commands needed for the Command Line Mystery homework: grep, sed, awk, head, tail, cat, wc, sort, uniq, cut, pipes, and redirection. Students follow along inside the lucascordova/dataeng Docker container using a practice dataset downloaded with curl.

Table of contents

1 Quick Review: Navigation Essentials **2**

2 The Filesystem: Your Map **3**

3 Hands-On Setup: Practice Dataset **8**

4 Core Commands: Reading Files **10**

5 Core Commands: Searching with `grep` **12**

6 Core Commands: Extracting with `sed` **14**

7 Core Commands: `cut`, `sort`, `uniq` **15**

8 Core Commands: `awk` **17**

9 Pipes and Redirection Revisited **18**

10 Working with Archives **19**

11 Putting It All Together: Mystery Prep **20**

12 Advanced Patterns **24**

13 Command Cheat Sheet **25**

14 What Is Next **26**

15 References **27**

1 Quick Review: Navigation Essentials

1.1 Last Time: The Big Ideas

Key takeaways from Lecture 06-1:

Concept	What It Means
Shell	The program interpreting your commands (bash, zsh)
Terminal	The window you type in
Docker	Portable, consistent environment for everyone
Pipes (|)	Connect output of one command to input of another
Redirection (>, >>)	Send output to a file instead of the screen
OSEMN	Obtain, Scrub, Explore, Model, iNterpret

Today we go deeper on **navigation** and learn the **text-processing commands** you need for the Command Line Mystery homework.

1.2 Fire Up Your Docker Container

Open your terminal and start the course container:

Mac/Linux:

```
docker run --rm -it -v ~/Downloads/data:/data lucascordova/dataeng
```

Windows (PowerShell):

```
docker run --rm -it -v ${HOME}\Downloads\data:/data lucascordova/dataeng
```

Verify you are in:

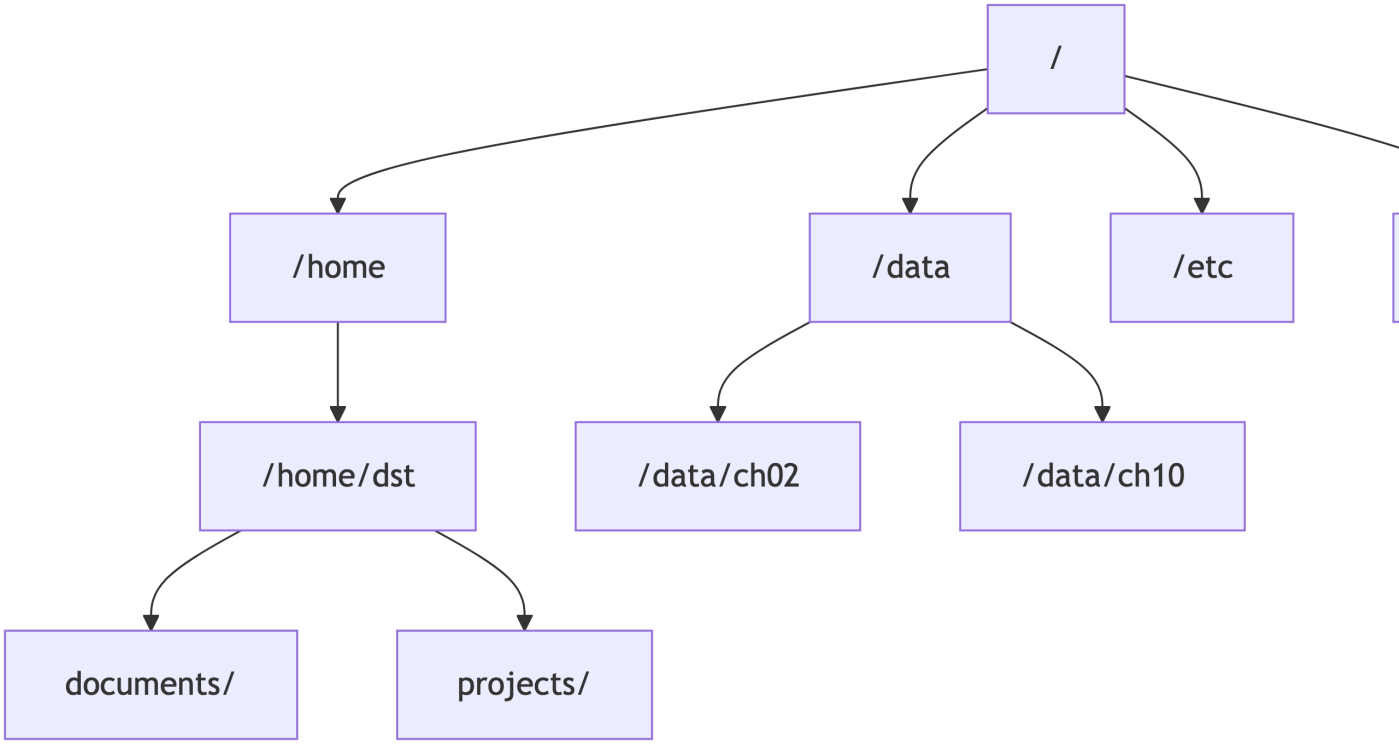
```
whoami
pwd
```

You should see you are the `dst` user at `/home/dst`.

2 The Filesystem: Your Map

2.1 The Directory Tree

Every Unix system is a tree of directories (folders) starting from the **root** `/`:



- `/` is the **root** – the top of everything
- `/home/dst` is your **home directory** (also called `~`)
- `/data` is where we mounted our local files

2.2 Where Am I? `pwd`

`pwd` = Print Working Directory. It tells you your current location as an **absolute path**:

```
$ pwd
/home/dst
```

Think of it like GPS for your terminal. When you are lost, `pwd` is your friend.

2.3 What Is Here? `ls`

`ls` lists what is in the current directory:

```
$ ls
$ ls -l          # long format (permissions, size, date)
$ ls -la         # include hidden files (starting with .)
$ ls -lh         # human-readable file sizes
$ ls /data       # list a specific directory
```

Common options:

Option	What It Shows
`-l`	Long format (one file per line with details)
`-a`	All files including hidden (dotfiles)
`-h`	Human-readable sizes (KB, MB, GB)
`-F`	Append `/` to directories, `*` to executables

2.4 Absolute vs Relative Paths

Absolute Path – starts from root `/`

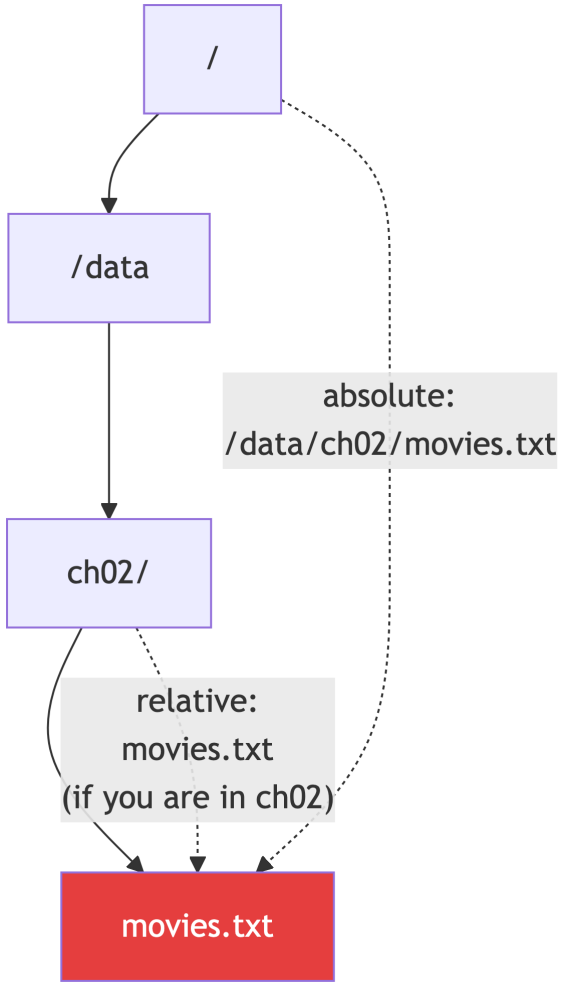
```
$ cat /data/ch02/movies.txt
```

Works from **anywhere**. Like a full street address.

Relative Path – starts from where you are

```
$ cd /data
$ cat ch02/movies.txt
```

Depends on your current directory. Like saying "two blocks north."



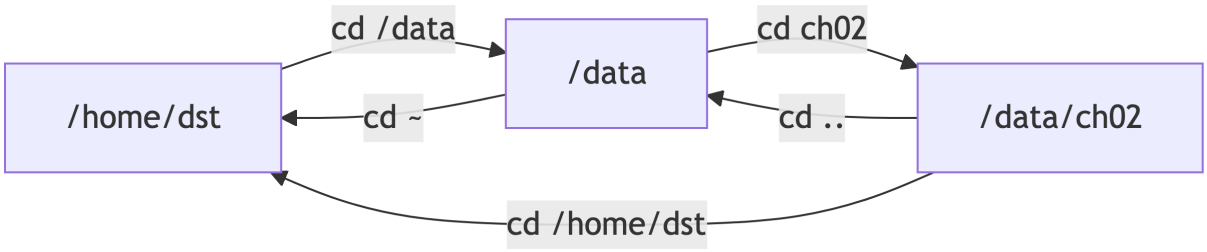
2.5 Navigating with `cd`

`cd` = Change Directory. The most-used command in your terminal life:

```
$ cd /data          # go to /data (absolute)
$ cd ch02           # go into ch02 subdirectory (relative)
$ cd ..             # go UP one level (parent directory)
$ cd ../..          # go UP two levels
$ cd ~              # go to home directory (/home/dst)
$ cd -              # go back to the PREVIOUS directory
$ cd                # same as cd ~ (go home)
```

2.6 Visual: Moving Around

Starting at `/home/dst`:



The `..` always means "one level up." You can chain them: `../../` means "two levels up."

2.7 Special Directory Symbols

Symbol	Meaning	Example
`/`	Root directory	`cd /`
`~`	Home directory	`cd ~` or just `cd`
`.`	Current directory	`ls .` (same as `ls`)
`..`	Parent directory	`cd ..`
`-`	Previous directory	`cd -` (toggle back)

These work **everywhere** – in `cd`, `ls`, `cat`, `cp`, `mv`, any command that takes a path.

2.8 Knowledge Check 1

You are currently in `/data/ch02`. What is the **absolute path** you end up at after running these commands?

```
cd ..
cd ch10
cd ../ch02
```

Answer: `/data/ch02`

Step by step:

1. `cd ..` – moves up to `/data`
2. `cd ch10` – moves into `/data/ch10`
3. `cd ../ch02` – moves up to `/data`, then into `ch02` = `/data/ch02`

You ended up right where you started!

2.9 Knowledge Check 2

You are in `/home/dst/projects`. Which of these commands will list the files in `/data`?

A) `ls data`

B) `ls /data`

C) `ls ../../data`

D) Both B and C

Answer: D) Both B and C

B is an absolute path – works from anywhere.

C is a relative path: from `/home/dst/projects`, `..` goes to `/home/dst`, `../..` goes to `/home`, and wait – that gives `/home/data` which does not exist. Actually let us re-trace: `/home/dst/projects` -> `..` = `/home/dst` -> `../..` = `/`. So `../../data` = `/data`. Hmm, actually: from `/home/dst/projects`, `..` = `/home/dst`, `../..` = `/home`, `../../data` = `/home/data`. So C is WRONG.

Corrected Answer: B only.

`ls /data` is an absolute path. `ls ../../data` from `/home/dst/projects` resolves to `/home/data`, which does not exist. `ls ../../../data` would be needed to reach `/data` (up 3 levels to `/`, then into `data`).

2.10 Creating and Removing Directories

```
$ mkdir practice          # create a directory
$ mkdir -p a/b/c          # create nested directories
$ rmdir practice          # remove empty directory
$ rm -r a                 # remove directory and everything in it
```

The `-p` flag on `mkdir` creates all parent directories as needed. Without it, `mkdir a/b/c` fails if `a/b` does not exist.

3 Hands-On Setup: Practice Dataset

3.1 Find Your Book Data

Last week you downloaded the data from the *Data Science at the Command Line* book. It should already be inside your `/data` directory. Let us find it:

```
$ ls /data
```

Depending on how you unzipped last week, your chapter directories might be in different places:

```
# If you unzipped directly into /data:
$ ls /data/ch02

# If you kept the outer folder:
$ ls /data/book_data/ch02

# If you used a different name:
$ ls /data/data-science-at-the-command-line-master/ch02
```

Use `ls` to poke around and find where `ch02` lives. Once you find it, `cd` into it:

```
$ cd /data/ch02       # adjust this path to match YOUR layout
$ ls
```

You should see files like `movies.txt`, `dates.txt`, and other text files.

3.2 Verify Your Setup

```
$ cat movies.txt
```

If you see a list of movies, you are good. If you get "No such file or directory," you are in the wrong place. Use `pwd` to check where you are, then `ls` to look around.

This is your first real navigation exercise. Finding files by exploring directory structures is exactly what you will do in the Command Line Mystery homework.

3.3 Grab a Larger Dataset

The book data is great for small examples, but we also want a real-world dataset. Let us download an Apache web server access log using `wget`:

```
cd /tmp
wget https://raw.githubusercontent.com/elastic/examples/master/Common%20Data%20Formats/apache
ls -lh apache_logs
```

That is it. One command, one file. `wget` saves the file using the name from the URL (`apache_logs`) by default.

3.4 `wget` – Download Files from the Web

`wget` is a command-line tool for downloading files. It is simpler than `curl` for straightforward downloads:

```
$ wget https://example.com/data.csv
```

Common options:

Flag	Meaning
`-O filename`	Save as a specific filename (capital O)
`-q`	Quiet mode (no progress output)
`-P dir/`	Save to a specific directory
`--no-check-certificate`	Skip SSL verification (use with caution)

3.5 `wget` vs `curl`

Both download files, but they have different strengths:

Feature	`wget`	`curl`
Default behavior	Saves to file	Prints to stdout
Recursive downloads	Yes (`-r`)	No
Resume interrupted downloads	Yes (`-c`)	Yes (`-C -`)
API requests (POST, headers)	Limited	Full support
Availability	Linux/Docker (common)	macOS/Linux (universal)

Rule of thumb: Use `wget` when you just need to download a file. Use `curl` when you need to interact with an API or need fine-grained control over the request.

3.6 Rename for Convenience

The downloaded file is called `apache_logs` (no extension). Let us rename it so it is clear what it is:

```
mv apache_logs access.log
wc -l access.log
```

Now you have a real Apache web server log file to explore. This is the kind of data you will encounter as data engineers.

3.7 Knowledge Check 3

What is the difference between these two commands?

```
wget https://example.com/data.csv
curl -O https://example.com/data.csv
```

Answer: They both download the file and save it as `data.csv`. `wget` saves to a file by default using the filename from the URL. `curl -O` (capital O) does the same thing – saves using the remote filename. Without `-O`, curl prints to stdout instead. For simple file downloads, `wget` is slightly more convenient. For API work, `curl` is the standard.

4 Core Commands: Reading Files

4.1 `cat` – Print Entire File

```
$ cat movies.txt
```

`cat` dumps the whole file to your terminal. Fine for small files. For large files, it will flood your screen.

Use it to combine files too:

```
$ cat file1.txt file2.txt > combined.txt
```

4.2 `head` and `tail` – Peek at Files

```
$ head movies.txt          # first 10 lines
$ head -n 3 movies.txt     # first 3 lines
$ tail movies.txt          # last 10 lines
$ tail -n 5 movies.txt     # last 5 lines
$ tail -n +2 data.csv      # everything EXCEPT the first line
```

`tail -n +2` is the classic trick to skip a header row in a CSV.

4.3 `wc` – Count Things

```
$ wc movies.txt            # lines, words, characters
$ wc -l movies.txt         # just line count
$ wc -w movies.txt         # just word count
$ wc -c movies.txt         # just byte count
```

Quick sanity check: "How big is this file?"

```
$ wc -l access.log
```

The access.log should have around 10,000 lines – a realistic web server log file.

4.4 `less` – Page Through Large Files

```
$ less access.log
```

Key	Action
Space / `f`	Next page
`b`	Previous page
`/pattern`	Search forward
`n`	Next search match
`q`	Quit

Unlike `cat`, `less` does not load the entire file into memory. Use it for big files.

4.5 Knowledge Check 4

How would you print lines 20 through 25 of a file called `data.txt`?

Answer: Several ways:

```
head -n 25 data.txt | tail -n 6
```

Or:

```
sed -n '20,25p' data.txt
```

The first approach: `head -n 25` gets lines 1-25, then `tail -n 6` gets the last 6 of those (lines 20-25).

5 Core Commands: Searching with `grep`

5.1 `grep` – Find Lines Matching a Pattern

`grep` is your search engine for files. It prints every line that matches your pattern:

```
$ grep "Star Wars" movies.txt
```

Essential flags:

Flag	Meaning
`-i`	Case-insensitive search
`-c`	Count matches (don't print them)
`-n`	Show line numbers
`-v`	Invert: show lines that do NOT match
`-l`	Show only filenames that contain a match
`-r`	Search recursively through directories
`-w`	Match whole words only

5.2 `grep` Context Flags

Sometimes you need to see what is around a match:

```
$ grep -A 3 "error" access.log   # 3 lines AFTER match
$ grep -B 2 "error" access.log   # 2 lines BEFORE match
$ grep -C 2 "error" access.log   # 2 lines BEFORE and AFTER
```

Think of it as: **A**fter, **B**efore, **C**ontext.

5.3 `grep` with Pipes

The real power comes from chaining grep with other commands:

```
$ cat access.log | grep "404" | wc -l
```

This counts how many 404 (Not Found) errors are in the log.

```
$ grep "404" access.log | head -n 5
```

Show the first 5 lines that contain "404".

5.4 Knowledge Check 5

Using the `access.log` file, write a command that counts how many requests came from the IP address `83.149.9.216`.

Answer:

```
grep -c "83.149.9.216" access.log
```

Or equivalently:

```
grep "83.149.9.216" access.log | wc -l
```

5.5 Knowledge Check 6

How would you search for the word "CLUE" in a file called `crimescene`, but only show the matching lines?

Answer:

```
grep "CLUE" crimescene
```

Plain grep prints matching lines by default. This is exactly the first step in the Command Line Mystery homework.

6 Core Commands: Extracting with `sed`

6.1 `sed` – Stream Editor

`sed` processes text line by line. Most common use: **print specific lines** and **find-and-replace**.

Print a specific line:

```
$ sed -n '5p' movies.txt          # print only line 5
$ sed -n '10,20p' movies.txt      # print lines 10 through 20
```

Find and replace:

```
$ sed 's/old/new/' file.txt       # replace first occurrence per line
$ sed 's/old/new/g' file.txt      # replace ALL occurrences per line
```

The `s/pattern/replacement/` syntax is the substitution command.

6.2 `sed` in Practice

Remove blank lines from a file:

```
$ sed '/^$/d' file.txt
```

Delete lines containing a pattern:

```
$ sed '/pattern/d' file.txt
```

Print only lines matching a pattern (like grep):

```
$ sed -n '/error/p' access.log
```

6.3 Why `sed -n 'Np'` Matters for the Mystery

In the Command Line Mystery, people have addresses like:

```
Annabel Church     F   38   Buckingham Place, line 179
```

To look up that address, you need line 179 of the street file:

```
$ sed -n '179p' streets/Buckingham_Place
```

This is faster than opening the whole file. You jump straight to the line you need.

6.4 Knowledge Check 7

A file called `people` has this entry:

```
Jeremy Bowers      M   34   Dunstable Road, line 284
```

Write the command to see what is on line 284 of `streets/Dunstable_Road`.

Answer:

```
sed -n '284p' streets/Dunstable_Road
```

7 Core Commands: `cut`, `sort`, `uniq`

7.1 `cut` – Extract Columns

`cut` pulls out specific columns from structured text:

```
$ cut -d',' -f1 data.csv          # first field, comma-delimited
$ cut -d',' -f1,3 data.csv        # fields 1 and 3
$ cut -d' ' -f1 access.log        # first field, space-delimited
$ cut -c1-10 file.txt             # characters 1 through 10
```

Flag	Meaning
`-d`	Delimiter (comma, tab, space, etc.)
`-f`	Field number(s) to extract
`-c`	Character positions to extract

7.2 `sort` – Sort Lines

```
$ sort names.txt               # alphabetical sort
$ sort -n numbers.txt          # numeric sort
$ sort -r names.txt            # reverse sort
$ sort -t',' -k2 data.csv     # sort by 2nd field, comma-delimited
$ sort -u names.txt            # sort and remove duplicates
```

7.3 `uniq` – Remove Adjacent Duplicates

`uniq` only removes **consecutive** duplicates, so you almost always use it with `sort`:

```
$ sort names.txt | uniq            # unique values
$ sort names.txt | uniq -c         # count occurrences
$ sort names.txt | uniq -d         # show only duplicates
```

7.4 The Classic Pipeline: `cut | sort | uniq -c | sort -rn`

This is the "top N" pattern. Find the most frequent values:

```
$ cut -d' ' -f1 access.log | sort | uniq -c | sort -rn | head -10
```

This answers: "What are the top 10 IP addresses in the access log?"

The pipeline:

1. `cut` extracts the IP address (field 1)
2. `sort` groups identical IPs together
3. `uniq -c` counts consecutive duplicates
4. `sort -rn` sorts by count, highest first
5. `head -10` shows the top 10

7.5 Knowledge Check 8

Using the `access.log`, write a pipeline that finds the **top 5 most requested URLs** (URLs are typically the 7th field in a space-delimited Apache log).

Answer:

```
cut -d' ' -f7 access.log | sort | uniq -c | sort -rn | head -5
```

Or using `awk`:

```
awk '{print $7}' access.log | sort | uniq -c | sort -rn | head -5
```

8 Core Commands: `awk`

8.1 `awk` – The Swiss Army Knife

`awk` is a mini programming language for text processing. Basic use: print specific fields.

```
$ awk '{print $1}' access.log          # print first field
$ awk '{print $1, $7}' access.log      # print fields 1 and 7
$ awk -F',' '{print $2}' data.csv      # comma-delimited, field 2
```

`awk` splits each line on whitespace by default. Use `-F` to change the delimiter.

8.2 `awk` with Conditions

Print only lines where a condition is true:

```
$ awk '$9 == 404 {print $7}' access.log
```

This prints the URL (field 7) for every line where the HTTP status code (field 9) is 404.

```
$ awk '$9 == 200 {count++} END {print count}' access.log
```

Count the number of successful (200) requests.

8.3 `awk` vs `cut`

Feature	`cut`	`awk`
Speed	Faster for simple extraction	Slightly slower
Delimiter	Single character only	Any pattern, regex
Conditions	None	Full programming logic
Multiple delimiters	No	Handles whitespace naturally
Math	No	Yes

Rule of thumb: use `cut` for simple extraction, `awk` for anything more complex.

8.4 Knowledge Check 9

Using `awk`, print the IP address and status code (fields 1 and 9) from `access.log` for all lines where the status code is `500`.

Answer:

```
awk '$9 == 500 {print $1, $9}' access.log
```

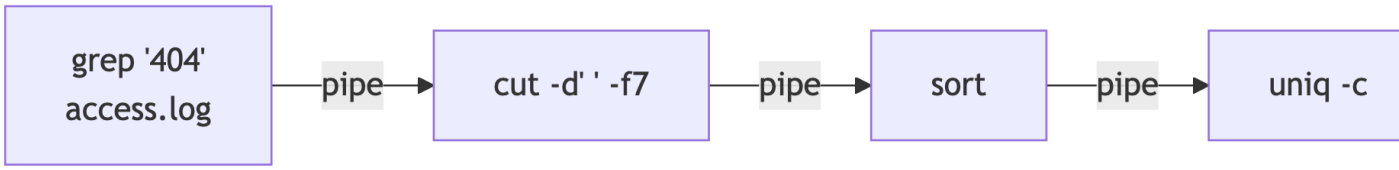
Or to also get a count:

```
awk '$9 == 500 {print $1}' access.log | sort | uniq -c | sort -rn
```

9 Pipes and Redirection Revisited

9.1 The Pipe | – Connecting Commands

The pipe takes stdout of one command and feeds it as stdin to the next:



Each command does one small job. Together they answer: "What are the top 5 URLs returning 404?"

9.2 Output Redirection

Overwrite (`>`):

```
$ grep "404" access.log > errors.txt
```

Creates or **overwrites** the file.

Append (`>>`):

```
$ echo "new line" >> errors.txt
```

Adds to the end of the file.

Suppress errors (`2>/dev/null`):

```
$ grep "x" missing.txt 2>/dev/null
```

Throws away error messages.

Redirect both (`&>`):

```
$ command &> output.txt
```

Captures both stdout and stderr.

9.3 Knowledge Check 10

Write a single pipeline that:

1. Finds all lines containing "GET" in `access.log`
2. Extracts just the URL (field 7)
3. Counts how many unique URLs there are

Answer:

```
grep "GET" access.log | awk '{print $7}' | sort -u | wc -l
```

Or:

```
grep "GET" access.log | cut -d' ' -f7 | sort -u | wc -l
```

10 Working with Archives

10.1 `tar` and `zip` – Packing and Unpacking

You will encounter compressed archives constantly as a data engineer:

zip/unzip:

```
$ unzip archive.zip              # extract to current directory
$ unzip archive.zip -d folder/   # extract to specific folder
$ zip -r backup.zip folder/      # create a zip from a folder
```

tar (tape archive):

```
$ tar xzf archive.tar.gz         # extract gzipped tar
$ tar xjf archive.tar.bz2        # extract bzip2 tar
$ tar czf backup.tar.gz folder/  # create gzipped tar
$ tar tf archive.tar.gz          # list contents without extracting
```

Think of `tar` flags as: e**x**tract, **c**reate, **z** for gzip, **f** for filename.

10.2 `gunzip` – Decompress Single Files

```
$ gunzip file.gz              # decompress (replaces .gz file)
$ gunzip -k file.gz           # decompress and keep original
$ gunzip -c file.gz > out.txt # decompress to stdout
```

10.3 Knowledge Check 11

You downloaded a file called `dataset.tar.gz`. Write the command to extract it into a directory called `mydata/`.

Answer:

```
mkdir mydata
tar xzf dataset.tar.gz -C mydata/
```

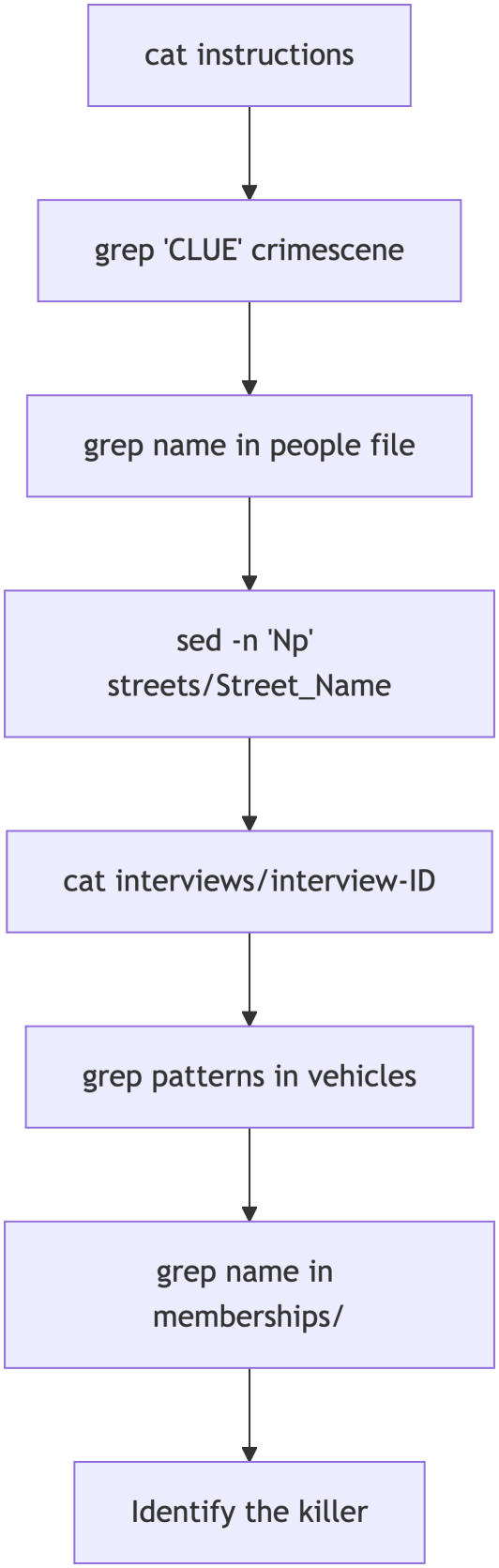
Or:

```
mkdir mydata && cd mydata && tar xzf ../dataset.tar.gz
```

11 Putting It All Together: Mystery Prep

11.1 The Command Line Mystery

Your homework involves solving a murder mystery using only command-line tools. Here is the workflow you will follow:


cat instructions
grep 'CLUE' crimescene
grep name in people file
sed -n 'Np' streets/Street_Name
cat interviews/interview-ID
grep patterns in vehicles
grep name in memberships/
Identify the killer

11.2 Commands You Will Need

Command	Mystery Use
`cat`	Read instructions, interviews
`grep`	Search crimescene for CLUE, search people, vehicles, memberships
`grep -A`	Show context lines after a vehicle match
`sed -n 'Np'`	Look up a specific line in a street file
`wc -l`	Count matches
Pipes (|)	Chain commands together
`cd`	Navigate into mystery/ and its subdirectories
`ls`	See what files and directories are available

11.3 Practice: Mini Mystery

Let us simulate the mystery workflow with our book data. Navigate back to ch02:

```
cd /data/ch02        # adjust to match your layout
ls
```

Try these patterns:

```
# Search for something in a file
grep "Matrix" movies.txt

# Count occurrences
grep -c "the" movies.txt

# Show line numbers
grep -n "Star" movies.txt
```

11.4 Knowledge Check 12

You are in the `mystery/` directory. The `people` file shows:

```
Joe Germuska       M   36   Plainfield Street, line 275
```

Write the commands to:

1. Look up line 275 of `streets/Plainfield_Street`

2. Read the interview file that is referenced there

Answer:

```
sed -n '275p' streets/Plainfield_Street
```

This outputs something like `SEE INTERVIEW #29741223`. Then:

```
cat interviews/interview-29741223
```

11.5 Knowledge Check 13

A clue says the suspect drives a blue Honda with a plate starting with "L337". The `vehicles` file has multi-line records. Write a command to find all matching vehicles and show 5 lines of context after each match.

Answer:

```
grep -A 5 "L337" vehicles | grep -B 2 -A 3 "Honda"
```

Or step by step:

```
grep -A 5 "L337" vehicles
```

Then narrow down:

```
grep -A 5 "L337" vehicles | grep "Blue" -B 3 -A 2
```

11.6 Knowledge Check 14

You narrowed suspects to two people: Joe Germuska and Jeremy Bowers. You need to check if each person is a member of the "AAA" club. The membership file is at `memberships/AAA`. Write the commands.

Answer:

```
grep "Joe Germuska" memberships/AAA
grep "Jeremy Bowers" memberships/AAA
```

If a name appears, they are a member. No output means they are not.

12 Advanced Patterns

12.1 Wildcards in File Paths

The `*` matches any characters in filenames:

```
$ ls *.txt                # all .txt files
$ grep "Bowers" memberships/*    # search ALL membership files
$ cat streets/B*          # all streets starting with B
```

This is called **globbing** and it is handled by the shell, not by the command.

12.2 `find` – Locate Files

```
$ find . -name "*.txt"              # find all .txt files
$ find . -name "*.log" -size +1M    # .log files over 1MB
$ find . -type d                    # find all directories
```

12.3 `xargs` – Build Commands from Input

When you need to run a command on each result from another command:

```
$ find . -name "*.txt" | xargs wc -l
```

This counts lines in every `.txt` file found by `find`.

12.4 Knowledge Check 15

Write a single command that searches ALL files in the `memberships/` directory for "Jeremy Bowers" and shows which files contain a match.

Answer:

```
1 grep -l "Jeremy Bowers" memberships/*
```

The `-l` flag prints only filenames, not the matching lines. This tells you which clubs Jeremy belongs to.

13 Command Cheat Sheet

13.1 Navigation

Command	What It Does
`pwd`	Print current directory
`cd dir`	Change to directory
`cd ..`	Go up one level
`cd ~` or `cd`	Go to home directory
`cd -`	Go to previous directory
`ls`	List directory contents
`ls -la`	List all files with details
`mkdir dir`	Create directory
`mkdir -p a/b/c`	Create nested directories

13.2 File Reading

Command	What It Does
`cat file`	Print entire file
`head -n N file`	First N lines
`tail -n N file`	Last N lines
`tail -n +N file`	Everything from line N onward
`less file`	Page through file
`wc -l file`	Count lines

13.3 Searching and Extracting

Command	What It Does
`grep "pattern" file`	Find matching lines
`grep -i`	Case-insensitive
`grep -c`	Count matches
`grep -n`	Show line numbers
`grep -v`	Invert (non-matching lines)
`grep -A N`	N lines after match
`grep -B N`	N lines before match
`grep -l`	Show filenames only
`grep -r`	Recursive search

Command	What It Does
`sed -n 'Np'`	Print line N
`sed 's/old/new/g'`	Find and replace

13.4 Transforming

Command	What It Does
`cut -d',' -f1`	Extract field 1 (comma-delimited)
`sort`	Sort lines alphabetically
`sort -n`	Sort numerically
`sort -rn`	Sort numerically, descending
`sort -u`	Sort and deduplicate
`uniq`	Remove adjacent duplicates
`uniq -c`	Count duplicates
`awk '{print $1}'`	Print first field
`awk -F',' '{print $2}'`	Print field 2 (comma-delimited)

13.5 Plumbing

Syntax	What It Does
`cmd1 | cmd2`	Pipe output to next command
`cmd > file`	Write output to file (overwrite)
`cmd >> file`	Append output to file
`cmd 2>/dev/null`	Suppress error messages
`curl -L -o file URL`	Download a file
`unzip file.zip`	Extract zip archive
`tar xzf file.tar.gz`	Extract gzipped tar

14 What Is Next

14.1 Your Mission

You now have all the commands you need to solve the Command Line Mystery.

Homework 6: Navigate to the mystery directory and follow the instructions. Use `grep`, `sed`, `cat`, pipes, and your navigation skills to find the killer.

```
cd /data/command-line-mystery/mystery
cat ../instructions
```

The hints are there if you get stuck (`cat ../hint1`, `cat ../hint2`, etc.), but try to solve it on your own first. You have the skills.

Good luck, detective.

15 References

15.1 References

1. Janssens, J. (2021). *Data Science at the Command Line* (2nd ed.). O'Reilly Media. https://datascienceatthecommandline.com
2. Veltman, N. *The Command Line Murders*. https://github.com/veltman/clmystery
3. GNU Coreutils Manual. https://www.gnu.org/software/coreutils/manual
4. Robbins, A. (2002). *sed & awk* (2nd ed.). O'Reilly Media.
5. Shotts, W. (2019). *The Linux Command Line* (2nd ed.). No Starch Press.