# The bash shell and UNIX commands

Chris Paciorek

2025-09-01

## Table of contents



## Overview

Reference:

- Newham and Rosenblatt, Learning the bash Shell, 2nd ed.
- Damien Irving, Kate Hertweck, Luke Johnston, Joel Ostblom, Charlotte Wickham, and Greg Wilson. Research Software Engineering with Python
- SCF Bash tutorial

# 1. Shell basics

The shell is the interface between you and the (UNIX-style) operating system.

I'll use 'UNIX' to refer to the family of operating systems that descend from the path-breaking UNIX operating system developed at AT&T's Bell Labs in the 1970s. These include MacOS and various flavors of Linux (e.g., Ubuntu, Debian, CentOS, Fedora).

When you are working in a terminal window (i.e., a window providing the command line interface), you're interacting with a shell. From the shell you can run UNIX commands such as `cp`, `ls`, `grep`, etc. (as well as start various applications).

Here's a graphical representation of how the shell relates to various programs, commands, and the operating system.

There are multiple shells (`sh`, `bash`, `zsh`, `csh`, `tcsh`, `ksh`). We'll assume usage of `bash`, as this is a very commonly-used shell in Linux, plus was the default for Mac OS X until Catalina (`zsh`, very similar to bash, is now the default), the SCF machines, and the UC Berkeley campus cluster (Savio). All of the various shells allow you to run UNIX commands.

For your work on this unit, either bash on a Linux machine, the older version of bash on MacOS, or zsh on MacOS (or Linux) are fine. I'll probably demo everything using bash on a Linux machine.

Note that the UNIX utilities on Linux have some annoying, but usually minor, differences from those on MacOS that may be occasionally confusing (in particular the options to various commands can differ on MacOS).

For example, you can use `ls --all` in Linux but not MacOS.

The Windows PowerShell and old cmd.exe/DOS command interpreter both provide a command-line interface on Windows, but not a UNIX-based one. We won't consider them here.

UNIX shell commands are designed to each do a specific task really well and really fast. They are modular and composable, so you can build up complicated operations by combining the commands. These tools were designed decades ago, so using the shell might seem old-fashioned, but the shell still lies at the heart of modern scientific computing. By using the shell you can automate your work and make it reproducible. And once you know how to use it, you'll find that enter commands quickly and without a lot of typing.

# 2. Using the bash shell

For this Unit, we'll rely on the bash shell tutorial for the details of how to use the shell. We won't cover the page on Managing Processes. For the moment, we won't cover the page on Regular Expressions, but when we talk about string processing and regular expressions in Unit 5, we'll come back to that material.

# 3. bash shell examples

Here we'll work through a few examples to start to give you a feel for using the bash shell to manage your workflows and process data.

First let's get the files from GitHub to have a set of files we can do interesting things with.

```
git clone https://github.com/berkeley-stat243/fall-2025
```

One important note is that most of the shell commands that work with data inside files (in contrast to commands like `ls` and `cd`) work only with text files and not binary files. Also the commands operate on a line-by-line basis.

**Our first mission** is some basic manipulation of a data file. Suppose we want to get a sense for the number of weather stations in different states using the *coop.txt* file.

```
cd fall-2025/data
gzip -cd coop.txt.gz | less
gunzip coop.txt.gz
cut -b50-70 coop.txt | less
cut -b60-61 coop.txt | uniq
cut -b60-61 coop.txt | sort | uniq
cut -b60-61 coop.txt | sort | uniq -c
## Do it all in one line with no change to the original file:
gzip -cd coop.txt.gz | cut -b60-61 coop.txt | sort | uniq -c
```

I could have done that in R or Python, but it would have required starting the program up and reading all the data into memory.

If you feel that manually figuring out the position of the state field is inconsistent with our emphasis on programmatic workflows, see the fifth challenge below.

**Our second mission**: how can I count the number of fields in a CSV file programmatically?

```
tail -n 1 cpds.csv | grep -o ',' | wc -l
nfields=$(tail -n 1 cpds.csv | grep -o ',' | wc -l)

nfields=$((${nfields}+1))
echo $nfields

## Alternatively, we can use `bc`.
nfields=$(echo "${nfields}+1" | bc)
```

Trouble-shooting: How could the syntax above get the wrong answer?

Extension: We could write a function that can count the number of fields in any file.

Extension: How could I see if all of the lines have the same number of fields?

**We'll work on Challenge #1 here.**

**Our third mission**: was the `requests` package in the five most recently modified Quarto Markdown files in the units directory?

```
cd ../units
grep -l 'import requests' unit4-goodPractices.qmd
ls -tr *.qmd
```

```
## If unit4-goodPractices.qmd is not amongst the 5 most recently used,
## let's artificially change the timestamp so it is recently used.
touch unit4-goodPractices.qmd

ls -tr *.qmd | tail -n 5
ls -tr *.qmd | tail -n 5 | grep requests
ls -tr *.qmd | tail -n 5 | grep "unit4-goodPractices"
ls -tr *.qmd | tail -n 5 | xargs grep 'import requests'
ls -tr *.qmd | tail -n 5 | xargs grep -l 'import requests'
```

Notice that `man tail` indicates it can take input from a FILE or from `stdin`. Here it uses `stdin`, so it is gives the last five lines of the output of `ls`, not the last five lines of the files indicated in that output.

`man grep` also indicates it can take input from a FILE or from `stdin`. However, we want grep to operate on the content of the files indicated in stdin. So we use `xargs` to convert `stdin` to be recognized as arguments, which then are the FILE inputs to `grep`.

An alternative to `xargs` is to embed the `ls` invocation, using `$()`, to explicitly pass the file names as the argument to `grep`:

```
grep -l 'import requests' $(ls -tr *.R | tail -n 5)
```

Here are some of the ways we can pass information from a command to somewhere else:

- Piping allows us to pass information from one command to another command via stdout to stdin.
- `$()` allows us to store the result of a command in a variable.
  - also used to create a temporary variable to pass the output from one command as an option or argument (e.g., the FILE argument) of another command
- File redirection operators such as `>` and `>>` allow us to pass output from a command into a file.

**We'll work on Challenge #2 here.**

**Our fourth mission**: write a function that will move the most recent $n$ files in your Downloads directory to another directory.

In general, we want to start with a specific case, and then generalize to create the function.

```
ls -rt ~/Downloads | tail -n 5
# Create dummy test files without any spaces.
touch ~/Downloads/test{1..4}
ls -rt ~/Downloads | tail -n 5

## Sometimes the ~ behaves weirdly in scripting, so let's use full path.
mv "/accounts/vis/paciorek/Downloads/$(ls -rt \
   /accounts/vis/paciorek/Downloads | tail -n 1)" ~/Desktop

function mvlast() {
    mv "/accounts/vis/paciorek/Downloads/$(ls -rt \
       /accounts/vis/paciorek/Downloads | tail -n 1)" $1
}
```

Note that the quotes deal with cases where a file has a space in its name.

If we wanted to handle multiple files, we could do it with a loop:

```
function mvlast() {
    for ((i=1; i<=${1}; i++)); do
        mv "/accounts/vis/paciorek/Downloads/$(ls -rt \
           /accounts/vis/paciorek/Downloads | tail -n 1)" ${2}
    done
}
```

Side note: if we were just moving files from the current working directory and with files without spaces in their names, it should be possible to use `tail -n ${1}` without the loop.

**We'll work on Challenge #3 here.**

**We'll work on Challenge #4 here.**

**Our fifth mission**: automate the process of determining what Python packages are used in all of the qmd code chunks here and install those packages on a new machine.

```
grep import *.qmd
grep --no-filename import *.qmd
grep --no-filename "^import" *.qmd
grep --no-filename "^import " *.qmd
grep --no-filename "^import " *.qmd | sort | uniq
grep --no-filename "^import " *.qmd | cut -d'#' -f1
grep --no-filename "^import " *.qmd | cut -d'#' -f1 | sed  "s/as .*//"
grep --no-filename "^import " *.qmd | cut -d'#' -f1 | \
                  sed  "s/as .*//" | sed "s/import //" > tmp.txt
sed "s/,/\n/g" tmp.txt | sed "s/ //g" | sort | uniq | tee requirements.txt

## Note: on a Mac, use 's/,/\\\n/g'
## See https://superuser.com/questions/307165/newlines-in-sed-on-mac-os-x

echo "There are $(wc -l requirements.txt | cut -d' ' -f1) \
unique packages we will install."
## Note: on Linux, wc -l puts the number as the first characters of the output.
## On a Mac, there may be a bunch of spaces preceding the number, so try this:
## echo "There are $(wc -l libs.txt | tr -s ' ' | cut -d' ' -f2) \
## unique packages we will install."

pip install -r requirements.txt
# or use Mamba/Conda
```

That was pretty quick-and-dirty and there are established ways to do that (e.g., `pipreqs` and `modulefinder`) – the main point was to illustrate how one can quickly hack together some code to do fairly complicated tasks.

**Our sixth mission**: suppose I've accidentally started a bunch of jobs (perhaps with a for loop in

bash!) and need to kill them. (This example uses syntax from the Managing Processes page of the bash tutorial, so it goes beyond what you were asked to read for this Unit.)

```
# Use a 'here document':
cat > job.py << EOF
import time
time.sleep(1e5)
EOF

# alternatively:
echo -e "import time\ntime.sleep(1e5)" > job.py

nJobs=30
for (( i=1; i<=${nJobs}; i++ )); do
   python job.py > job-${i}.out &
done

# on Linux:
ps -o pid,pcpu,pmem,user,cmd -C python
ps -o pid,pcpu,pmem,user,cmd,start_time --sort=start_time -C python | tail -n 30
ps -o pid --sort=start_time -C python | tail -n ${nJobs} | xargs kill

# on a Mac:
ps -o pid,pcpu,pmem,user,command | grep python
# not clear how to sort by start time
ps -o pid,command | grep python | cut -d' ' -f1 |  tail -n ${nJobs} | xargs kill
```

Or, if you started the jobs all at once, then the process IDs are likely to be in sequential order, so you could use brace expansion in combination with `kill`:

```
kill {871841..871870}
```

# 4. bash shell challenges

## 4.1 First challenge

Consider the file `cpds.csv`. How would you write a shell command that returns "There are 8 occurrences of the word 'Belgium' in this file.", where '8' should instead be the correct number of times the word occurs.

Extra: make your code into a function that can operate on any file indicated by the user and any word of interest.

## 4.2 Second challenge

Consider the data in the `RTADataSub.csv` file. This is a subset of data giving freeway travel times for segments of a freeway in an Australian city. The data are from a kaggle.com competition. We want to

try to understand the kinds of data in each field of the file. The following would be particularly useful if the data were in many files or the data were many gigabytes in size.

1. First, take the fourth column. Figure out the unique values in that column.
2. Next, automate the process of determining if any of the values are non-numeric so that you don't have to scan through all of the unique values looking for non-numbers. You'll need to look for the following regular expression pattern `[^0-9]`, which is interpreted as NOT any of the numbers 0 through 9.

Extra: do it for all the fields, except the first one. Have your code print out the result in a human-readable way understandable by someone who didn't write the code. For simplicity, you can assume you know the number of fields.

### 4.3 Third challenge

1. For Belgium, determine the minimum unemployment value (field #6) in `cpds.csv` in a programmatic way.
2. Have what is printed out to the screen look like "Belgium 6.2".
3. Now store the unique values of the countries in a variable, first stripping out the quotation marks.
4. Figure out how to automate step 1 to do the calculation for all the countries and print to the screen.
5. How would you instead store the results in a new file?

### 4.4 Fourth challenge

Let's return to the `RTADataSub.csv` file and the issue of missing values.

1. Using the data from `RTADataSub.csv`, create a new file without any rows that have an 'x' (which indicate a missing value).
2. Turn the code into a function that also prints out the number of rows that are being removed and that sends its output to stdout so that it can be used with piping.
3. Now modify your function so that the user could provide the missing value string and the input filename.

### 4.5 Fifth challenge

Consider the `coop.txt` weather station file.

Figure out how to use `grep` to tell you the starting position of the state field. Hints: search for a known state-country combination and figure out what flags you can use with `grep` to print out the "byte offset" for the matched state.

Use that information to automate the first mission where we extracted the state field using `cut`. You'll need to do a bit of arithmetic using shell commands.

# 5. Data storage and file formats on a computer

## Text and binary files

In general, files can be divided into text files and binary files. In both cases, information is stored as a series of bits. Recall that a bit is a single value in base 2 (i.e., a 0 or a 1), while a byte is 8 bits.

A **text file** is one in which the bits in the file encode individual characters. Note that the characters can include the digit characters 0-9, so one can include numbers in a text file by writing down the digits needed for the number of interest. Examples of text file formats include CSV, XML, HTML, and JSON.

Text files may be simple ASCII files (i.e., files encoded using ASCII) or files in other encodings such as UTF-8, both covered in Unit 5.

- ASCII files have 8 bits (1 byte) per character and can represent 128 characters (the 52 lower and upper case letters in English, 10 digits, punctuation and a few other things – basically what you see on a standard US keyboard).
- UTF-8 files have between 1 and 4 bytes per character.

Some text file formats, such as JSON or HTML, are not easily interpretable/manipulable on a line-by-line basis (unlike, e.g., CSV), so they are not as amenable to processing using shell commands.

A **binary file** is one in which the bits in the file encode the information in a custom format and not simply individual characters. Binary formats are not (easily) human readable but can be more space-efficient and faster to work with (because it can allow random access into the data rather than requiring sequential reading). The meaning of the bytes in such files depends on the specific binary format being used and a program that uses the file needs to know how the format represents information. Examples of binary files include netCDF files, Python pickle files, R data (e.g., .Rda) files, , and compiled code files.

Numbers in binary files are usually stored as 8 bytes per number. We'll discuss this much more in 243B.

## Common file types

Here are some of the common file types, some of which are text formats and some of which are binary formats.

1. 'Flat' text files: data are often provided as simple text files. Often one has one record or observation per row and each column or field is a different variable or type of information about the record. Such files can either have a fixed number of characters in each field (*fixed width format*) or a special character (a *delimiter*) that separates the fields in each row. Common delimiters are tabs, commas, one or more spaces, and the pipe (|). Common file extensions are `.txt` and `.csv`. Metadata (information about the data) are often stored in a separate file. CSV files are quite common, but if you have files where the data contain commas, other delimiters might be preferable. Text can be put in quotes in CSV files, and this can allow use of commas within the data. This is difficult to deal with from the command line, but `read_table()` in Pandas handles this situation.

    - One occasionally tricky difficulty is as follows. If you have a text file created in Windows, the line endings are coded differently than in UNIX. Windows uses a newline (the ASCII

character `\n`) and a carriage return (the ASCII character `\r`) whereas UNIX uses only a newline in UNIX). There are UNIX utilities (`fromdos` in Ubuntu, including the SCF Linux machines and `dos2unix` in other Linux distributions) that can do the necessary conversion. If you see `\^M` at the end of the lines in a file, that's the tool you need. Alternatively, if you open a UNIX file in Windows, it may treat all the lines as a single line. You can fix this with `todos` or `unix2dos`.

2. In some contexts, such as textual data and bioinformatics data, the data may be in a text file with one piece of information per row, but without meaningful columns/fields.
3. Data may also be in text files in formats designed for data interchange between various languages, in particular XML or JSON. These formats are "self-describing"; namely the metadata is part of the file. The `lxml` and `json` packages are useful for reading and writing from these formats. More in Section 4.
4. You may be scraping information on the web, so dealing with text files in various formats, including HTML. The `requests` and `BeautifulSoup` packages are useful for reading HTML.
5. In scientific contexts, netCDF (`.nc`) (and the related HDF5) are popular format for gridded data that allows for highly-efficient storage and contains the metadata within the file. The basic structure of a netCDF file is that each variable is an array with multiple dimensions (e.g., latitude, longitude, and time), and one can also extract the values of and metadata about each dimension. The `netCDF4` package in Python nicely handles working with netCDF files.
6. Data may already be in a database or in the data storage format of another statistical package (`Stata`, `SAS`, `SPSS`, etc.). The `Pandas` package in Python has capabilities for importing Stata (`read_stata`), SPSS (`read_spss`), and SAS (`read_sas`) files, among others.
7. For Excel, there are capabilities to read an Excel file in Python (see the `read_excel` function in Pandas), but you can also just go into Excel and export as a CSV file or the like and then read that into Python. In general, it's best not to pass around data files as Excel or other spreadsheet format files because (1) Excel is proprietary, so someone may not have Excel and the format is subject to change, (2) Excel imposes limits on the number of rows, (3) one can easily manipulate text files such as CSV using UNIX tools, but this is not possible with an Excel file, (4) Excel files often have more than one sheet, graphs, macros, etc., so they're not a data storage format per se.
8. Python can easily interact with databases (SQLite, DuckDB, PostgreSQL, MySQL, Oracle, etc.), querying the database using SQL and returning results to Python. More in Unit 6 and in the large datasets tutorial.

## CSV vs. specialized formats such as Parquet

CSV is a common format (particularly in some disciplines/contexts) and has the advantages of being simple to understand, human readable, and readily manipulable by line-based processing tools such as shell commands. However, it has various disadvantages:

- storage is by row, which will often mix values of different types;
- extra space is taken up by explicitly storing commas and newlines; and
- one must search through the document to find a given row or value – e.g., to find the 10th row, we must search for the 9th newline and then read until the 10th newline.

A popular file format that has some advantages over plain text formats such as CSV is Parquet. The storage is by column (actually in chunks of columns). This works well with how datasets are often structured in that a given field/variable will generally have values of all the same type and there may be many repeated values, so there are opportunities for efficient storage including compression. Storage by column also allows retrieval only of the columns that a user needs. As a result data stored in the Parquet format often takes up much less space than stored as CSV and can be queried much faster. Also note that data stored in Parquet will often be stored as multiple files.

Here's a brief exploration using a data file not in the class repository because of its size but available online here.

```
import time, os
import pandas as pd

## Read from CSV.
t0 = time.time()
data_from_csv = pd.read_csv(os.path.join('..', 'data', 'airline.csv'))
print(time.time() - t0)
```

```
3.0278584957122803
```

```
## Write out Parquet-formatted data.
data_from_csv.to_parquet(os.path.join('..', 'data', 'airline.parquet'))

## Read from Parquet.
t0 = time.time()
data_from_parquet = pd.read_parquet(os.path.join(
                                    '..', 'data', 'airline.parquet'))
print(time.time() - t0)
```

```
1.0865087509155273
```

The CSV file is 51 MB while the Parquet file is 8 MB.

```
ls -l ../data/airline.csv
ls -l ../data/airline.parquet
```

```
-rw-r--r-- 1 paciorek scfstaff 51480244 Aug 11  2025 ../data/airline.csv
-rw-r--r-- 1 paciorek scfstaff 8132907 Aug 21 14:56 ../data/airline.parquet
```

## 6. Regular expressions

Regular expressions ("regex") are a *domain-specific language* for finding and manipulating patterns of characters and are a key tool used in UNIX commands such as `grep`, `sed`, and `awk` as well as in scripting languages such as Python and R.

For the moment we'll focus on learning regular expression syntax in the context of the shell, but in Unit 5, we'll also use regular expressions within Python using the `re` package.

The basic idea of regular expressions is that they allow us to find matches of strings or patterns in strings, as well as do substitution. Regular expressions are good for tasks such as:

- extracting pieces of text;
- creating variables from information found in text;
- cleaning and transforming text into a uniform format; and
- mining text by treating documents as data.

Please see the bash shell tutorial for a description of regular expressions. I'll assign a small set of regex problems due as an assignment, and we'll talk through those problems in class to explore the regex syntax.

Other resources include:

- Here's a website where you can interactively test regular expressions on example strings.
- Duncan Temple Lang (UC Davis Statistics) has written a nice tutorial covering regular expressions, illustrated in R.
- Sections 9.9 and 11 of Paul Murrell's book
- The back/second page of RStudio's `stringr` cheatsheet has a cheatsheet on regular expressions.

In addition to the regular expression functionality we'll cover, there is a lot more advanced functionality we won't cover, such as callbacks, named groups, recursion, word boundaries, and lookaround assertions.

## Versions of regular expressions

One thing that can cause headaches is differences in version of regular expression syntax used. As discussed in `man grep`, *extended regular expressions* are standard, with *basic regular expressions* providing less functionality and *Perl regular expressions* additional functionality.

The bash shell tutorial provides a full documentation of the *extended regular expressions* syntax, which we'll focus on here. This syntax should be sufficient for most usage and should be usable in Python and R, but if you notice something funny going on, it might be due to differences between the regular expressions versions.

- In bash, `grep -E` (or `egrep`) enables use of the extended regular expressions, while `grep -P` enables Perl-style regular expressions.
- In Python, the `re` package provides syntax "similar to" Perl.
- In R, `stringr` provides *ICU regular expressions* (see `help(regex)`), which are based on Perl regular expressions.

More details about Perl regular expressions can be found in the regex Wikipedia page.

## General principles for working with regex

The syntax is very concise, so it's helpful to break down individual regular expressions into the component parts to understand them. As Murrell notes, since regex are their own language, it's a good idea to build up a regex in pieces as a way of avoiding errors just as we would with any computer code. `re.findall` in Python and `str_detect` in R's `stringr`, as well as regex101.com are particularly useful in seeing *what* was matched to help in understanding and learning regular expression syntax and

debugging your regex. As with many kinds of coding, I find that debugging my regex is usually what takes most of my time.

## Challenge problem

*Challenge*: Let's think about what regex syntax we would need to detect any number, integer- or real-valued. Let's start from a test-driven development perspective of writing out test cases including:

- various cases we want to detect,
- various tricky cases that are not numbers and we don't want to detect, and
- "corner cases" – tricky (perhaps unexpected) cases that might trip us up.