# Lecture 3 — Computer systems and the command line

C1 Research Computing — MPhil in Data Intensive Science, Michaelmas 2026

## Contents


**1 The stored-program computer** **2**
- 1.1 Computing before the stored program . . . 2
- 1.2 The von Neumann model . . . 3
- 1.3 The memory hierarchy . . . 4
- 1.4 Core memory and disks . . . 6

**2 The operating system and the shell** **6**
- 2.1 The operating system and the kernel . . . 6
- 2.2 The shell . . . 7
- 2.3 Processes and environment variables . . . 8

**3 The file system** **9**
- 3.1 The file system tree . . . 9
- 3.2 File-system formats . . . 10
- 3.3 Files and directories from the shell . . . 11
- 3.4 The data file from the shell . . . 11
- 3.5 Permissions . . . 13

**4 Shell scripts** **13**
- 4.1 Scripts, the shebang and arguments . . . 14
- 4.2 Redirection and exit status . . . 14
- 4.3 Loops over the ten runs . . . 15

**Troubleshooting** **17**

**Key points** **18**

**Course environment** **18**

**Exercises** **19**

**Solutions to the exercises** **20**

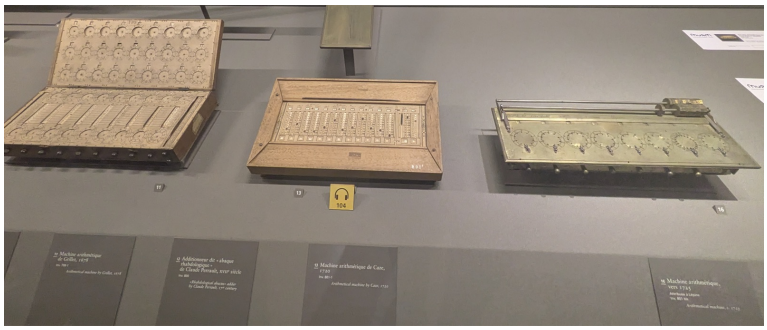

The predator–prey experiment of Blasius et al. [1] was run ten times. The `attractor` project holds the ten runs as the files `C1.csv` to `C10.csv`, each a table of densities sampled over several hundred days. Before any Python is written, three questions are worth answering at the terminal: how many samples each run holds, on which day the algae density peaks, and whether the ten files are the ones published on figshare. The script `summarise.sh` answers the first two for one file in four lines of shell, and a loop applies it to all ten. The chapter follows that script downwards: the processor and memory that execute it, the kernel that opens its files, the shell that starts it, the file system in which it finds its data, and the permissions, streams and exit status through which it reports.

The outputs shown were produced on the lecture laptop, a MacBook Pro with an M4 Max processor running macOS 15.6, with zsh 5.9 as the interactive shell and `/bin/bash` 3.2 as the

interpreter of the scripts. Where an output comes from a Linux machine, the text says so. The commands are the same on Linux, with the small differences in options and output format noted where they occur.

# 1 The stored-program computer

## 1.1 Computing before the stored program



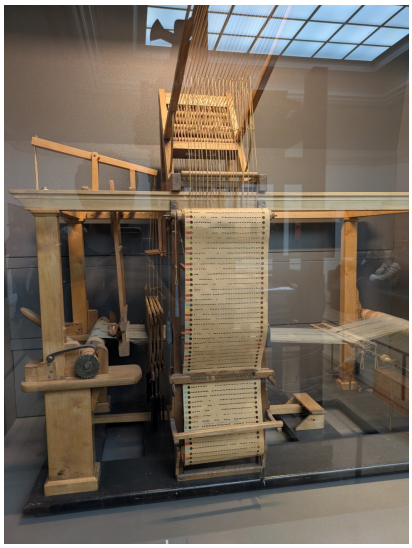


**Figure 1:** Left: arithmetical machines by Grillet (1678), Perrault (17th century) and Caze (1720). Right: a Jacquard loom with its chain of punched cards. Musée des Arts et Métiers, Paris, photographs by the author, October 2026.

The mechanical calculators of the 17th and 18th centuries (Figure 1, left) performed one arithmetic operation at a time, under the hand of an operator. Pascal's calculator of 1642 [2] has one dial per decimal digit. When a wheel completes a turn, it advances the next wheel by one step, which is the carry of written addition implemented in brass. The machines of Grillet, Perrault and Caze in the figure add in the same way. A multiplication is a series of additions, and the operator performs them one after the other, turning the dials and reading the result each time. The machine holds the current numbers. The sequence of operations exists only in the operator's head. Goldstine's history [3] follows this line from Pascal to the machines of 1945.

The Jacquard loom of 1804 [4], on the right of the figure, stored a sequence on punched cards. Each card controls one row of the fabric: a hole lets a hook lift its thread above the shuttle, and without a hole the thread stays below. The chain of cards is the pattern. Replacing the chain changes the pattern without rebuilding the loom, and a workshop could keep its patterns in boxes of cards. Babbage's Analytical Engine, described in 1837 and never completed [5], was designed to read both operations and numbers from cards. Hollerith's tabulators counted the 1890 census of the United States from punched cards [6], and the company that grew out of them became IBM. The IBM card introduced in 1928 had 80 columns and 12 rows and stored one character per column. Programs were supplied as decks of such cards, read by a card reader, into the 1970s.

## 1.2 The von Neumann model

**Definition 1.1:** Stored-program computer

In the model described by von Neumann in 1945 [7, 8] a computer has a **processor**, made of a control unit, an arithmetic-logic unit and registers, one **memory** that holds both the program and its data, and input and output devices. The parts communicate over a **bus**, a shared set of lines carrying addresses, data and control signals. The instructions are bit patterns in memory like the data, so a new program is loaded without any change to the wiring [9]. The processor repeats one cycle for every instruction: fetch it from memory, decode it, execute it.

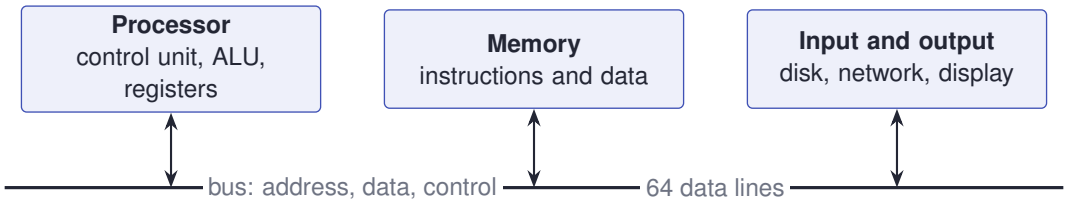



**Figure 2:** The von Neumann model. One bus stands for every connection between the processor, the memory and the devices. A 64-bit data bus moves 64 bits in one transfer.

The Electronic Computer Project of the Institute for Advanced Study in Princeton, led by von Neumann from 1945, built one of the first machines on this model [10]. ENIAC, its contemporary, was configured with cables and switches, and the operators rewired it for each new problem. The stored program removed that step.

The width of the data bus and of the addresses follows the word of the previous chapter. A 64-bit data bus transfers 64 bits at a time. With 32-bit byte addresses a program can name at most $2^{32}$ locations, that is 4 GiB of address space. Some 32-bit systems reached more physical memory through extensions such as Intel's PAE [11], and the 64-bit addresses of today remove the limit in practice. The single bus of Figure 2 is a simplification. A current computer has several interconnects [12]: memory channels connect the processor to its RAM, and PCI Express links [13], point-to-point links of 1, 4, 8 or 16 lanes, connect it to devices such as GPUs and NVMe drives. The lecture workstation of 2026 is one example: an AMD Threadripper PRO 9995WX processor with 8 DDR5 memory channels to 512 GB of RAM and PCIe 5.0 links to two NVIDIA RTX PRO 6000 GPUs and two 4 TB NVMe drives [14]. Its persistent storage also holds a 28 TB hard disk. These figures are those of one machine bought in 2025.

**Definition 1.2:** Core and hardware thread

A **core** is a unit of a processor with its own control logic, arithmetic units and registers, executing its own stream of instructions, and a processor contains several. A **hardware thread** is one such instruction stream with its own register state. A core built for simultaneous multithreading [15] runs two hardware threads that share its execution units. The operating system counts each hardware thread as one **logical CPU**, and their number is the number of processes that can run at the same instant.

Listing 1 is the report of the workstation's processor by `lscpu` [16]: one socket, 96 cores, 2 threads per core, hence 192 logical CPUs. Listing 2 is the same information on the lecture laptop, where `sysctl` reads the kernel's settings and `-n` prints the values without their names. The M4 Max has 16 cores, 12 performance and 4 efficiency cores, and no simultaneous multithreading, so `hw.logicalcpu` equals `hw.physicalcpu`. A Monte Carlo sampler with 16 walkers occupies the laptop fully, and the workstation takes 192 such processes.

The number of simulations a machine runs at once is its number of logical CPUs, which is not always twice the number of cores. Apple silicon has no simultaneous multithreading: the 16 cores of the lecture laptop are 16 hardware threads, and `htop` shows 16 bars. The workstation's 192 logical CPUs come from 96 cores running 2 threads each.

```
$ ssh orion lscpu
Model name:          AMD Ryzen Threadripper PRO 9995WX
CPU(s):              192
Thread(s) per core:  2
Core(s) per socket:  96
Socket(s):           1
$ ssh orion free -h
Mem:                 502Gi total
```

**Listing 1:** The processor and memory of the Linux workstation, reported by `lscpu` and `free -h` over `ssh`. Output of 13 October 2026, selected lines.

```
$ sysctl -n machdep.cpu.brand_string hw.physicalcpu hw.logicalcpu
Apple M4 Max
16
16
$ sysctl -n hw.perflevel0.physicalcpu hw.perflevel1.physicalcpu
12
4
```

**Listing 2:** Cores and logical CPUs of the lecture laptop. The two `perflevel` values are the performance and efficiency cores.

### 1.3 The memory hierarchy

A processor executes an instruction in about one **cycle**, one tick of its clock. At 3 to 4 GHz a cycle lasts 0.25 to 0.33 ns, and the performance cores of the M4 Max run at up to 4.5 GHz, hence 0.22 ns. Memory is slower than the processor by factors that grow with distance from it, and the levels of Table 1 exist to hide that gap [17].

- The **registers** are the few dozen storage slots inside a core on which instructions operate directly [18]. A 64-bit processor has 64-bit general-purpose registers, 16 of them per hardware thread in x86-64 and 31 in arm64, and wider vector registers.
- The **caches** are small SRAM memories, L1, L2 and sometimes L3, holding copies of data recently fetched from RAM [19]. L1 is private to a core and holds tens of KiB. L2 is larger and shared by a group of cores on some processors. A cache hit costs a few cycles. A RAM access costs hundreds.
- **RAM**, the main memory, is built from DRAM chips. An access takes about 100 ns, roughly 300 cycles at 3 GHz. Its contents are lost at power-off.
- A **solid-state drive**, SSD, stores files as electric charge in flash-memory cells [20, 21]. It has no moving part and answers in about 100 µs.
- A **hard disk drive**, HDD, stores bits as magnetised regions of spinning platters, read by a head that moves across the tracks [22]. A seek takes several milliseconds. Data on another machine, over the network, is in the same range.

Nothing in an SSD is magnetic. Each bit is a charge held in a flash-memory cell, which is why the drive has no moving part and no seek time. The magnetic medium is the hard disk, whose head must move to the track before the bits under it can be read.

**Table 1:** The memory hierarchy, with typical capacities and access times of the hardware generations of 2024 to 2026 [23]. The cycle counts assume 3 GHz.

| level | contents | capacity | access time |
|---|---|---|---|
| registers | values used directly by instructions | 16 to 31 of 64 bits per hardware thread | 1 cycle, 0.3 ns |
| L1 cache | copies of recent data, per core | 32 to 128 KiB | 4 cycles, 1 ns |
| L2, L3 cache | larger copies, per core or shared | 1 to 16 MiB, tens of MiB | 4 to 20 ns |
| RAM | main memory, DRAM chips | 8 to 128 GiB on a laptop, 256 to 512 GiB on a cluster node | 300 cycles, 100 ns |
| SSD | files, flash memory | TB | $10^5$ cycles, 100 µs |
| HDD, network | files on spinning platters, on another machine | TB to PB | $10^7$ cycles, 1 to 10 ms |

Listing 3 reads the laptop's figures. The RAM is $137\,438\,953\,472 = 128 \times 2^{30}$ bytes, 128 GiB. The L1 data cache of a performance core is 128 KiB and the L2 cache shared by the performance cores 16 MiB. The efficiency cores have 64 KiB and 4 MiB, and the generic `hw.l1dcachesize` reports their value. The disk, a 1 TB SSD of $10^{12}$ bytes, appears as 926 GiB in `df -h` [24], because `df` counts in powers of two and macOS reserves part of the disk. Apple silicon does not report its clock rate through `sysctl`, and `sudo powermetrics --samplers cpu_power -n 1` prints the live frequency of each core cluster instead. On Linux, `lscpu` prints the model, cores, frequency and cache sizes, `free -h` [25] the memory, `df -h` the disks and `cat /proc/cpuinfo` one block per logical CPU.

```
$ sysctl -n hw.memsize hw.perflevel0.l1dcachesize hw.perflevel0.l2cachesize
137438953472
131072
16777216
$ df -h /
Filesystem        Size    Used   Avail Capacity iused ifree %iused  Mounted on
/dev/disk3s1s1   926Gi    17Gi    15Gi    53%    426k  162M    0%   /
```

**Listing 3:** RAM, caches and disk of the lecture laptop. `sysctl -n` prints kernel values without their names, `df -h` the size of the file system mounted at `/` in binary units.

The two unit families of the previous chapter meet again in RAM specifications. A disk sold as 512 GB holds $512 \times 10^9$ bytes, which is 476.8 GiB. A RAM module sold as 64 GB holds $64 \times 2^{30}$ bytes, so the workstation's eight modules provide 512 GiB nominal, and `free -h` in Listing 1 reports 502 GiB. The exact `MemTotal` is 502.99 GiB. About 9 GiB is reserved by the firmware and the devices before Linux reports its usable total [26].

The hierarchy decides where working data belongs. A program whose inputs sit on a hard disk waits milliseconds for every access, and a fast algorithm is then bounded by the disk. The usual workflow copies the data set from the large disk to the SSD, runs there, and keeps the archive on the disk. On the workstation the two are separate directories: a scratch area on the two 4 TB NVMe drives for the data in use and a research data store on the 28 TB disk for everything else. A NumPy array, one contiguous block, lets the caches serve consecutive elements from one fetch. A Python list holds references to objects scattered through memory, and each element is a separate fetch.

### 1.4 Core memory and disks

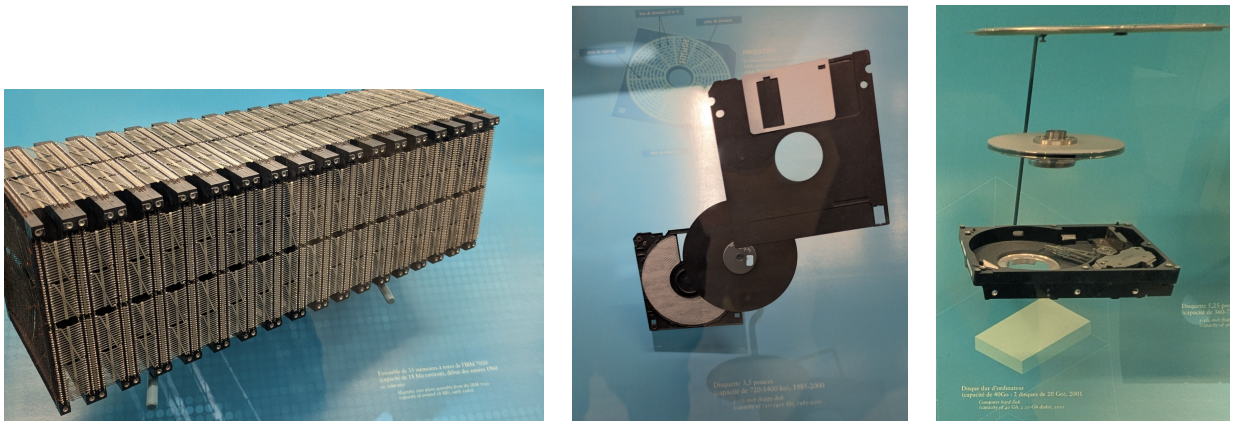


**Figure 3:** Left: magnetic-core planes from an IBM 7030 Stretch, early 1960s. Middle: 3.5-inch floppy disks, 1985 to 2000. Right: a 40 GB hard disk of 2001 with its two platters. Musée des Arts et Métiers, Paris, photographs by the author, October 2026.

Magnetic-core memory [27] was the main memory of computers from the mid-1950s to the early 1970s: one ferrite ring per bit, threaded by wires, the direction of its magnetisation being the 0 or the 1. The planes of Figure 3 come from an IBM 7030 Stretch of 1961 [28], whose memory held at most $262\,144 = 2^{18}$ words. Each word carried 64 data bits and 8 check bits, 72 cores, so the full memory was $2^{18} \times 72 = 18\,874\,368$ cores, about 19 megabits, holding $2^{18} \times 8 = 2\,097\,152$ bytes, 2 MiB, of data. The 128 GiB of the lecture laptop is $2^{37}/2^{21} = 65\,536$ times that maximum, in a few chips of DRAM. The core planes on display are one part of the Stretch memory.

> A core plane stores one bit per ring, so its capacity is counted in bits: the Stretch memory of 18.9 million cores is 18 megabits, about 2 MiB with the check bits removed, not 18 megabytes.

> Core memory is no longer in use. It kept its contents without power and tolerated radiation, which is why it flew in the Apollo Guidance Computer and in the Space Shuttle's computers until 1991. The International Space Station uses radiation-hardened semiconductor memory.

The first hard disk, the IBM 350 of 1956 [29], stored 5 million characters on fifty platters of 24 inches. The 3.5-inch floppy disk of 1985 to 2000 [30] held 1.44 MB on a flexible magnetic disc. The hard disk of 2001 in the figure holds 40 GB on two rigid platters, read by a head that moves across the tracks, which is why an access takes milliseconds. The lecture laptop's 1 TB SSD answers in about 100 µs, a thousand times slower than its RAM and a hundred times faster than a spinning disk.

## 2 The operating system and the shell

### 2.1 The operating system and the kernel

**Definition 2.1:** Kernel and system call

An **operating system** manages the hardware and provides services to programs [31]. Its **kernel** is the privileged program at its centre [32]: it schedules processes on the logical CPUs, manages the virtual memory and the file systems, and controls access to the devices and the network. A program obtains each of these services through a **system call**, a request to the kernel to open a file, read bytes, allocate memory or start a process. Utilities, libraries

and graphical interfaces are the rest of the operating system.

Listing 4 shows the path of one request. The Python expression `open("C9.csv").readline()` asks Python to open the file and return its first line. Python asks the kernel through the `open` and `read` system calls. The kernel checks the permissions of Section 3.5, finds the bytes on the disk and returns them. The expression is Python and runs at Python's `>>>` prompt or through `python3 -c`. Typed at a shell prompt it is parsed by the shell, where the parentheses have another meaning, and Python never runs.

```
$ python3 -c 'print(open("C9.csv").readline()[:34])'
time (days), algae (10^6 cells/ml)
$ open("C9.csv").readline()
zsh: parse error near `()'
```

**Listing 4:** The first line of `C9.csv` through a system call. The same characters typed at the zsh prompt are a shell parse error.

macOS and Linux are both descendants of Unix, and the commands of this chapter run on both. The lecture laptop runs macOS 15.6, Sequoia. The CSD3 cluster runs Rocky Linux 8 [33], a rebuild of Red Hat Enterprise Linux. A Linux distribution packages the Linux kernel with its utilities, and Ubuntu, Rocky Linux and Debian differ in those packages and in their release schedules rather than in the kernel. Windows is outside the chapter, and the Windows Subsystem for Linux gives it a Linux distribution in which everything here applies.

The version of macOS is read with `sw_vers`, that of a Linux distribution from `/etc/os-release`. The macOS numbering is not continuous: Sierra was version 10.12 of 2016, Ventura version 13 of 2022, Sequoia version 15 of 2024 and Tahoe version 26 of 2025, so a number alone does not place a release in time [34].

## 2.2 The shell

**Definition 2.2:** Shell

A **shell** is a program that reads a command line, finds the named program, starts it with the given arguments, waits for it, then prints a prompt and reads the next line [35]. A **command line** is a program name followed by options and arguments: `ls -l C9.csv` runs `ls` with the option `-l` and the argument `C9.csv`. The shell finds the program by trying the directories of the `PATH` variable from left to right and running the first file of that name it finds [36]. A **terminal** is the application providing the window and the keyboard, such as Terminal, iTerm or the terminal panel of VS Code. The shell is a separate program running inside it.

zsh is the default shell of macOS [37] and bash [38] the usual one on Linux, including CSD3. The two agree on everything in this chapter. Short options are one dash and one letter, `-l`, and long options two dashes and a word, `--help`. `man ls` opens the reference manual of a command, which `q` closes. On Linux most commands print a summary with `--help`. The commands of macOS come from BSD rather than from the GNU coreutils [39], and many of them do not accept `--help`.

The shell, not the kernel, searches `PATH`. The kernel executes the file the shell hands to it and knows nothing of `PATH`. The command `type python` prints which file the shell would run, or that the name is a builtin, as in Listing 5. A missing program or a venv that is not activated is therefore a shell problem, visible in `PATH`.

```
$ which ls git python3.13
/bin/ls
/usr/bin/git
/opt/homebrew/bin/python3.13
$ type ls python3 cd
ls is /bin/ls
python3 is /usr/bin/python3
cd is a shell builtin
```

**Listing 5:** Where the shell finds a command. `which` prints the first match in `PATH` and `type` also recognises the builtins, commands executed by the shell itself.

A shell reads its **startup files** before its first prompt. Bash reads `~/.bash_profile` at login and `~/.bashrc` for each interactive shell [40]. zsh reads `~/.zprofile` and `~/.zshrc` [41]. They hold settings such as additions to `PATH`, the prompt and aliases, and the line `alias ll='ls -l'` defines a command `ll` that expands to `ls -l`. The prefix `~/` means the file is in the home directory. A startup file is also where a variable that must exist in every terminal is set, for instance the API key of a service, and such a file is not shown on a screen or committed to a repository. The next chapter keeps project settings in a file of their own instead.

## 2.3 Processes and environment variables

**Definition 2.3:** Process and environment variable

A **process** is a running program, with its own memory and a **process ID**, PID, assigned by the kernel [42]. The shell starts a child process for each command and waits for it, unless the command ends with `&`, which puts it in the background. An **environment variable** is a name–value pair that every process inherits from the process that started it [43]. `PATH` and `HOME` are environment variables, and activating a venv sets `VIRTUAL_ENV` and prepends the venv to `PATH`.

`ps` lists the processes attached to the terminal and `top` or `htop` display all of them, refreshed live, with their memory and CPU use. `Ctrl-C` sends an interrupt to the foreground process. `kill` sends the termination signal to the process whose PID it is given [44]. Listing 6 starts a `sleep` in the background, lists it and ends it. The shell variable `$!` holds the PID of the last background command and `$$` that of the shell itself.

`kill` takes a process ID, not a name: `kill 29230` ends the one process with that PID, read from `ps` or `htop`. To act on a name, `pkill python` sends the signal to every process called `python`, which may be more processes than intended.

`echo $PATH` prints the value of `PATH`, the dollar sign asking for the value of the name that follows. `printenv` prints every environment variable with its value, and `printenv HOME` one of them. `export NAME=value` sets a variable in the shell and marks it for inheritance by every process the shell starts afterwards, as in Listing 7. Without `export` the variable exists in the shell only.

```
$ ps -o pid,tty,time,comm -p $$
  PID TTY           TIME COMM
29203 ttys002    0:00.01 zsh
$ sleep 300 &
[1] 29230
$ ps -o pid,tty,time,command -p $!
  PID TTY           TIME COMMAND
29230 ttys002    0:00.00 sleep 300
$ kill $!
[1]  + terminated  sleep 300
```

**Listing 6:** A background process, listed and ended by its PID. The `-o` option of `ps` chooses the columns.

```
$ echo $PATH | tr ':' '\n' | head -4
/Users/boris/.local/bin
/opt/homebrew/bin
/opt/homebrew/sbin
/Users/boris/Library/pnpm
$ printenv HOME
/Users/boris
$ export MYVAR=1 ; echo $MYVAR
1
```

**Listing 7:** Reading and setting environment variables. The first entries of `PATH` on the lecture laptop are those added by the startup files, and the system directories follow.

# 3 The file system

## 3.1 The file system tree

**Definition 3.1:** File system and path

A **file system** organises stored data into named files and directories [45]. The operating system presents every file system as one tree rooted at `/`. A **path** names a file by its position in the tree. An **absolute path** starts at the root, `/Users/boris/attractor/C9.csv`. A **relative path** starts at the **current directory** of the process, `C9.csv` or `../attractor/C9.csv`, where `.` is the current directory, `..` its parent and `~` the home directory of the user. A second disk or a network share is **mounted** at a directory of the tree, its **mount point**, and its files are reached through that directory [46].

The Filesystem Hierarchy Standard [47] fixes the roles of the top-level directories of Linux (Table 2), and macOS shares most of the names. Listing 8 looks at a few of them on the lecture laptop. `ls /` lists the root. `ls /usr/bin | wc -l` counts 915 programs, with `wc -l` counting the lines that `ls` printed. `tail -2 /etc/shells` prints the last two shells a user may choose. `ls -ld /tmp` lists the directory entry itself rather than its contents, and the leading `l` of the mode shows that `/tmp` on macOS is a symbolic link [48], a name that points to another path, here `/private/tmp`. The `c` of `/dev/null` marks a character device: the file discards whatever is written to it and reads as empty.

A USB stick appears in two places. The device gets a node under `/dev`, `/dev/disk4` on macOS, and its file system is mounted at a directory, `/Volumes/<name>` on macOS, `/media` or `/mnt` on Linux. Files are reached through the mount point, never through the device node.

**Table 2:** Top-level directories of the file system tree [47]. The last three rows are macOS and cluster conventions outside the standard.

| directory | contents | example |
|---|---|---|
| `/bin`, `/usr/bin` | the system's programs, its binaries | `/bin/ls`, `/usr/bin/git` |
| `/sbin`, `/usr/sbin` | the administrator's programs | `/sbin/mount` |
| `/usr` | programs, libraries in `/usr/lib`, manual pages, `/usr/local` for local installs | `/usr/share/man` |
| `/opt` | optional add-on software, one folder per package | `/opt/homebrew/bin/python3.13` |
| `/etc` | the configuration of the system, as text files: the names of machines, the shells a user may choose, the defaults of `ssh` | `/etc/hosts`, `/etc/shells`, `/etc/ssh/ssh_config` |
| `/var` | variable data written while the system runs: logs, databases, mail, print queues | `/var/log` |
| `/tmp` | temporary files, cleared at reboot | |
| `/dev` | the devices, presented as files | `/dev/disk3s1s1`, `/dev/null` |
| `/home`, `/root` | home directories on Linux, the administrator's | `/home/<crsid>` |
| `/proc`, `/sys` | kernel and process information, Linux only | `/proc/cpuinfo` |
| `/mnt`, `/media` | mount points for other disks on Linux | |
| `/Users`, `/Volumes` | home directories and mount points on macOS | `/Users/boris` |
| `/Applications`, `/Library`, `/System` | applications and support files on macOS | `~/Library/Caches` |
| `/rds` | the research data store of a cluster | `/rds/user/<crsid>/hpc-work` |

```
$ ls /
Applications    etc             private         Users
bin             home            sbin            usr
cores           Library         System          var
dev             opt             tmp             Volumes
$ ls /usr/bin | wc -l
     915
$ tail -2 /etc/shells
/bin/tcsh
/bin/zsh
$ ls -ld /tmp
lrwxr-xr-x@ 1 root  wheel  11 16 Aug  2025 /tmp -> private/tmp
$ ls -l /dev/null
crw-rw-rw-  1 root  wheel  0x3000002 20 Oct 23:06 /dev/null
```

**Listing 8:** The root of the tree on the lecture laptop. In `ls -l` the first character of the mode is `d` for a directory, `l` for a symbolic link, `c` for a character device and `-` for a regular file.

## 3.2 File-system formats

A **file-system format** specifies how names, data and metadata are laid out on a volume. Formats differ in case sensitivity, in the largest file they allow and in the permission metadata they store (Table 3). The paths look the same whatever the format underneath, which is why the format is easy to forget until a file name or a permission bit behaves differently on another machine. Listing 9 asks macOS for the format of its system volume: `diskutil info /` describes the volume mounted at `/` and `grep Personality` keeps the one line naming the format. `mount` lists every mounted volume with its mount point and format. On Linux, `df -Th` adds the for-

mat in a Type column. The macOS system volume is sealed and read-only, and user files live on a separate data volume.

**Table 3:** File-system formats met in research computing.

| format | used on | properties |
|---|---|---|
| APFS [49] | macOS, the disk of the lecture laptop | case-insensitive by default, journaled |
| ext4 [50], XFS [51] | Linux: Ubuntu, Rocky Linux, cloud machines | case-sensitive, journaled |
| NTFS [52] | Windows | read by macOS, not written without extra software |
| exFAT [53], FAT32 | USB sticks, SD cards | no owner and no execute bit, so `chmod` has no effect. FAT32 limits a file to less than 4 GiB |
| NFS [54] | network: the home directories of a cluster | one server's disk, seen by every node |
| Lustre [55] | parallel: the research data store of a cluster | data spread over many servers, read and written by many jobs at once |

A journaled format records each change before applying it, so that the record can be replayed after a crash. A network file system makes one server's disk visible on many machines. A parallel file system such as Lustre distributes the data of one file system across servers so that many jobs can read and write at once, which is what a cluster needs for the input and output of its jobs.

> APFS is case-insensitive by default: on the lecture laptop `C9.csv` and `c9.csv` name the same file, and a directory cannot hold both. Linux file systems, including those of CSD3, are case-sensitive. A script that works on the laptop can fail on the cluster because of one capital letter in a file name.

```
$ diskutil info / | grep Personality
   File System Personality:   APFS
$ mount | head -1
/dev/disk3s1s1 on / (apfs, sealed, local, read-only, journaled)
```

**Listing 9:** The format of the system volume of the lecture laptop.

### 3.3 Files and directories from the shell

Listing 10 sets up the project folder. `cd` changes the current directory and `pwd` prints it as an absolute path. `curl -LO` downloads a file: `-L` follows redirects and `-O` saves the file under its remote name [56]. `unzip -j` extracts the files of an archive without their directory paths, and `-q` silences the listing. `A && B` runs `B` only when `A` succeeds, so the archive is deleted only after a successful extraction. `ls -l` lists the files in long format and `ls -a` includes the hidden names, those starting with a dot. `mkdir` creates a directory, `cp` copies a file, `mv` moves or renames it, `rm -r` removes a directory with everything in it. Removed files cannot be restored from the command line. Names containing spaces are quoted, `"my file.csv"`.

### 3.4 The data file from the shell

The questions of the opening are answered by a few commands and a pipe (Listing 11). `wc -l` counts the lines of a file [57]: 142, the header line plus 141 samples. `head -3` prints the first three lines. `cut -d, -f1,2` keeps fields 1 and 2 of each line, the fields being separated by

```
$ cd ~/attractor
$ curl -sLO https://teaching.borisbolliet.io/c1-2026/materials/class03-demo.zip
$ unzip -jq class03-demo.zip && rm class03-demo.zip
$ ls -l
total 216
-rw-r--r--@ 1 boris  staff  14347 13 Oct 11:55 C1.csv
-rw-r--r--@ 1 boris  staff   7693 13 Oct 11:55 C10.csv
-rw-r--r--@ 1 boris  staff   7407 13 Oct 11:55 C2.csv
-rw-r--r--@ 1 boris  staff   7074 13 Oct 11:55 C3.csv
-rw-r--r--@ 1 boris  staff   5499 13 Oct 11:55 C4.csv
-rw-r--r--@ 1 boris  staff   3306 13 Oct 11:55 C5.csv
-rw-r--r--@ 1 boris  staff   9406 13 Oct 11:55 C6.csv
-rw-r--r--@ 1 boris  staff   9505 13 Oct 11:55 C7.csv
-rw-r--r--@ 1 boris  staff   7071 13 Oct 11:55 C8.csv
-rw-r--r--@ 1 boris  staff   5566 13 Oct 11:55 C9.csv
-rwxr--r--@ 1 boris  staff    255 13 Oct 11:55 fetch_runs.sh
-rw-r--r--@ 1 boris  staff     61 13 Oct 11:55 period.py
-rw-r--r--@ 1 boris  staff    121 13 Oct 11:55 runs.txt
-rwxr--r--@ 1 boris  staff    249 13 Oct 11:55 summarise.sh
$ mkdir scratch && cp C9.csv scratch/copy.csv && ls scratch
copy.csv
$ mv scratch/copy.csv scratch/C9_copy.csv
$ rm -r scratch
$ ls scratch
ls: scratch: No such file or directory
```

**Listing 10:** Fetching the ten runs into the project folder and moving files about. The `@` after the mode marks a macOS extended attribute, here the record of where the file came from.

commas [58]. `grep -c ",0.00$"` counts the lines matching a pattern [59], here the 68 samples whose last field, the external medium, is zero. `sort -t, -k2 -n` sorts the lines numerically by their second field [60], and `tail -1` keeps the last one, the sample with the largest algae density: 2.78 million cells per ml on day 14. The **pipe** `|` connects the output of one command to the input of the next, so that the four programs of the last line run as one assembly line without any intermediate file.

> `wc -l`, with a lower-case L, counts lines. `wc -w` counts words and `wc -c` bytes. There is no `-n` option, and `wc -n C9.csv` is an error.

```
$ wc -l C9.csv
     142 C9.csv
$ head -3 C9.csv | cut -d, -f1,2
time (days), algae (10^6 cells/ml)
0.00,1.92
1.02,1.58
$ grep -c ",0.00$" C9.csv
68
$ sort -t, -k2 -n C9.csv | tail -1 | cut -d, -f1,2
14.00,2.78
```

**Listing 11:** Four questions about `C9.csv` answered without Python.

### 3.5 Permissions

**Definition 3.2:** Permission mode

On a Unix file system each file has an owning user, an owning group and a **permission mode** of nine bits: read, write and execute for the owner, for the group and for all other users [61]. `ls -l` prints the mode as three triplets after the type character, `rw-r--r--`. Each triplet is also read as an octal digit, read counting 4, write 2 and execute 1, so that `rw-r--r--` is 644 and `rwxr-xr-x` is 755. For a directory, execute means that the directory may be entered.

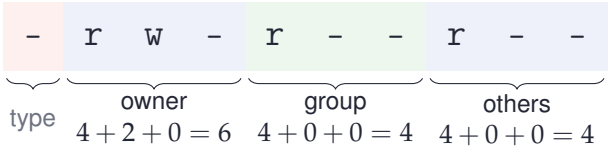



**Figure 4:** The mode string `-rw-r--r--` of `period.py` as printed by `ls -l`: the type character, then read, write and execute for the owner, the group and the others. The octal form is 644.

Table 4 lists the triplets that occur in practice. `chmod` changes the mode either symbolically, `chmod u+x period.py` adding execute for the user who owns the file, with `g` and `o` for the group and the others and `-x` removing, or numerically, `chmod 644 file` setting `rw-r--r--`. In Listing 12 the mode of `period.py` goes from 644 to 744. The execute bit lets the kernel start the file as a program, which is how `./period.py` differs from `python period.py`: the second needs only read permission, because the program being started is Python. A file started directly must also say which interpreter runs it, in its first line, which is the subject of the next section. `sudo` runs one command as the administrator. Nothing in this course needs it inside the home directory or a venv, and on CSD3 it is not available.

**Table 4:** Permission triplets and their octal digits, and the modes they combine into.

| triplet | digit | meaning | typical use |
|---|---|---|---|
| `rwx` | 7 | read, write, execute | owner of a script, 744 or 755 |
| `rw-` | 6 | read, write | owner of a data file, 644 |
| `r-x` | 5 | read, execute | group and others of a program, 755 |
| `r--` | 4 | read only | group and others of a data file, 644 |
| `---` | 0 | no access | others of a private file, 600 or 700 |

```
$ ls -l period.py
-rw-r--r--@ 1 boris  staff  61 13 Oct 11:55 period.py
$ chmod u+x period.py && ls -l period.py
-rwxr--r--@ 1 boris  staff  61 13 Oct 11:55 period.py
$ ./period.py
./period.py: line 1: syntax error near unexpected token ‘(’
./period.py: line 1: ‘def period(freq):’
$ python3 period.py
7.8125
```

**Listing 12:** Adding the execute bit for the owner. Without a first line naming the interpreter, the executable file is run as a shell script and fails at its first Python line.

## 4 Shell scripts

### 4.1 Scripts, the shebang and arguments

A **shell script** is a text file of shell commands, run in order by a new shell process [62]. Listing 13 is `summarise.sh`, which prints the number of samples and the peak algae density of one run, and the same script runs unchanged on any of the ten files.

- The first line, the **shebang** `#!/bin/bash`, names the interpreter that the kernel starts to run the file. `#!/usr/bin/env bash` finds bash through `PATH` instead of a fixed location. Direct execution as `./summarise.sh` requires the execute bit. `bash summarise.sh` requires only read permission.
- `$1`, `$2` and so on are the arguments given on the command line, `$#` is their number and `"$@"` all of them, each kept whole [63]. A variable is set as `file="$1"`, without spaces around `=`, and read as `$file`. Every variable is written between double quotes, `"$file"`. Unquoted, a value with a space such as `my file.csv` is split into two arguments.
- `$(command)` is replaced by the output of the command, a **command substitution**. `$((expression))` evaluates an integer expression, an **arithmetic expansion**.
- `set -euo pipefail` [64] makes the script stop at the first failing command, `-e`, treat an unset variable as an error, `-u`, and make a pipeline fail when any of its commands fails, `-o pipefail`. A missing first argument therefore stops the script at `file="$1"`, as the last command of the listing shows.

`cat` prints a file. `wc -l < "$file"` reads the file from standard input, so that `wc` prints the count alone, without the file name, and the arithmetic expansion subtracts the header line: $142 - 1 = 141$ samples. The `sort | tail | cut` pipeline is the one of Section 3.4, applied to whichever file the script received. Its output is stored in `peak` and printed by `echo` inside a double-quoted string, in which `$file`, `$n` and `$peak` are replaced by their values.

```
$ cat summarise.sh
#!/bin/bash
# summarise.sh -- number of samples and peak algae density of one run
set -euo pipefail
file="$1"
n=$(( $(wc -l < "$file") - 1 ))
peak=$(sort -t, -k2 -n "$file" | tail -1 | cut -d, -f1,2)
echo "$file: $n samples, peak (day,algae) $peak"
$ ls -l summarise.sh
-rwxr--r--@ 1 boris  staff  249 13 Oct 11:55 summarise.sh
$ ./summarise.sh C9.csv
C9.csv: 141 samples, peak (day,algae) 14.00,2.78
$ ./summarise.sh
./summarise.sh: line 4: $1: unbound variable
```

**Listing 13:** The summarising script, run on `C9.csv` and then without its argument. The archive preserves the execute bit of the script. A script written in an editor starts without it and needs `chmod u+x`.

### 4.2 Redirection and exit status

**Definition 4.1:** Standard streams and exit status

Every process has three **standard streams**: standard input, `stdin`, number 0, from which it reads, standard output, `stdout`, number 1, to which it writes its results, and standard error, `stderr`, number 2, to which it writes its messages. In a terminal all three are the terminal, and the shell can connect each of them to a file or to another process [65]. Every

command ends with an **exit status**, an integer that is 0 for success and 1 to 255 for a failure [66]. The shell keeps the status of the last command in `$?`.

The redirection operators connect the streams. `command > file` writes the standard output to the file, replacing its contents, and `>>` appends. `< file` reads the standard input from the file. `2> file` writes the error messages to the file. `2>&1` sends stream 2 to wherever stream 1 goes at that point in the command line, so it is written after `> file`: `command > all.txt 2>&1` collects both streams. `> /dev/null` discards the output. The exit status drives the control operators: `A && B` runs `B` only after a 0, `A || B` only after a failure, and `set -e` stops a script at the first non-zero status. Listing 14 builds a summary file line by line, then runs the script on a missing file. The script fails at its line 5, where `wc` cannot open the file. The message goes to `errors.txt`, `set -e` ends the script there, and `$?` reports 1.

Redirections apply from left to right. `command 2>&1 > file` first sends the errors to the terminal, where stream 1 points at that moment, and only then sends stream 1 to the file, so the file receives no error message. The order that collects both streams is `> file 2>&1`.

```
$ ./summarise.sh C9.csv > summary.txt
$ ./summarise.sh C1.csv >> summary.txt
$ cat summary.txt
C9.csv: 141 samples, peak (day,algae) 14.00,2.78
C1.csv: 366 samples, peak (day,algae) 335.05,1.91
$ ./summarise.sh missing.csv 2> errors.txt
$ echo $?
1
$ cat errors.txt
./summarise.sh: line 5: missing.csv: No such file or directory
$ ./summarise.sh C9.csv > all.txt 2>&1
```

**Listing 14:** Output and errors sent to files, and the exit status of a failure.

### 4.3 Loops over the ten runs

A loop repeats commands over a list of values [67]. In `for f in C*.csv; do ...; done` the shell first replaces the pattern `C*.csv` by the names that match it in the current directory, sorted as strings in the order of the current locale, then runs the body once per name with `f` set to it [68]. In a pattern, `*` matches any string, `?` one character and `[0-9]` one digit. The loop is one command: its output as a whole can be piped or redirected, and its exit status is that of its last turn. Listing 15 applies the script to the ten runs and stores the result in `peaks.txt`. The runs hold between 86 and 366 samples. The peak algae density ranges from 1.60 to 8.26 million cells per ml, and in run C6 it is the first sample, on day 0.

The names matching `C*.csv` come sorted as strings, not in the numerical order of the runs, and the string order depends on the system. On macOS, and anywhere under `LC_ALL=C`, bytes are compared and the order is `C1.csv`, `C10.csv`, `C2.csv`. On Linux under `en_US.UTF-8` the collation ignores the dot on its first pass and puts the digit of `C10csv` before the letter of `C1csv`: the order is `C10.csv`, `C1.csv`, `C2.csv`. A loop whose order matters names its files with zero-padded numbers, `C01.csv`, or sorts the list with `sort -V`.

The ten runs are published on figshare with the paper [1], each as a file with a numeric id. The file `runs.txt` lists the id of each run, read from `https://api.figshare.com/v2/articles/10045976/files`, and `fetch_runs.sh` in Listing 16 downloads them with a second kind of

```
$ for f in C*.csv; do ./summarise.sh "$f"; done
C1.csv: 366 samples, peak (day,algae) 335.05,1.91
C10.csv: 198 samples, peak (day,algae) 7.00,2.19
C2.csv: 190 samples, peak (day,algae) 133.00,1.85
C3.csv: 182 samples, peak (day,algae) 94.13,2.12
C4.csv: 139 samples, peak (day,algae) 34.97,1.91
C5.csv: 86 samples, peak (day,algae) 85.43,1.60
C6.csv: 232 samples, peak (day,algae) 0.00,8.21
C7.csv: 235 samples, peak (day,algae) 61.97,8.26
C8.csv: 184 samples, peak (day,algae) 8.90,4.75
C9.csv: 141 samples, peak (day,algae) 14.00,2.78
$ for f in C*.csv; do ./summarise.sh "$f"; done > peaks.txt
$ wc -l peaks.txt
      10 peaks.txt
```

**Listing 15:** One summary line per run, printed and then stored. The order is that of the file names as strings on macOS.

loop. `while read -r name id; do ...; done < runs.txt` runs one turn per line of the file, splitting the line at whitespace into `name` and `id`, with `-r` keeping backslashes literal. The loop ends when `read` finds no more lines. The `curl` options are `-f` to fail on an HTTP error instead of saving the error page as if it were the data, `-s` to hide the progress bar, `-S` to show errors nevertheless, `-L` to follow the redirect from figshare to its storage server and `-o` to name the output file [56]. The figshare link redirects to a storage address that expires after ten seconds, so on a slow connection a download can fail with a 403 or a 404, and rerunning the script requests a fresh link. The downloaded files are byte for byte those shipped in the course archive, as `cmp` confirms.

```
$ cat fetch_runs.sh
#!/bin/bash
# fetch_runs.sh -- download the runs listed in runs.txt from figshare
set -euo pipefail
while read -r name id; do
  echo "$name.csv <- figshare file $id"
  curl -fsSL -o "$name.csv" "https://ndownloader.figshare.com/files/$id"
done < runs.txt
$ head -2 runs.txt
C1      18105239
C2      18105233
$ ./fetch_runs.sh
C1.csv <- figshare file 18105239
C2.csv <- figshare file 18105233
C3.csv <- figshare file 18105230
C4.csv <- figshare file 18105227
C5.csv <- figshare file 18105224
C6.csv <- figshare file 18105221
C7.csv <- figshare file 18105218
C8.csv <- figshare file 18105212
C9.csv <- figshare file 18105215
C10.csv <- figshare file 18105236
$ cmp C9.csv ~/attractor/C9.csv && echo same
same
```

**Listing 16:** Downloading the ten runs from figshare with a `while read` loop. The download of all ten took 2.3 s on 20 October 2026.

## Troubleshooting

Nine messages point back to the concepts of this chapter.

| symptom | cause | remedy |
|---|---|---|
| `No such file or directory` | A relative path is taken from the current directory, which is not the one expected (Definition 3.1). | `pwd` and `ls` show where the shell is and what it sees. Use an absolute path or `cd` first. |
| `Permission denied` on `./script.sh` | The file has no execute bit (Definition 3.2). | `bash script.sh`, or `chmod u+x script.sh`. |
| `command not found` | The program is in no directory of `PATH`, or the venv is not activated (Section 2.2). | `type name` shows what the shell finds. Activate the venv or add the directory to `PATH`. |
| `bad interpreter: No such file or directory` | The script has Windows line endings. The carriage return becomes part of the interpreter's name in the shebang (Section 4.1). | `sed -i '' 's/\r$//' script.sh` on macOS, `sed -i 's/\r$//' script.sh` or `dos2unix script.sh` on Linux. |
| `syntax error near unexpected token` on `./period.py` | The file is executable but has no shebang, so it is run as a shell script (Listing 12). | Add `#!/usr/bin/env python3` as the first line. |
| `unbound variable` | `set -u` found a variable that was never set, often a misspelt name or a missing argument (Section 4.1). | Pass the argument, or give a default with `${1:-default}`. |
| The output file is empty and the messages appeared on the terminal | The redirections were in the wrong order, `2>&1 > file`, or the output went to standard error (Definition 4.1). | `> file 2>&1`. |
| A pipeline failed but the script went on | Without `pipefail` the status of a pipeline is that of its last command (Section 4.1). | `set -o pipefail`, or test each stage. |
| `zsh: no matches found` or `parse error` | The shell interpreted `*`, `?` or parentheses in an argument before the program saw it (Definition 2.2). | Quote the argument, `'a*b'`. |

## Key points

- A program and its data are bit patterns in the same memory, and the processor runs one instruction stream per hardware thread (Definitions 1.1 and 1.2). Each level of memory is slower than the one above by one to three orders of magnitude, from 1 ns in cache to 10 ms on a disk (Table 1).
- The kernel owns the hardware and serves system calls. The shell is an ordinary program that searches `PATH` and starts other programs as processes, which inherit its environment variables (Definitions 2.1, 2.2 and 2.3).
- A path names a position in one tree, absolute from `/` or relative to the current directory. The format underneath decides case sensitivity and which permission bits exist (Definition 3.1, Table 3).
- The nine permission bits read as three octal digits, 644 for a data file and 744 for a script of its owner, and a file runs directly only with the execute bit and a shebang (Definition 3.2).
- A script quotes every variable, starts with `set -euo pipefail`, and reports through its standard streams and its exit status, 0 for success (Definition 4.1). A `for` loop over a pattern visits the names in string order (Listing 15).

## Course environment

The ten runs, the two scripts and `period.py` are in the archive `class03-demo.zip` on the course site, fetched in Listing 10 into the `attractor` folder of Lecture 1. CSD3 runs Rocky Linux 8 with bash. Its home directories, `/home/<crsid>`, are served by NFS, have a small quota and are snapshotted. The research data store, `/rds/user/<crsid>/hpc-work`, is a Lustre file system for the input and output of jobs, while the home directory is for code and small files [69]. `sudo` is not available on CSD3. `top` is part of every macOS and Linux system, and `htop` is installed on the lecture laptop through Homebrew.

## Exercises

Solutions are given from page 20.

### Cycles and access times

The performance cores of the lecture laptop run at up to 4.5 GHz. (a) How long is one cycle? (b) How many cycles does a 100 ns RAM access take, and how many a 100 µs SSD access and a 5 ms disk seek? (c) The Stretch core memory of Section 1.4 holds 2 MiB. How many of its 72-bit words would hold the $141 \times 7$ `float64` values of `C9.csv`, and what fraction of the memory is that?

### The ten runs in one command

Without the script, using only `wc`, `sort`, `tail` and `head`: (a) print the total number of lines of the ten files and deduce the total number of samples. (b) Find the run with the most samples. (c) Explain why `wc -l C*.csv` alone does not give the number of samples.

### Permission modes and the shebang

(a) Write the mode string of 750 and say who can run the file. (b) After `chmod 644 summarise.sh`, what do `bash summarise.sh C9.csv` and `./summarise.sh C9.csv` print, and what is the exit status of the second? (c) Make `period.py` run as `./period.py`.

### Where the error message went

(a) Predict the contents of `out.txt` after `./summarise.sh missing.csv 2>&1 > out.txt` and after `./summarise.sh missing.csv > out.txt 2>&1`. (b) Predict the exit status of `./summarise.sh missing.csv 2>/dev/null | wc -l` in an interactive bash, without and with `set -o pipefail`.

## Solutions to the exercises

### Cycles and access times

(a) $1/4.5$ GHz $= 0.222$ ns. (b) $100\text{ ns}/0.222\text{ ns} = 450$ cycles for RAM, $4.5 \times 10^5$ cycles for the SSD and $2.25 \times 10^7$ cycles for the seek. (c) The array is $141 \times 7 \times 8 = 7\,896$ bytes, that is 987 words of 8 data bytes, out of 262 144: 0.38% of the memory.

**Answer check**

```
>>> 1 / 4.5e9, 100e-9 * 4.5e9, 100e-6 * 4.5e9, 5e-3 * 4.5e9
(2.2222222222222222e-10, 450.0, 450000.0, 22500000.0)
>>> 141 * 7 * 8 // 8, 141 * 7 * 8 / 8 / 2**18
(987, 0.003765106201171875)
```

### The ten runs in one command

(a) `wc -l` prints one line per file and a total: 1963 lines. Each file has one header line, so the ten runs hold $1963 - 10 = 1953$ samples. (b) `sort -n` orders the lines of `wc` by their count, and the last line before the total is the longest run, C1 with 367 lines, 366 samples. (c) Every count includes the header line, and the total includes ten of them.

**Answer check**

```
$ wc -l C*.csv | tail -1
    1963 total
$ wc -l C*.csv | sort -n | tail -2
     367 C1.csv
    1963 total
```

### Permission modes and the shebang

(a) $7 = 4+2+1$, $5 = 4+0+1$, 0: `rwxr-x---`. The owner and the members of the group can run the file, and other users cannot read it. (b) `bash summarise.sh C9.csv` prints the summary line, because bash only needs to read the file. `./summarise.sh C9.csv` fails with `Permission denied`, and the shell reports exit status 126, its code for a command that was found but could not be executed. (c) Add the shebang `#!/usr/bin/env python3` as the first line and set the execute bit.

**Answer check**

```
$ chmod 750 summarise.sh && ls -l summarise.sh
-rwxr-x---@ 1 boris  staff  249 13 Oct 11:55 summarise.sh
$ chmod 644 summarise.sh
$ bash summarise.sh C9.csv
C9.csv: 141 samples, peak (day,algae) 14.00,2.78
$ ./summarise.sh C9.csv
zsh: permission denied: ./summarise.sh
$ echo $?
126
$ printf '#!/usr/bin/env python3\n' | cat - period.py > p2.py
$ chmod u+x p2.py && ./p2.py
7.8125
```

### Where the error message went

(a) With `2>&1 > out.txt` the error stream is joined to the terminal before the output stream is sent to the file, so the message appears on the terminal and `out.txt` is empty. With `> out.txt 2>&1` the file receives the message. (b) Without `pipefail` the status of the pipeline is that of `wc`, which succeeded, so 0. With `pipefail` it is the status of the failed script, 1. In both cases `wc` prints 0, since the script wrote nothing to its standard output.

**Answer check**

```
$ ./summarise.sh missing.csv 2>&1 > out.txt
./summarise.sh: line 5: missing.csv: No such file or directory
$ wc -c out.txt
       0 out.txt
$ ./summarise.sh missing.csv > out.txt 2>&1; cat out.txt
./summarise.sh: line 5: missing.csv: No such file or directory
$ ./summarise.sh missing.csv 2>/dev/null | wc -l; echo $?
       0
0
$ set -o pipefail
$ ./summarise.sh missing.csv 2>/dev/null | wc -l; echo $?
       0
1
```

## References


[1] Bernd Blasius, Lars Rudolf, Guntram Weithoff, Ursula Gaedke, and Gregor F. Fussmann. Long-term cyclic persistence in an experimental predator–prey system. *Nature*, 577:226–230, 2020. Data: https://doi.org/10.6084/m9.figshare.10045976, CC BY 4.0.

[2] Pascal's calculator. Wikipedia, https://en.wikipedia.org/wiki/Pascaline. accessed 20 October 2026.

[3] Herman H. Goldstine. *The Computer from Pascal to von Neumann*. Princeton University Press, Princeton, 1972.

[4] Jacquard machine. Wikipedia, https://en.wikipedia.org/wiki/Jacquard_machine. accessed 20 October 2026.

[5] Analytical engine. Wikipedia, https://en.wikipedia.org/wiki/Analytical_engine. accessed 20 October 2026.

[6] Punched card. Wikipedia, https://en.wikipedia.org/wiki/Punched_card. accessed 20 October 2026.

[7] John von Neumann. First draft of a report on the EDVAC. Technical report, Moore School of Electrical Engineering, University of Pennsylvania, 1945. Reprinted in IEEE Annals of the History of Computing 15(4), 27–75, 1993.

[8] Von neumann architecture. Wikipedia, https://en.wikipedia.org/wiki/Von_Neumann_architecture. accessed 20 October 2026.

[9] Stored-program computer. Wikipedia, https://en.wikipedia.org/wiki/Stored-program_computer. accessed 20 October 2026.

[10] Institute for Advanced Study. Electronic computer project. https://www.ias.edu/electronic-computer-project. accessed 20 October 2026.

[11] Physical address extension. Wikipedia, https://en.wikipedia.org/wiki/Physical_Address_Extension. accessed 20 October 2026.

[12] Bus (computing). Wikipedia, https://en.wikipedia.org/wiki/Bus_(computing). accessed 20 October 2026.

[13] PCI express. Wikipedia, https://en.wikipedia.org/wiki/PCI_Express. accessed 20 October 2026.

[14] AMD. AMD ryzen threadripper PRO 9995wx specifications. https://www.amd.com/en/products/processors/workstations/ryzen-threadripper/9000-wx-series/amd-ryzen-threadripper-pro-9995wx.html. accessed 20 October 2026.

[15] Simultaneous multithreading. Wikipedia, https://en.wikipedia.org/wiki/Simultaneous_multithreading. accessed 20 October 2026.

[16] `lscpu(1)`. Linux man-pages, https://man7.org/linux/man-pages/man1/lscpu.1.html. accessed 20 October 2026.

[17] Memory hierarchy. Wikipedia, https://en.wikipedia.org/wiki/Memory_hierarchy. accessed 20 October 2026.

[18] Processor register. Wikipedia, https://en.wikipedia.org/wiki/Processor_register. accessed 20 October 2026.

[19] CPU cache. Wikipedia, https://en.wikipedia.org/wiki/CPU_cache. accessed 20 October 2026.

[20] Solid-state drive. Wikipedia, https://en.wikipedia.org/wiki/Solid-state_drive. accessed 20 October 2026.

[21] Flash memory. Wikipedia, https://en.wikipedia.org/wiki/Flash_memory. accessed 20 October 2026.

[22] Hard disk drive. Wikipedia, https://en.wikipedia.org/wiki/Hard_disk_drive. accessed 20 October 2026.

[23] Jonas Bonér. Latency numbers every programmer should know. GitHub Gist, https://gist.github.com/jboner/2841832. accessed 20 October 2026.

[24] `df(1)`. Linux man-pages, https://man7.org/linux/man-pages/man1/df.1.html. accessed 20 October 2026.

[25] `free(1)`. Linux man-pages, https://man7.org/linux/man-pages/man1/free.1.html. accessed 20 October 2026.

[26] National Institute of Standards and Technology. Prefixes for binary multiples. https://physics.nist.gov/cuu/Units/binary.html. accessed 20 October 2026.

[27] Magnetic-core memory. Wikipedia, https://en.wikipedia.org/wiki/Magnetic-core_memory. accessed 20 October 2026.

[28] IBM 7030 stretch. Wikipedia, https://en.wikipedia.org/wiki/IBM_7030_Stretch. accessed 20 October 2026.

[29] IBM 350. Wikipedia, https://en.wikipedia.org/wiki/IBM_350. accessed 20 October 2026.

[30] Floppy disk. Wikipedia, https://en.wikipedia.org/wiki/Floppy_disk. accessed 20 October 2026.

[31] Operating system. Wikipedia, https://en.wikipedia.org/wiki/Operating_system. accessed 20 October 2026.

[32] Kernel (operating system). Wikipedia, https://en.wikipedia.org/wiki/Kernel_(operating_system). accessed 20 October 2026.

[33] Rocky Enterprise Software Foundation. Rocky linux. https://rockylinux.org/. accessed 20 October 2026.

[34] macOS version history. Wikipedia, https://en.wikipedia.org/wiki/MacOS_version_history. accessed 20 October 2026.

[35] Unix shell. Wikipedia, https://en.wikipedia.org/wiki/Unix_shell. accessed 20 October 2026.

[36] Free Software Foundation. Bash reference manual: command search and execution. https://www.gnu.org/software/bash/manual/html_node/Command-Search-and-Execution.html. accessed 20 October 2026.

[37] The Z shell manual. https://zsh.sourceforge.io/Doc/. accessed 20 October 2026.

[38] Free Software Foundation. Bash reference manual. https://www.gnu.org/software/bash/manual/. accessed 20 October 2026.

[39] Free Software Foundation. GNU coreutils manual. https://www.gnu.org/software/coreutils/manual/. accessed 20 October 2026.

[40] Free Software Foundation. Bash reference manual: Bash startup files. https://www.gnu.org/software/bash/manual/html_node/Bash-Startup-Files.html. accessed 20 October 2026.

[41] The Z shell manual: startup files. https://zsh.sourceforge.io/Doc/Release/Files.html. accessed 20 October 2026.

[42] ps(1). Linux man-pages, https://man7.org/linux/man-pages/man1/ps.1.html. accessed 20 October 2026.

[43] environ(7). Linux man-pages, https://man7.org/linux/man-pages/man7/environ.7.html. accessed 20 October 2026.

[44] kill(1). Linux man-pages, https://man7.org/linux/man-pages/man1/kill.1.html. accessed 20 October 2026.

[45] File system. Wikipedia, https://en.wikipedia.org/wiki/File_system. accessed 20 October 2026.

[46] Mount (computing). Wikipedia, https://en.wikipedia.org/wiki/Mount_(computing). accessed 20 October 2026.

[47] The Linux Foundation. Filesystem hierarchy standard, version 3.0. https://refspecs.linuxfoundation.org/FHS_3.0/fhs/index.html, 2015. accessed 20 October 2026.

[48] Symbolic link. Wikipedia, https://en.wikipedia.org/wiki/Symbolic_link. accessed 20 October 2026.

[49] Apple file system. Wikipedia, https://en.wikipedia.org/wiki/Apple_File_System. accessed 20 October 2026.

[50] ext4. Wikipedia, https://en.wikipedia.org/wiki/Ext4. accessed 20 October 2026.

[51] XFS. Wikipedia, https://en.wikipedia.org/wiki/XFS. accessed 20 October 2026.

[52] NTFS. Wikipedia, https://en.wikipedia.org/wiki/NTFS. accessed 20 October 2026.

[53] exFAT. Wikipedia, https://en.wikipedia.org/wiki/ExFAT. accessed 20 October 2026.

[54] Network file system. Wikipedia, https://en.wikipedia.org/wiki/Network_File_System. accessed 20 October 2026.

[55] Lustre (file system). Wikipedia, https://en.wikipedia.org/wiki/Lustre_(file_system). accessed 20 October 2026.

[56] curl.1 the man page. https://curl.se/docs/manpage.html. accessed 20 October 2026.

[57] wc(1). Linux man-pages, https://man7.org/linux/man-pages/man1/wc.1.html. accessed 20 October 2026.

[58] cut(1). Linux man-pages, https://man7.org/linux/man-pages/man1/cut.1.html. accessed 20 October 2026.

[59] grep(1). Linux man-pages, https://man7.org/linux/man-pages/man1/grep.1.html. accessed 20 October 2026.

[60] `sort(1)`. Linux man-pages, https://man7.org/linux/man-pages/man1/sort.1.html. accessed 20 October 2026.

[61] `chmod(1)`. Linux man-pages, https://man7.org/linux/man-pages/man1/chmod.1.html. accessed 20 October 2026.

[62] Free Software Foundation. Bash reference manual: shell scripts. https://www.gnu.org/software/bash/manual/html_node/Shell-Scripts.html. accessed 20 October 2026.

[63] Free Software Foundation. Bash reference manual: shell parameter expansion. https://www.gnu.org/software/bash/manual/html_node/Shell-Parameter-Expansion.html. accessed 20 October 2026.

[64] Free Software Foundation. Bash reference manual: the `set` builtin. https://www.gnu.org/software/bash/manual/html_node/The-Set-Builtin.html. accessed 20 October 2026.

[65] Free Software Foundation. Bash reference manual: redirections. https://www.gnu.org/software/bash/manual/html_node/Redirections.html. accessed 20 October 2026.

[66] Free Software Foundation. Bash reference manual: exit status. https://www.gnu.org/software/bash/manual/html_node/Exit-Status.html. accessed 20 October 2026.

[67] Free Software Foundation. Bash reference manual: looping constructs. https://www.gnu.org/software/bash/manual/html_node/Looping-Constructs.html. accessed 20 October 2026.

[68] Free Software Foundation. Bash reference manual: filename expansion. https://www.gnu.org/software/bash/manual/html_node/Filename-Expansion.html. accessed 20 October 2026.

[69] University of Cambridge Research Computing Services. CSD3 user guide: I/O management. https://docs.hpc.cam.ac.uk/hpc/user-guide/io_management.html. accessed 20 October 2026.

## Index

absolute path, 9
arithmetic expansion, 14

bus, 3

caches, 4
command line, 7
command substitution, 14
core, 3
current directory, 9
cycle, 4

environment variable, 8
exit status, 15

file system, 9
file-system format, 10

hard disk drive, 4
hardware thread, 3

kernel, 6

logical CPU, 3

memory, 3
mount point, 9
mounted, 9

operating system, 6

path, 9
permission mode, 13
pipe, 12
process, 8
process ID, 8
processor, 3

RAM, 4
registers, 4
relative path, 9

shebang, 14
shell, 7
shell script, 14
solid-state drive, 4
standard streams, 14
startup files, 8
system call, 6

terminal, 7