Command Line Handbook

Petr Stribny

Table of Contents

Command Line Handbook

Version 2025-05.

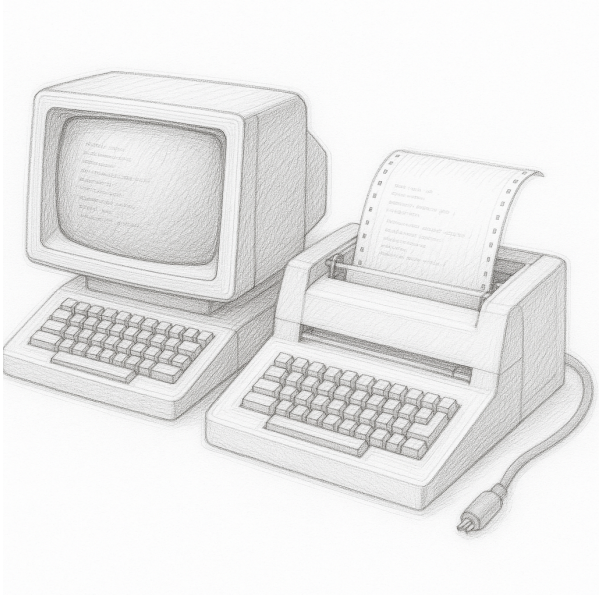
Self-published.

https://commandline.stribny.name

https://stribny.name

Icons are from https://www.flaticon.com.

Introduction



The *command line* is a text-based interface that allows users to interact with computers by typing commands, and it has been used in various forms since the early days of computing. People used to interact with computers using *terminals*—separate devices connected to computers, such as mainframes—in the form of teletype machines equipped with keyboards and paper printers or video terminals with computer screens. The users would type their command input on a keyboard and receive a text result after each executed command.

When people think of command lines today, they most likely think of text interfaces

accessed from graphical environments through a *terminal emulator* (also called a *terminal*), an application that mimics the behavior of older physical terminals. Modern terminal emulators bring enhanced graphical capabilities, but the major difference of modern command lines is the use of sophisticated command languages to issue commands. The command languages have evolved significantly into programming environments that we call *shells*.

In general, shells are user interfaces to interact with operating systems. We often use *desktop environments* in the form of *Graphical User Interface* (GUI), with visual elements like windows, menus, and buttons. Such environments are graphical shells. Text-based shells are mostly not used directly but rather through the terminal emulators that render the text input and output on the screen and run as applications inside the graphical desktop environment.

Command Line Interfaces (CLIs) are still widely used even though the paradigm of running individual commands remains the same as in the past. But invoking single commands and getting single output is not the only way to use the command line. *Terminal User Interfaces* (TUIs), applications that run in terminals and take over the terminal screen, can provide rich interactive experiences. Applications like text editors, file managers, monitoring tools, and others can thus be accessed directly on the command line.

In software development and system administration, it is almost impossible not to use a command line in some capacity. And that's not by accident, as there are many benefits:

- Systems that don't have any graphical environment installed, including remote servers, virtual machines, and containers like Docker, can still be controlled and administered.
- Many programs exist exclusively for the command line, especially developer tools and system utilities.
- Command line programs can be combined together, using the output of one as the input for another.
- Text-based shells can be scripted to automate tasks easily.
- Creating CLI and TUI applications has a low barrier to entry, simplifying development and cross-platform support.

Despite these benefits, the command-line world is vast, complex, and sometimes peculiar due to its history, ongoing development, platform variations, and the sheer volume of

knowledge to acquire.

This book aims to guide you through the modern Linux command line. While not strictly a tutorial, how-to guide, or reference, it blends elements of each to quickly familiarize you with essentials. Consider this handbook a starting point, providing resources and ideas for deeper exploration later on.

Reading or skimming the book from start to finish is recommended. New concepts are introduced progressively, and later sections assume familiarity with previous material.

This handbook is primarily for Linux users. Examples are based on standard keyboards and programs in Linux-based systems, though most concepts should apply similarly across platforms. Windows or macOS users, or those with non-standard keyboards, may encounter some differences.

Shells and Terminals

Shells

There are several text-based shells that we can use, but the most common and useful to know is *Bourne Again SHell*, known as Bash[1]. It is the shell commonly associated with Linux since it is installed by default on most Linux distributions, such as Ubuntu or Fedora. Another popular shell is Z shell[2] (Zsh), now the default shell on macOS. While Bash is an excellent choice for scripting thanks to its ubiquity, Zsh is used by developers and sysadmins for its interactivity and configurability. fish shell[3] is also gaining popularity, being designed to be a user-friendly shell with good defaults. However, it is less compatible with the other mentioned shells. Windows operating system also has its shell called *PowerShell*.

This book focuses on working with Bash and Z shells because of their practicality and popularity among programmers. All examples and programs are relevant to these shells unless explicitly said otherwise. Still, the programs and techniques can also serve as inspiration when using other shells.

Terminals

Modern terminals work as terminal emulators, emulating text-based environments in a graphical interface. They are responsible for taking textual input through a keyboard, passing it to the running shell and programs, and displaying the text output coming from the shell and programs that we run. Even though some terminals support images now, it is best to think of a terminal window as a grid of characters—graphical elements are usually drawn using letters and special font symbols.

Depending on the operating system, we can choose between a number of terminal applications. In theory, we can combine different shells with different terminals and operating systems, but not all programs work with each other in practice. The following table presents some typical situations.

Table 1. Combination of operating systems, terminals, and shells

Operating System	**Shell**	**Alternatives**	**Terminal**	**Alternatives**
Fedora, Ubuntu, … (with Gnome)	Bash	Z shell and other Unix/Linux shells	Gnome's Terminal	Konsole, xterm, Tilix
macOS	Z shell	Bash and other Unix/Linux shells	Terminal	iTerm
Windows	PowerShell	console (legacy) or Bash, Z shell with Windows Subsystem for Linux	Windows Terminal	ConEmu, Console2, PuTTY

Terminals also come embedded in other applications, most typically inside programming editors and IDEs like Visual Studio Code. Besides the advantage of being easily accessible during specific tasks, they might also prioritize features that are helpful in the specific environment.

Nowadays it seems that most people either use the default terminal that comes with their operating system, use embedded terminal in a third-party application, or install one of the most advanced modern terminal emulator, typically one from the list below:

- Alacritty[4]
- Kitty[5]
- WezTerm[6]
- Ghostty[7]

It is not required to use any specialized terminal to learn and use the command line with this book. It is more than okay to use any default application you already have. It is also common to use multiple terminals for different needs.

Essential Terminal Features

Terminal emulators handle many tasks behind the scenes, such as starting the shell of our choice, managing text encoding, and rendering output correctly. Most modern emulators support these basics seamlessly. Let's explore some essential features typically built into terminal emulators today:

- **Scrollback**. Command inputs and outputs often exceed the visible screen space. The *scrollback buffer* stores all previously displayed text, including commands and their output, allowing us to scroll back and review it. This is useful for checking previous commands or examining long outputs. Scrolling can be done with the mouse or keyboard shortcuts.

Some terminals require special key combinations (e.g., `Ctrl` + `Shift` + `Up`) to scroll, as standard keys like `Up`, `Down`, `PageUp` and `PageDown` are passed to the shell by default. Check your terminal's documentation for the correct shortcuts.

Note that the scrolling keys or keyboard shortcuts for scrolling on the terminal might not be the same as the ones used by TUI applications started in the terminal. Applications typically control their own keybindings. Once a TUI application starts, it takes control of the screen. The scrollback buffer becomes visible again after exiting the application.

- **Autoscrolling**. By default, terminals automatically scroll down to show new output as it is generated, ensuring we see the latest updates. This behavior can usually be

paused by manually scrolling up during a long command execution when the text is continuosly appearing on the screen. We can also turn it off in the terminal settings.

- **Scrollback search**. Many terminals allow searching within the scrollback buffer, which is invaluable for investigating long outputs. The experience and shortcuts for this feature vary between terminals.
- **Copy and paste**. Copying and pasting text, such as commands or output, is essential in terminal workflows. Terminal emulators often use different shortcuts than standard applications (e.g., `Ctrl` + `Shift` + `C` instead of `Ctrl` + `C`). Check your terminal's settings to learn or customize these shortcuts.
- **Mouse support.** The mouse is often used for selecting and copying text or scrolling through the scrollback buffer. Some command-line programs, such as text editors and file managers, also support mouse interactions for navigation or triggering actions, together with scrolling.

Check if your terminal applications support mouse interactions. If not, consider alternatives that do, as it is not always worth it to learn keyboard shortcuts for everything.

- **24-bit colors.** The terminal experience nowadays is full of colors. With the exception of scripting, it is a good idea to opt for programs that utilize colors for better user experience on the terminal, as most emulators support it.

Advanced Terminal Features

There is a lot of development in features and experiences in new terminal emulators. Depending on your workflow, you might or might not need these features. I collected some important aspects to consider when choosing a terminal emulator below:

- **Cross-platform compatibility**: While not essential for everyone, using a cross-platform terminal emulator can be advantageous. It ensures a consistent experience and simplifies sharing settings across different operating systems.
- **Font ligature support**: Not all emulators support font ligatures, which can enhance readability, especially for programming tasks. If you rely on ligatures, consider emulators that support them.
- **Image rendering**: Some terminals support displaying images. While not a common requirement, this feature can be useful when browsing files in terminal-based file managers.

- **GPU acceleration**: Modern terminal emulators with GPU support offer faster rendering, which is beneficial for handling large volumes of output or fast image previews.
- **Tabs and panes**: Terminals with built-in tabs or split-screen panes allow running multiple shells or applications simultaneously. This feature is particularly useful if you prefer managing sessions within a single window, though it may be less critical with a good window manager.
- **Clickable links**: Many terminal emulators auto-detect URLs and allow opening them conveniently (e.g., via `Ctrl` + mouse click or a context menu). Some specialized terminals, like the one in Visual Studio Code, can also open file paths, making debugging stack traces much faster.
- **Customization**: Preferences vary—some users prefer minimal configuration, while others seek extensive customizability. Terminals that allow configuration through simple text files are ideal, as they are easy to back up and reuse.
- **AI assistance**: While not yet a standard feature, AI assistance is emerging in some terminals. However, AI tools can also be integrated externally. Later in the book, we'll explore how to leverage AI on the command line using specialized applications and shell configuration.

[1] https://www.gnu.org/software/bash/

[2] http://zsh.sourceforge.net/

[3] https://fishshell.com/

[4] https://alacritty.org/

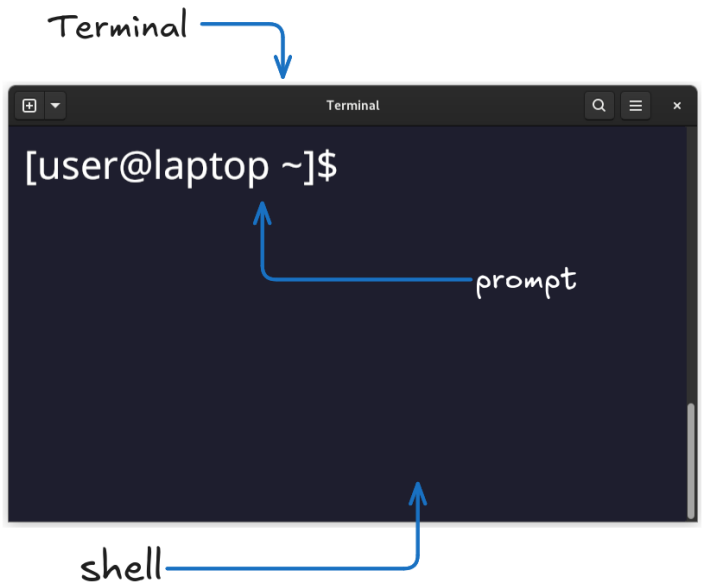
[5] https://sw.kovidgoyal.net/kitty/

[6] https://wezterm.org/

[7] https://ghostty.org/

Command Line Basics

Prompt



Opening a terminal will start a new shell session. We are then given access to a command line to execute commands. The beginning of the command line before the cursor is known as the *command prompt* (the `[user@laptop ~]$` in the picture). Command prompts are configurable and usually slightly different on different systems. Typically, the following is displayed:

- The user name of the user running the shell. The user affects all executed commands, as they will be run within this user's permissions.
- The name or an IP address of the computer where commands will be executed.
- The currently selected directory (the `~` in the picture after the computer name). We will learn soon that a tilde refers to the user's home directory.
- The account type. By convention, the dollar sign (`$`) denotes a regular user, whereas the hash (`#`) is used for system administrators. In documentation and various tutorials, these symbols might be used to distinguish whether something should be run under administrator privileges.

In this book, a prompt is marked with the greater than symbol (`>`) in all cases. If you are going to copy-paste something from the interactive examples, don't copy this symbol. Lines that don't begin with this symbol are outputs.

Running Commands

Running commands is as simple as typing them and hitting `Enter`. However, it is important to know where a command is executed in the file system. Typically, a new shell session will start at our home directory like `/home/username` where *username* will be the name of our user on the system. Many shells display the current file system location inside the prompt. Either way, we can simply see where we are by running the command `pwd` (print working directory):

```
> pwd
/home/username
```

A command like this seems straightforward, but it is not. The problem is that a command can run a script, an executable binary, a custom shell function, an alias, or a *shell built-in* (built-ins are utilities implemented directly in the shell itself). We will see all of those things in action later. For now, let's find out what has been run.

The easiest way to do that is with the command `type -a pwd`. It will print all executable things named `pwd`.

```
> type -a pwd
pwd is a shell builtin ①
pwd is /usr/bin/pwd ②
```

① `pwd` is a shell built-in. Because it is a shell built-in, it will take precedence over external programs. Notice that the output mentions it first.

② `pwd` is also a separate executable program in `/usr/bin/pwd`.

It turns out that `pwd` is two separate things on the system. Even though they both serve the same purpose, they are, in fact, different. If the built-in command weren't part of the shell, the program at `/usr/bin/pwd` would get executed instead. But how does the shell know where to look for programs? Can we expect the shell to find all programs on the file system?

Shells will search for executable programs in a list of locations called the *PATH*. On Linux, the PATH looks something like this:

```
/home/username/.local/bin:/usr/share/Modules/bin:/usr/local/bin:/usr/local/sb
in:/usr/bin:/usr/sbin:/home/username/bin:/var/lib/snapd/snap/bin
```

We can see that the PATH is just a list of file system paths separated by semicolons. Any program in one of these locations can be executed without specifying its location. Because `/usr/bin` is mentioned here, the shell can find `/usr/bin/pwd` just based on the file name. If multiple programs have the same name, the order of the paths will be taken into account.

What can we do if a program is not on the PATH, or when we need to run a program overshadowed by another program or built-in of the same name? We need to run it by typing the absolute or relative file path.

```
> /usr/bin/pwd ①
/home/username ②
```

① We run the exact program by specifying the full path to the executable.

② In this case, the output of the program is the same as it was from the built-in.

The PATH is stored as a *process environment variable*, exposed in a shell as a *shell variable*.

Variables and Environment Variables

A shell variable has a case-sensitive name made out of a sequence of alphanumeric characters and underscores. It stores a value, usually a string or a number, and can have several attributes. For instance, a variable can be marked as read-only or having a particular type, although working with these attributes is often unnecessary.

Variables can be defined or reassigned using the equal sign (`=`) and used with the help of *variable substitution*. When a variable is prefixed by the dollar sign (`$`) in a command or inside a double-quoted string, it will be replaced by its value. To see it in action, let's introduce another built-in shell command, `echo`. It simply prints anything passed to it.

```
> my_var=10 ①
> echo $my_var ②
10
> echo "My variable value is $my_var" ③
My variable value is 10
```

① Variable assignment. Notice that there is no space around the equal sign (`=`).

② The shell will substitute `$my_var` with the value of `my_var` and pass this value to `echo`. `echo` prints it.

③ Variable substitution used inside a double-quoted string.

The full syntax for variable substitution uses curly braces. So the example would look like `echo ${my_var}`. This full syntax is useful to know if we ever need to place other characters directly around the variable.

Environment Variables

Process environment variables (also *env vars*) are not the same as shell variables. They are configuration variables that every process has in Unix-like operating systems. As a shell session is also a process, it receives some env vars from the system when it is run. The shell imports them at startup time as regular variables but marks them as *exported*.

Every variable marked as exported will be copied into the process environment of all child processes started in the shell. This behavior makes it possible to pass configuration and secrets to all programs we run.

To see all variables marked as exported (which will eventually become environment variables of the child processes), use the command `export`.

```
> export
PATH=/home/username/.local/bin:/usr/share/Modules/bin:/usr/local/bin:/us
r/local/sbin:/usr/bin:/usr/sbin:/home/username/bin:/var/lib/snapd/snap/b
in ①
...
```

① All exported variables are printed. The shell set them as environment variables automatically, or we exported them ourselves before. The output is shortened, only showing that the PATH variable is included.

Useful environment variables set by the system include `PATH`, `USERNAME`, and `HOME`. Since the shell imported them all as variables when it started, we can print their values the same way we did before.

```
> echo $PATH
/home/username/.local/bin:/usr/share/Modules/bin:/usr/local/bin:/usr/local/sb
in:/usr/bin:/usr/sbin:/home/username/bin:/var/lib/snapd/snap/bin
> echo $USERNAME
username
```

To promote a regular variable into an environment variable, we need to export it. A variable can be exported and assigned at the same time:

```
> ENV_VAR='Just a regular variable now' ①
> export ENV_VAR ②
> export ENV_VAR_2='Already an environment variable' ③
```

① Standard variable assignment as before. Using single quotes to denote an exact string is safer when variable substitution is not needed.

② Export of an existing variable. `ENV_VAR` will now be passed as an environment variable to programs started in the same shell session.

③ Export of a completely new variable.

Now we can change the PATH. It will make it possible to execute programs on a command line just by their name from any location. For instance, we can create a new `bin` folder inside a home directory to store programs and scripts and include it on the PATH for convenience:

```
> export PATH="$HOME/bin:$PATH" ①
```

① We set `PATH` to a new value, using variable substitution to use both the path to our home directory and the previous value of `PATH`. We export it to make it available to child processes as well.

After changing the `PATH`, we can place any executables inside the `/home/username/bin` directory and execute them by typing just their name. However, this change is not permanent and will be gone when the current shell session is closed. The *Configuration* section later in the book demonstrates how to make it permanent.

Absolute and Relative Paths

File paths that start with a forward slash `/` are absolute, fully determining the location of a file or a folder. We can also use relative paths relative to the location of the working directory. Such paths start with one or two dots, pointing to the current or parent directory. The *tilde expansion* provides quick access to the home directories when the path starts with `~`. See the table below:

Table 2. Referring to files and folders

The path starts with	Points to
`/`	The root directory of the file system
`.`	The current directory
`..`	The parent directory
`~`	The home directory of the current user
`~username`	The home directory of the user `username`

Changing the current working directory can be done with the built-in command `cd` (change directory). Now consider a program named "myprogram" in the home folder `/home/username` and assume that it will output a simple hello message. We can execute it from various locations like this:

```
> # Assuming the folders /home/username/folder/folder2 exists ①
> cd ~/username ②
> pwd ③
/home/username
> ./myprogram ④
Hello!
> cd /home
> ./username/myprogram
Hello!
> cd /home/username/folder
> ../myprogram
Hello!
> cd /home/username/folder/folder2
> ../../myprogram ⑤
Hello!
```

① Every line that starts with `#` is treated as a comment in Bash and Zsh. In other words, it doesn't do anything.

② We use `cd` with a tilde expansion to move to the `username` home directory.

③ We can check that the working directory was changed with `pwd`.

④ It is a good practice to run programs in the current directory with `./` prefix to avoid name collision with other programs on the PATH.

⑤ We can traverse to parent directories by repeating `../`.

The shell also stores the current and previous working directories in variables called `PWD` and `OLDPWD`. Running `cd -` can switch to the previous working folder directly. Running it multiple times will keep switching between the last two working directories.

Arguments

We have already been using arguments when running previous commands. Technically speaking, all tokens separated by a space on the command line are *command line arguments*, the first being a utility name or a script we want to run. Once a shell parses a command, it will know what to execute based on this first argument. The program, function, or built-in being called will use the rest of the arguments as parameters for its execution.

Because spaces separate arguments, all spaces within an argument need to be escaped by a backslash. Alternatively, we can quote them with single or double quotes, as we saw already with strings. Single quotes are used for exact strings, while double quotes allow for various substitutions.

Arguments that are passed one after another are called *positional arguments*. Programs and scripts can access such arguments based on the numerical position or iterate over all of them.

```
> echo Hello World ①
Hello World ②
> cd directory\ with\ spaces ③
> cd 'directory with spaces' ④
```

① The first argument is `echo`, determining what will be called. The following "Hello

World" is, in fact, two arguments separated by space. This means that the built-in command `echo` is passed two positional arguments, "Hello" and "World."

② `echo` prints all its positional arguments one after another.

③ The first argument is `cd`. The shell will call this built-in with one positional argument because the spaces are escaped. Assuming the directory exists, the working directory will be changed to "directory with spaces."

④ It is usually preferable to quote such arguments for readability.

Besides positional arguments, there are also two types of named arguments. They are referred to as *options*, meaning that they are usually optional and change how the program behaves. Short options start with a hyphen (`-`), followed by a single character. The long ones begin with a double hyphen (`--`), followed by the option name.

For demonstration, let's look at a program called `ls` (list directory). `ls` prints the contents of directories that are passed as positional arguments. Without any arguments, it prints the contents of the current working directory. With `-a` and `-l` options, we can change the output:

```
> ls ①
dir1 file1.txt file2.pdf
> ls -a ②
. .. .gitignore dir1 file1.txt file2.pdf ③
> ls --all ④
. .. .gitignore dir1 file1.txt file2.pdf
> ls -l ⑤
total 12
drwxr-xr-x.  3 groupname username 4096 Dec 12 15:05 .
drwxrwxr-x. 15 groupname username 4096 Dec 10 16:56 ..
drwxr-xr-x.  2 groupname username 4096 Dec 12 15:05 dir1
-rw-r--r--.  1 groupname username    0 Dec 12 15:05 file1.txt
-rw-r--r--.  1 groupname username    0 Dec 12 15:05 file2.pdf
```

① `ls` prints what is inside the current working directory. By default, hidden files are not shown (on Unix and Linux-based systems, that means all files beginning with a dot `.` are omitted).

② A short option `-a` will tell `ls` to display all files.

③ Besides the `.gitignore` file, we can also see the references to the current and parent directories (`.` and `..`). It is because the references are implemented using *symlinks*,

which are also files.

④ We use a long-form option called `--all` to tell `ls` to display all files. It is just a variation of the short option `-a`. It is often the case that many named arguments have both a short and a long variant.

⑤ The output when using `-l` option displays additional information about the files and directories like the type, permissions, size, modified date, and so on. We will look at this output again in more detail in the section *Accounts and Permissions*.

Positional and named arguments can be combined in one command (e.g., `ls -l --all`). Short arguments can sometimes be written together (`ls -la` vs. `ls -l -a`). Options are typically written before positional arguments (`ls -l ~/`).

You may encounter the use of a double hyphen `--` as a separator to indicate that all subsequent arguments should be treated as a single argument rather than being processed individually. This is often used to pass an entire command or string as a single argument. For example, in `someprogram arg1 arg2 -- ls -la`, the `--` tells `someprogram` to treat `ls -la` as a single argument rather than parsing it further.

Options can be used as flags or take an *option argument* (again, separated by a space). Long options can also take a value as `--argument-name=value`. Although programs that follow the POSIX[1] standard share some common characteristics, there is no standard in handling program arguments. So it is always best to consult the documentation of each program.

Multiline Commands

It is often helpful to write commands on multiple lines in scripts and on the command line to make it more readable. To do so, use a backslash `\` at the end of each line.

```
> ls \
> -l \
> -a ①
```

① The end of the command.

Command History

Both Bash and Z shells can preserve command history (all commands that we issued on the command line). By pressing `Up` and `Down` keys, we can rotate between previously entered commands. It is especially useful if we want to run a command that we had run recently.

While Bash stores the command history in a file by default, in Zsh, it has to be enabled by setting the appropriate variable (`HISTFILE=~/.zsh_history`), otherwise the history won't be preserved. There are other relevant settings for history as well, such as the number of entries to keep. We will get back to it in the section *Improved History*.

Pressing `Ctrl` + `R` will open *interactive search* of the command history. Once open, type a portion of the command to be retrieved. Pressing `Ctrl` + `R` repeatedly will navigate between matches.

Table 3. Navigating history using shortcuts

Shortcut	Operation
`Ctrl` + `R`	Start/continue interactive search
`Ctrl` + `G`	Abort interactive search
`Ctrl` + `N` or `Down`	Next item from the history
`Ctrl` + `P` or `Up`	Previous item from the history

The last executed command can be reused with two exclamation marks (`!!`) that will automatically expand it. Another option is to prepend a name of the command with one exclamation mark (`!`), which will execute the same program with the same arguments as the last time. For instance, to execute `ls -al` again, use `!ls`.

There is also a trick to insert the last argument of previous commands. This trick is useful since many programs have a file path as their last argument. Pressing `Esc` + `.` or `Alt` + `.` will rotate the last arguments of previous commands:

```
> ls path/to/directory ①
> cd ②
> cd path/to/directory ③
> cd !$ ④
```

① `ls` utility is used with a folder path as an argument.

② After seeing the contents of the directory, we decide this is the right directory to move to and type `cd` followed by a space.

③ Pressing `Alt` + `.` will autocomplete the path.

④ `cd !$` will do the same.

History Command

A command named `history` lists things we have executed in the past.

```
> history
1968  ls
1969  ls -al
```

We can then execute any previous command again using an exclamation mark and the printed number. So to run `ls -al` again, all we have to do is to type `!1969`, followed by `Tab` or `Enter`.

Commands inside comments (starting with `#`) are stored in the history as well. This behavior can be leveraged to store complete or incomplete commands for later use.

The complete history is typically configured to be stored in the history files `~/.bash_history` for Bash and `~/.zsh_history` for Z shell or other location of our choosing. This behavior might be problematic when issuing sensitive commands, but fortunately, there are ways to avoid storing current commands in history. In Bash, putting one space character before the command is enough, and the same behavior can be enabled in Zsh with `setopt HIST_IGNORE_SPACE`.

It is possible to quickly edit the last run command with a shell built-in `fc` (fix command). It will take you to a default text editor configured in the shell with the last command as the text. Edit it and save it in order to execute it again. While pressing `Up` is quicker to access the last command, `fc` allows editing the command in a more powerful text editor and has other features you can explore.

Another way to correct the last command is to use a program called thefuck[2]. It has

various rules for some commonly used programs and usages and can correct the command automatically. Just type `fuck` after the accident.

Using thefuck to correct misspelled commands

```
> cdd ~/ ①
bash: lss: command not found...
> fuck
cd ~/ [enter/up/down/ctrl+c] ②
```

① The intention is to run `cd ~/`, but it was misspelled to `cdd`.

② After running `fuck`, we are offered the correct command and can confirm it by pressing `Enter`.

Autocompletions

Both shells also offer command auto-completion by pressing `Tab` when typing commands. Both can suggest command names and files where files are expected or auto-complete argument names. There are ways to improve the autocomplete experience in both shells—we will discuss some options later on. For now, it is enough to know that the autocompletion of commands and paths exists and use it to shorten the time to write commands.

Programs mentioned with a link will need to be installed first. The book won't distinguish between built-in commands, standard utilities, and external programs from now on.

Shell Expansions

Besides the tilde expansion that we have already seen, there are some other interesting expansions available that can save us a lot of time.

When referring to files and folders, we can use the *file name expansion* which replaces a wildcard pattern with all matching file paths (also called *globbing*). The basic wildcards are `?` for exactly one character and `*` for zero or multiple characters. Let's imagine that there are two text files, one containing `Contents of A` and the other containing `Contents of B`. We can then print their contents with a program called `cat` that accepts multiple files as its arguments or use the file name expansion to do the same.

```
> cat a.txt b.txt ①
Contents of A
Contents of B
> cat *.txt ②
Contents of A
Contents of B
```

① Similar to `echo`, `cat` performs its operation with all positional arguments. In this case, the contents of both files will be printed to the console.

② Using a globbing pattern instead, it is enough to provide just one argument containing the pattern.

Besides wildcard characters, file globbing supports multiple different matching mechanisms, similar to regular expressions:

- Matching specific characters from options written inside brackets, e.g., `[ab]` (a or b)
- Matching characters from a sequence written inside brackets, e.g., `[0-9]` to match a number character or `[a-d]` to match a, b, c, or d
- Caret (`^`) character to negate the selection
- Dollar sign (`$`) to denote the end of the name

- Choosing between several patterns using `(PATTERN|PATTERN2)`

All of them can be combined to arrive at more complex patterns. See the table below for examples.

Table 4. Globbing examples

Pattern	Explanation
`backup-*.gzip`	Match all files starting with `backup-` that have `.gzip` file extension.
`[ab]*`	Match all files that start with either `a` or `b`.
`[^ab]*`	Match all files that don't start with `a` or `b`.
`*.???`	Match all files whose file extension has exactly three characters.
`(*.???|*.??)`	Match all files whose file extension has exactly two or three characters.

The asterisk character (`*`) will by default not match any hidden files starting with a dot. To prevent Bash or other shells from performing globbing, it is needed to either escape the characters with a backslash (`\*`) or quote them.

The *brace expansion* allows composing names from multiple variants by placing them in curly braces like `{variant,variant2}`.

Creating multiple files with different names, but the same extension

```
> touch {file1,file2,file3}.txt ①
> ls
file1.txt file2.txt file3.txt ②
```

① The original purpose of the program `touch` was to update modified timestamps on files, but it can be also used to create empty files.

② Three files were created, sharing the same file extension.

Aliases

We can create an alias when we don't want to type or remember complex commands. An alias can point to anything that would be typed on a command line.

```
> alias project1="cd /home/username/project1" ①
> project1 ②
> pwd
/home/username/project1 ③
```

① An alias called `project1` is set to change the working directory to a project folder.

② The alias can be invoked as any other program just by typing its name.

③ The command was executed through the alias.

An alias will take precedence over external programs and even built-in commands. That means we can also overshadow standard utilities like `cd` or `ls` and replace them with something else.

Standard Input, Standard Output, Standard Error

Standard input, output, and error streams, also referred to as `stdin`, `stdout`, and `stderr`, are created when a command is run. These are text data streams that applications can use to pass data to each other. Many standard utilities and command-line applications work with input or output streams. The error stream is a separate stream where a program can output error messages that can also be redirected to another program or a file. The streams behave like files and are identified by file descriptors, 0 being the input stream, 1 being the output stream, and 2 being the error stream. The streams don't have any specific format and can contain any text.

We have already seen the standard output stream in action. When commands such as `echo` or `cat` are executed, their standard output is displayed in the shell. It is the default behavior for standard output when a program is run from a command line. We can redirect this output stream using `>` and `>>` operators. A classic example is saving the output of the stream to a file:

```
> echo "Hello!" > file1.txt ①
> cat file1.txt
Hello! ②
> echo "Hello!" >> file1.txt ③
> cat file1.txt
Hello!
Hello!
```

① We will not see any output from the `echo` command in the terminal, because the output got redirected to a file.

② We can check that there is now a file `file1.txt` with the contents "Hello!".

③ With `>>` operator, we can append text to a file instead of overwriting it.

```
> cat header.csv spreadsheet.csv > combined.csv ①
```

① Multiple files can be merged into one. `cat` will read and output all files passed to it, while the operator will redirect the whole output to a new file.

We can redirect the standard error stream similarly, identifying the stream by its file descriptor.

```
> cat file_which_does_not_exist.txt
cat: file_which_does_not_exist.txt: No such file or directory ①
> cat file_which_does_not_exist.txt 2> error.txt ②
> cat error.txt
cat: file_which_does_not_exist.txt: No such file or directory
```

① The standard error stream is also visible in the shell by default.

② We can prepend the `>` or `>>` operators with the descriptor identifying the stream, in this case, 2 for `stderr`, to redirect the stream to a file. Again, no output will be visible in the shell now.

The error stream will be printed to the terminal when we redirect just the standard output. If we redirect just the error stream, the standard output will be printed in the terminal. Note that a program can use both streams at once.

It is also possible to redirect both the standard output and the error stream at once by chaining the operations together or by using the ampersand (`&`) instead of the stream number. Let's assume that we want to run a script called `streams.sh` that writes into both streams:

```
> ./streams.sh 1> output.txt 2> error.txt ①
> ./streams.sh &> output_and_error.txt ②
> ./streams.sh &> /dev/null ③
```

① Both standard streams are captured in separate files by placing the redirects one after another.

② Alternatively, we can capture the streams into the same file. `&>` will overwrite the file while `&>>` will append to it.

③ We can discard all output of a program by sending the outputs to `/dev/null`, which is something like a black hole. In this case, the script or program will run without its output being outputted to the screen nor saved in any file.

A pipe operator (`|`) is used to pass the standard output stream to another program as the input stream. Piping programs together is the most common way to compose a command that needs different programs and functions to work.

```
> echo "Hello, NAME" | sed "s/NAME/Petr/"
Hello, Petr
```

In this example, the standard output of `echo` is fed into `sed`, a utility for transforming text. Sed has a "replace" operation that expects a regular expression to match a text and replace it with something else, in our case matching `NAME` and replacing it with `Petr`. We will go into more detail later in the section *Text Data Wrangling*. For now, it is enough to understand how text data flows from one program to another.

To pipe all streams at once, including the `stderr`, we can replace `|` with `|&`. This change will put both streams on the standard input for the consumption of the following program.

The utility `tee` can split the output of a program. It redirects it to multiple files and/or to the screen. It is handy when we want to save the output to a file but see it in the terminal

simultaneously. Building on our previous example, we will now use `echo` together with a pipe and `tee` to get the output both to the screen and to multiple files at once:

```
> echo "Hello!" | tee file1.txt file2.txt
Hello!
> cat file1.txt
Hello!
> cat file2.txt
Hello!
```

Tee can also append the output to a file instead of overwriting it with `-a` option. This is useful when we need to check a program's output multiple times and keep all the history, e.g., when using a profiler.

Finally, the opposite operators `<` and `<<` are used to redirect the standard input, in other words, to send some text data to a program. However, using these operators is not that common because it is possible to use the command line arguments or pipes to achieve the same thing in most situations.

Here documents

Here documents are used to provide a multiline input to a program. A here document is created using the standard input `<<` operator, followed by a delimiter that will be used to end the document (text) later.

```
> my_var="value"
> cat << END ①
. first line ②
. second line with ${my_var}
. END ③
first line ④
second line with value ⑤
```

① A multiline string is provided to `cat` with a delimiter called `END`. It doesn't matter how we name the delimiter here. We can also omit the space in front of it.

② Once the command is entered, we are given a different prompt that allows us to write text on multiple lines (using `Enter`).

③ After the delimiter is encountered, the entered multiline string will be passed to `cat`.

④ `cat` prints the string.

⑤ The text will roughly behave like entering text using double quotes. For example, our variable `my_var` will be replaced.

The ending delimiter doesn't necessarily end the command. For instance, we can continue the command and store the result in a file with `cat << END > file.txt` and proceed like before.

Command and Process Substitutions

Substituting a command with its result is done with `$(CMD)`. This syntax makes it possible to use the command's standard output in a string or as an argument for another command. If we need a file path as the argument, we can use the *process substitution* by writing `<(CMD)`. It will run the command, place its standard output in a temporary file and return the location to the file.

```
> echo "You are now in the directory $(pwd)"
You are now in the directory /output/of/pwd/command
> diff <(ls /location/to/firstfolder) <(ls /location/to/secondfolder) ①
... ②
> echo <(ls /location/to/firstfolder)
/proc/self/fd/12 ③
```

① The `diff` utility can compare files and print their differences. If we pass it the result of `ls` commands, we can compare the contents of directories.

② The output is omitted for brevity.

③ We can use `echo` to demonstrate that the process substitution returns a path to the temporary file.

Running Multiple Commands

It is possible to run multiple commands consecutively and conditionally with `&&`, `||` and `;` operators. A semicolon is just a *command delimiter*, allowing us to place multiple commands on the same line. `&&` and `||` are *short-circuiting operators* that we know from other languages. They will only evaluate the following command if the preceding one

will exit with a true or a false value.

```
> true && echo "This will be executed"
This will be executed
> false || echo "This will be executed"
This will be executed
> false ; echo "This will be executed"
This will be executed
```

The evaluation of whether the run of a command is true or false is based on *exit codes*. We will see how it works in the section *Programming in Bash*.

Using xargs

A program called `xargs` can be used to execute a program multiple times. It reads the standard input and converts it into arguments for another program.

```
> ls
a.png  a.txt  b.png  b.txt
> find . -name "*.png"
./a.png
./b.png
> find . -name "*.png" | xargs rm -rf
> ls
a.txt  b.txt
```

`find` is a standard utility that can find files. The location where to look for them is passed as a positional argument, while the searching pattern is specified using the `-name` option. Consult the section *Search* for more information about finding files.

`rm -rf` is used to delete files on a disk recursively. The typical usage would be to call it with arguments (`rm -rf a.png b.png`), but in this example, it is invoked through `xargs`. `xargs` converts the input stream from `find` into arguments for `rm`.

Shell Shortcuts

Various shell shortcuts make it possible to operate quickly on a command line when entering commands. The table lists basic shortcuts that work in both Bash and Zsh.

Table 5. Common shell shortcuts

Operation	Shortcut
Rotate previously entered commands	`Up` and `Down`
Go to the beginning of the line	`Ctrl` + `A`
Go to the end of the line	`Ctrl` + `E`
Go back one word	`Alt` + `B`
Go forward one word	`Alt` + `F`
Delete the previous word	`Alt` + `Backspace`
Delete the next word	`Alt` + `D`
Cut from the cursor to the end of the line	`Ctrl` + `K`
Cut from the beginning of the line to cursor	`Ctrl` + `U` (Bash only, Zsh will cut the whole line)
Paste the deleted text back	`Ctrl` + `Y`
Open history search	`Ctrl` + `R`
Clear the screen	`Ctrl` + `L` (or type `clear`)
Edit the current command line in the default text editor	`Ctrl` + `X`, `Ctrl` + `E` (Bash only)
Incrementally undo previous changes on the command line	`Ctrl` + `/`
Cancel the command execution with `SIGINT` signal	`Ctrl` + `C`
Cancel the command execution with `SIGQUIT` signal	`Ctrl` + `\`
Pause the command execution with `SIGSTOP` signal	`Ctrl` + `Z`
Close the shell session	`Ctrl` + `D` (or type `exit`)

Learning these shell shortcuts is incredibly useful. I recommend practicing them often to explore the power behind them. We can use them in many real-life situations:

- *Incrementally undo previous changes on the command line* to quickly correct mistakes when writing commands.
- *Rotate previously entered commands* and *Open history search* to access previous commands without typing them again.
- *Edit the current command line in the default text editor* to edit multiline, complicated, or copy-pasted commands much easier because they can be edited in a mature text editor with syntax highlighting and other features.
- *Cancel the command execution* to quickly exit programs when they are not responding.

We will see how to customize them in the *Configuration* section so that we can alter the behavior of the current shell, for example, to get Bash-only behavior in Zsh (and vice versa).

The `SIGINT`, `SIGQUIT` and `SIGSTOP` signals will be explained later in the section *Job Control*.

Terminal emulators or multiplexers will also have their own set of valuable shortcuts worth learning for tasks like switching between terminal sessions.

Getting Help

The usual way to ask programs for help is to use `-h` or `--help` options. Many programs will understand this convention and print their documentation. As an example, look at the output of `cat --help`. The content that will be printed is entirely on the program being run, although typically a short “one page” documentation will be displayed. Some modern applications also use colors to make it nicer.

Another way is to use Linux manual pages, often abbreviated to “man pages”. Man pages are a form of software documentation that provides information about built-in commands and installed programs, among other things. It is a collection of text files that follow a standardized structure so all manual pages look similar to each other, which

makes them easy to digest and navigate.

Neither man pages nor help options are guaranteed to be available. Manual pages are typically available for system utilities (things that come with your operating system), while help options are meant to be used as convention with any executable programs or scripts.

The `man` program displays manual pages right in the terminal. Man uses the program called `less` designed for page-by-page browsing under the hood because man pages can be long. To test it, open the manual page of the man command itself with `man man`, or provide another program name as the argument. The documentation will be opened in `less` by default. See the table for some tips on how to navigate the docs comfortably.

Table 6. Navigating in less, e.g., when browsing manual pages

Operation	Command/shortcut
Quit	`q`
Display help for navigating in less	`h`
Navigate up and down	`Up` and `Down`
Navigate pages up and down	`PageUp` and `PageDown`
Go to the beginning	`g`
Go to the end	`G`
Find a text located after the cursor's position	`/` + type pattern
Find a text located before the cursor's position	`?` + type pattern
When multiple patterns are found, move to the next one	`n`
When multiple patterns are found, move to the previous one	`N`

We can use `less` to browse any large text file and use the same interface and shortcuts as we do when viewing manual pages. Just execute `less` explicitly with the text file as the argument (`less file.txt`).

Manual pages can also be browsed online at man7.org[3] or many other websites that provide man pages as well-formatted HTML documents. It is worth looking for alternatives, as different websites source man pages from different sets of programs, and might offer additional features. For instance, ManKier[4] offers built-in search and also shows additional usage examples.

Setting up a ManKier custom search in your web browser with the keyword "man" will allow you to access manual pages in the address bar the same way as in the terminal (by typing `man <program>`).

Explain Shell[5] is an online application that will explain any combination of standard commands that we come across and don't understand. The advantage is that we don't have to look up documentation for individual commands separately. It is enough to copy and paste the entire command and Explain Shell will explain it.

Cheatsheets

"Cheatsheet" utilities provide quick help on the common usage patterns. TLDR.sh[6] is one of them. It can be installed as a command-line application or used online.

Using TLDR

```
> tldr echo
  echo

  Print given arguments.
  More information: https://www.gnu.org/software/coreutils/echo.

  - Print a text message. Note: quotes are optional:
    echo "Hello World"

...
```

navi[7] is a tool for creating custom cheatsheets and executing them interactively. It makes it easy to store help texts in files and access them on the command line.

Community-sourced command patterns from the TLDR database can be accessed with `navi --tldr`.

Using AI

It can be time-consuming or difficult to figure out complex commands. The modern Large Language Models (LLMs)—AI of the day—offer the easiest, albeit faulty, tool to generate commands and interact with computers. The most straightforward way to get AI on the command line is to use a tool such as llm-cmd[8], a plugin for the popular utility `llm`.

Using `llm` to generate commands

```
> llm cmd 'list directory contents' ①
> ls -a ②
```

① The tool is invoked with `llm cmd`, followed by the instructions. Use quotes to pass in all instructions as one string.

② The tool generated the response for us using a preconfigured model. We can examine the command and hit `Enter` if we deem it correct.

You will need to install and configure `llm` to use the model of your choice. You might need to provide API keys or download a model locally.

The `llm` utility can be set up to use a local large language model. In this way, the use of such a tool won't leak any secrets or confidential information and will also provide access without an internet connection.

Use with caution. If you are unsure, consult Explain Shell and run commands under a user account that can't do too much damage.

We can also use LLMs to explain the command:

Using `llm` to explain commands

```
> llm 'explain the command ls -a'
The `ls -a` command is a Unix-based command used to list all files and
directories in the current directory. Here's a breakdown of what each part of
the command does:

* `ls`: This stands for "list" and is the command itself, which lists the
contents of a file system.
* `-a`: This flag stands for "all" or "hidden". When used with `ls`, it tells
the command to list all files and directories, including hidden ones (those
that start with a dot, like `.hiddenfile`).
...
```

[1] https://en.wikipedia.org/wiki/POSIX

[2] https://github.com/nvbn/thefuck

[3] https://man7.org/linux/man-pages/index.html

[4] https://www.mankier.com

[5] https://explainshell.com

[6] https://tldr.sh/

[7] https://github.com/denisidoro/navi

[8] https://github.com/simonw/llm-cmd

Configuration

Configuring Terminals

The first order of business is to configure the terminal emulator. Typically we can configure the following things:

- Scrollback behavior like scroll on output, scroll on input, and scrollback buffer size
- Cursor shape and blinking behavior
- Font family, font size, and cell spacing between characters
- Color theme
- Shortcuts/keybindings

It makes sense to increase the scrollback buffer size if it is too small, pick a good font and comfortable font size, check and reconfigure shortcuts, and choose a good color theme. It might happen that colors in the terminal will not look good by default, so configuring a color theme is not only good for aesthetics but also basic accessibility.

Make sure that you get a good font from Nerd fonts[1] that contains additional glyphs (icons). The extra glyphs can be leveraged in prompts, status bars, and other places instead of text to communicate ideas elegantly.

Configuring Shells

Changing Shell

We can switch between several installed shells simply by typing their name. So when we are in Bash, we can type `zsh` to enter Z shell and type `bash` to enter Bash. This approach starts a new process every time, but it is the quickest way to do it if necessary.

To make a choice of shell permanent, we need to use a program called `chsh` (change shell).

Change default shell with chsh

```
> chsh -s $(which zsh) ①
```

① A *command substitution* is used to replace `which zsh` with its result. `which` finds the path to Zsh. If Zsh is installed, it will be set as the default shell.

Startup Scripts

All shells have *startup files* or *startup scripts* that are executed as soon as a new shell session starts. Because a shell can start in an interactive or a non-interactive mode (e.g., when running a script from a file), there are different startup files that we might want to use.

This section only covers the interactive mode, which is used when the shell runs from a terminal. Bash and Zsh shells store their startup files in the user's home directory. Bash in `~/.bashrc` and Zsh in `~/.zshrc`. The script itself is just a file with commands written on new lines, so we can reuse everything we have learned so far.

In startup scripts, we usually want to:

- Set *shell options* and other settings that control the behavior of the shell itself
- Export environment variables
- Configure the prompt
- Set up custom aliases and functions
- Define keybindings
- Load shell plugins

The following example demonstrates what a startup script can look like. Don't worry about the specific setting here; this is just for illustration.

An example ~/.zshrc config for Z shell

```
# SHELL OPTIONS
setopt EXTENDED_HISTORY ①

# ENVIRONMENT VARIABLES
export EDITOR="vim" ②
export PATH="$HOME/bin:$PATH" ③

# PROMPT
export PS1="> " ④

# ALIASES
alias ls="eza" ⑤
```

① One of the common ways to set shell options is with `setopt` built-in. In this case, Zsh is configured to use "Extended history" that will save timestamps for commands logged into history. Refer to the *Improved History* section of the book to explore history settings for both Bash and Zsh.

② We set the default application for editing files by setting the `EDITOR` env var.

③ Adding new paths to the `PATH` is probably the most common configuration in startup scripts. Some applications edit shell startup scripts as part of their installation to modify the PATH.

④ Configuring the prompt is done with the `PS1` variable. The setting here would recreate the look of the prompt from this book.

⑤ We create an alias to replace one program with another.

When making changes to a startup script, it is necessary to re-apply the configuration from the command line using the command `source` with the path to the shell startup script (`source ~/.zshrc`). That's because the configuration code need to be executed first to take effect. Otherwise, the changes would be visible only in a new shell session.

`source` loads and executes the provided file in the current session and can also be used in scripts to execute other files.

Keybindings

Shell keybindings (keyboard shortcuts) control what happens when keys are pressed on the command line. We have seen already that Bash and Z shells come with useful default keybindings, but we can extend or change them in the shell startup scripts.

Keybindings and character sequences

Bash and Zsh have different approach to see and alter keybindings. We can see the current bindings with the commands `bindkey` in Zsh or `bind -P` in Bash:

Printing keybindings in Zsh

```
> bindkey
"^@" set-mark-command ①
"^A" beginning-of-line
"^B" backward-char
...
"^X^B" vi-match-bracket
"^X^E" edit-command-line
...
"^[^D" list-choices
"^[^G" send-break
...
"!"-"~" self-insert
"^?" backward-delete-char
```

① `bindkey` prints Zsh keybindings. We can see that each keybinding is usually a combination of a character sequence representing the pressed keys and an action to perform (something akin to a function).

The way keys are represented can differ in different environments. A common way to determine the exact character sequence for a key on the keyboard, especially for special keys like arrow keys or function keys, is to use `Ctrl` + `V` (verbatim input) shortcut on the command line:

```
> ①
^[[A ②
```

① Press `Ctrl` + `V` and then a key of your choice.

② The character sequence will be printed on the screen. In this case, `^[[A` refers to the pressed arrow key `Up`.

It might happen that the key press in verbatim input will be captured by another program instead, like your graphical shell, terminal, etc. You will need to either disable the behavior or find another way to get the key sequence.

Creating shortcuts typically involves the use of leading keys like `Ctrl`, `Alt`. The table below summarizes the most common sequences:

Table 7. Character sequences for common keys

Key sequence	Character sequence
`Ctrl` + another `<key>`	`\C-<key>` in Bash or `^<key>` in Zsh
`Alt` + another `<key>`	`\e<key>`
Specific keys	`\t` for Tab, `\r` for Enter

Bash Keybinding Configuration

In Bash, keybindings are defined using the `bind` command, followed by the string containing both the character sequence and the command to invoke.

Changing keybinding for Z shell

```
# Print current directory with F12 key
bind '"\e[24~":"pwd\n"' ①
```

① `^[[24~` stands for `F12`. When pressed, the `pwd` command will be executed right away (and we will see the current working directory printed on the screen). This is achieved with the newline character `\n` at the end.

Zsh Keybinding Configuration

In Zsh, keybindings are defined using the `bindkey` command, followed by the character sequence and the desired ZLE (Zsh Line Editor) widget. Widgets are special actions that are either built-in or user-defined functions. Some widgets are not loaded into memory by default even though they are part of the Zsh standard library. In that case, we need to load them with `autoload` and define them as widgets using `zle -N`. We can use the same

technique to use our custom shell functions as widgets.

Technically speaking, `autoload` makes sure that the function is loaded only when needed, e.g., when invoking the defined shortcut.

Changing keybinding for Z shell

```
# Make the Del character behave as expected
bindkey "^[[3~" delete-char ①

# Use Ctrl x, Ctrl e to jump into editor
autoload -U edit-command-line
zle -N edit-command-line
bindkey '\C-x\C-e' edit-command-line ②
```

① If `Del` doesn't do what you expect in Zsh, set `Del` explicitly to delete a character. We use built-in Zsh widget called `delete-char` to do it.

② Emulate the helpful `Ctrl` + `X`, `Ctrl` + `E` Bash keybinding to open the command in the default editor. `zle -N` defines a custom widget from the `edit-command-line` Zsh function, as it is not loaded by default.

Prompts

While Easy Bash Prompt Generator[2] makes it easier to configure the prompt using the `PS1` variable, most people should opt for installable prompt and command-line status bar tools that provide a superior experience. There are plenty to choose from, e.g., the beautiful and configurable shell plugin Starship[3].

Shell Plugins

Shell plugins are essentially premade shell scripts that provide additional functionality like aliases, functions, keybindings, themes, and other settings. They can be downloaded and sourced like any other script using the `source` command. You will find many different shell plugins for both Bash and Zsh online.

Programs that manage such plugins on our behalf are referred to as *frameworks*. Frameworks usually have some sort of plugin management system that keeps plugins up to date and/or a default set of plugins that are installed and configured to work together. Some of the most famous ones are Oh My BASH[4] and Bash-it[5] for Bash and Oh My

Zsh[6].

Using heavyweight frameworks that ship a lot of functionality out of the box might not be the best idea for learning purposes. Loading dozens and more plugins you don't know well can be troublesome, as changed keybindings and aliases can make things confusing. Using a lot of plugins can also make the shell startup time slow. Alternatively, consider simpler frameworks that provide plugin management only and carefully pick the plugins you want to use.

Dotfiles

Similar to shells, many other applications store their configuration in files that start with a dot (`.`). They are hidden files in Linux and Unix-like operating systems. Developers and sysadmins commonly use the term "dotfiles" specifically for the subset of the hidden files that are relevant for configuring the applications and utilities that we care about. Dotfiles are typically scattered in the user home directory (`~/`) or other application-specific locations between other files and folders.

As with almost anything, we typically want these files to be under some version control system so that we can share them between multiple computers or revert them to a previously working configuration. There are multiple ways to achieve that, but the most common approach is to keep all dotfiles together in one folder and use a version control system like Git to track this folder as a repository.

To achieve that, we can keep dotfiles inside a single location and link them to the original locations (the file paths expected by the applications) using *symlinks*. Symlinks, or symbolic links, are special Linux/Unix files that point to other files or folders. Creating them manually for all our configuration files would be tiresome, but there is an excellent symlink manager called Stow[7] that can do it for us.

There are several steps involved in using Stow to manage dotfiles:

- Install Stow.
- Move respective dotfiles to a new directory like `~/dotfiles`. The folder should be placed inside the home directory because Stow will use the parent folder as the default target location. The structure inside it should mimic the structure of the

original locations, so if a config file was in `~/subfolder/.conf` previously, it should become `~/dotfiles/subfolder/.conf`. For example, to manage Zsh configuration file with Stow, move `~/.zshrc` to `./dotfiles/.zshrc`.

- Run `stow .` (note the dot denoting the current directory) inside your new dotfiles directory, e.g. inside `~/dotfiles`. The command will create all your symlinks.
- Optionally version and back up the directory using a tool like Git.

Once done, you should have a single location for your config files and have symbolic links created for you that point from the original location to your new folder.

Please note that if you are not able to run Stow on your operating system, like Windows, you might need to look for alternative utilities.

Reusing Dotfiles

It may be handy to split configuration into different files. Besides keeping the configuration neat and organized, we might want to share some configuration between multiple shells or load configuration conditionally, for instance, to only use it for the intended operating system.

Let's consider that we want to use a collection of aliases defined in `~/dotfiles/.aliases` in both Bash and Z shells. To do so, we can load the shared config file in both `~/.bashrc` or `~/.zshrc`:

Loading configuration from a file

```
# using source with a check whether the file we want to source exists
if [ -f ~/dotfiles/.aliases ]; then
    source ~/dotfiles/.aliases
fi
```

We will explore conditional expressions in more detail later on in the *Shell Scripts* section.

If we need to make scripts more flexible, we can run code conditionally based on the type of the shell, the operating system used, or even based on a specific machine:

Loading configuration based on certain criteria

```
# example: apply only on Linux
if [[ "$(uname)" == "Linux" ]]; then
    # place the code here
fi

# example: apply only in Z shell
if [[ "$SHELL" == "/usr/bin/zsh" ]]; then
    # place the code here
fi

# example: apply only when the computer's hostname is set to laptop
if [[ "$(hostname)" == "laptop" ]]; then
    # place the code here
fi
```

[1] https://www.nerdfonts.com

[2] http://ezprompt.net/

[3] https://starship.rs/

[4] https://ohmybash.nntoan.com/

[5] https://github.com/Bash-it/bash-it

[6] https://ohmyz.sh/

[7] https://www.gnu.org/software/stow/

Job Control

When a program is executed on a command line, it will write its output directly to the terminal. We are then unable to issue other commands or interact with the command line until it finishes. We say that such commands run in the *foreground* and call them *jobs*. We can also run a program in the *background* if we wish to, detaching its output from the terminal. This detachment is achieved by appending `&` character when issuing commands. Let's see how this works with the program `sleep` that will simulate a long-running command:

```
> sleep 20000 &
[1] 11774
>
```

Once a program is started in the background, we can immediately use the command line again. We also receive two numbers that can be used to control the job. The first is a simple reference number that will allow us to reference the job in the terminal session with `%X` like `%1` and a *pid*, a system process id.

We can use the `jobs` command to get the list of all unfinished jobs started in a shell session. In this case, it will print information about the sleep program running in the background (to see also the process ids, pass the `-p` option):

```
> jobs
[1]  + running    sleep 20000
```

```
> jobs -p
[1]  + 11774 running    sleep 20000
```

We can bring any program running in the background back to the foreground with `fg`, using the job reference number:

```
> fg %1
```

The process can be paused with `Ctrl` + `Z` and resumed later in foreground or background with `fg %1` or `bg %1`.

Every process in a Unix-like operating system can be controlled by sending *signals*. The process id (*PID*) will identify the process. To terminate, suspend or resume a process, we will use a program called `kill`. The basic usage is `kill -SIGNAL <pid>` where *SIGNAL* can be the signal identifier or its number.

Table 8. The most important signals for terminating, suspending and resuming processes

Signal (number)	**Explanation**
SIGINT, SIGTERM (15)	Interrupts/terminates a process gracefully. The program is allowed to handle the signal. `Ctrl` + `C` generates SIGTERM for the running process. This shortcut is often used on the command line to stop a running command.
SIGSTOP (19)	Suspends a process which can be resumed later. `Ctrl` + `Z` generates SIGSTOP.
SIGCONT (18)	Resumes a suspended process.
SIGQUIT (3)	Terminates a process, cleans up the resources used, and generates a core dump. `Ctrl` + `\` generates SIGQUIT.
SIGKILL (9)	Kills the process without the cooperation with the program. It is typically used when the program is not responding to other signals.

Let's see it in action on our process, assuming it is currently suspended in the

background:

```
> kill -SIGCONT 11774 ①
> kill -9 11774 ②
> jobs ③
>
```

① The program will be resumed in the background.

② The program execution is ended with the signal number 9 (we could have used `-SIGKILL` instead).

③ After the process is killed, the `jobs` program won't output anything since there are no more unfinished jobs started in the session.

Finding the process id

It is also possible to control a process that wasn't started in the current terminal session. The program `top` (table of processes) can be used to list running processes and their ids:

```
> top
top - 16:09:30 up  4:30,  1 user,  load average: 0.69, 0.85, 0.73
Tasks: 307 total,   2 running, 305 sleeping,   0 stopped,   0 zombie
%Cpu(s):  2.0 us,  1.8 sy,  0.0 ni, 95.5 id,  0.3 wa,  0.3 hi,  0.2 si,
0.0 st
MiB Mem :  15819.2 total,   6701.9 free,   4542.2 used,   4575.1
buff/cache
MiB Swap:   7988.0 total,   7988.0 free,      0.0 used.  10567.8 avail
Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+
COMMAND
   2492 pstribny  20   0 3356388 243780 132092 S  17.6   1.5   6:10.13
gnome-shell
   6520 pstribny  20   0  632084  81636  64916 S  10.3   0.5   0:52.28
tilix
   3247 pstribny  20   0 3930492 515908 159928 S   1.0   3.2  14:53.14
firefox
```

```
...
```

A typical situation might be a process blocking a port that we want to use for something else. For instance, we want to run a development web server on port 80, but the port is already taken, perhaps because we already started it before. We can determine which process occupies the port with the command `lsof` (`lsof -i :80`). The process can then be killed using the process id, effectively freeing up the port for another use. To do that in one go, we can use the *command substitution*:

Using kill and lsof together to free a port

```
> kill $(lsof -t -i:80) ①
```

① `-t` option will switch to terse output that shows only process identifiers. The command substitution `$( )` will replace the command with its result, making it an argument to `kill`. Since it wasn't specified which signal should be sent, `kill` will use `SIGTERM` by default.

System Administration

Accounts and Permissions

We can manipulate system user accounts, switch between them, run programs under different users, and control permissions within a command line. On Unix-like systems, users also belong to one or more groups. Therefore permissions can be set per group, rather than just per user. The primary programs for managing users and groups are listed below.

Utility	Description
`useradd`	Add new users
`userdel`	Remove users
`groupadd`	Add new groups
`groupdel`	Remove groups
`usermod`	Add users to groups
`passwd`	Change passwords for users

It is necessary to have special privileges to perform all these actions, often done by performing such actions as *root* users. A root user is a superuser. It is essentially an administrator's account. For practical and security reasons, other accounts are usually used for administration instead, with the ability to run commands under a root user with the `sudo` command.

```
> useradd -m newuser ①
useradd: Permission denied. ②
useradd: cannot lock /etc/passwd; try again later.
> sudo useradd -m newuser ③
```

① Creating a new user. `-m` option will also create a home directory for this user.

② We can't create a user with the current permissions.

③ By prefixing the command with `sudo` and entering the correct root password, the system allows us to run the command. For this to work, our user needs to be in a special *sudo group* (the concrete name differs based on the OS).

Whenever such error occurs due to running a command under a user other than root, we can type `sudo !!`, and the shell will expand the previously entered command for us.

It is also possible to use the utility `su` to switch to the root user permanently, although this is not usually necessary. Both `sudo` and `su` are for running commands under any other user. If we don't specify the user, they will assume the command should be run under `root`.

Before we move on, let's see how to check if a user is a part of a particular group:

```
> id username ①
uid=1000(username) gid=1000(username)
groups=1000(username),10(wheel),972(docker) ②
```

① The `id` utility displays basic information about users. If no arguments are provided, `id` shows the information about the user running the shell.

② The user is part of three groups, one of them being `username`, since each user has its group as well. `wheel` is a standard group on Fedora systems that allows users to use `sudo`.

File Permission System

Linux/Unix permission system is based on *file permissions*, since all resources are accessible in a virtual file system where everything appears as files. We can always examine the file and folder permissions with `ls -l`. Each file and folder has a user owner, a group owner, and a set of permissions. The first column of the output displays the permissions, while the group and user owners appear later.

```
> ls -l
drwxr-xr-x.  2 group user 4096 Dec 12 15:05 dir1
-rw-r--r--.  1 group user    0 Dec 12 15:05 file1.txt
-rw-r--r--.  1 group user    0 Dec 12 15:05 file2.pdf
```

The first character on the line denotes the file type, `d` for directory and `-` for a regular file. The permission output follows after that. It can be divided into three groups (each group having three characters):

- Permissions of the user that owns the file or the directory
- Permissions for the group the user performing the operation would need to be part of
- Permissions for other users and groups

These three groups decide who can do what with the file or the folder on the system.

The output uses characters like `r` for reading, `w` for writing, and `x` for executing files. The combination of these operations can also be represented by a single number, called *octal notation*. We will use the proper number later to change the permissions.

Allowed operations	Number
Nothing (—)	0
Execute (–x)	1
Write (-w-)	2
Read (r–)	4
Write and execute (-wx)	3
Read and execute (r-x)	5
Read and write (rw-)	6
Read, write, and execute (rwx)	7

When we put together the proper numbers for the user owner, group owner and other users, we will arrive at a three digit number, like `400` for `r--------` or `777` for `rwxrwxrwx`.

Changing Ownership and Permissions

The owner of files can be changed with the utility `chown` (change owner). Imagine a situation when we might want a web server to be the owner of some static assets that it should serve over HTTP. In that case, it is advantageous if the webserver has its user to set the permissions exactly as needed. Once users and groups are set, the permissions set by `chmod` (change mode) will control the allowed operations. Let's see how to apply it in practice.

```
> ls -l
-rw-r--r--.  1 group user    0 Mar  8 17:23 file1 ①
> chown webserveruser file1.txt ②
```

```
> chown :somegroup file1.txt ③
> ls -l
-rw-r--r--.  1 somegroup webserveruser    0 Mar  8 17:23 file1 ④
> chmod 400 file1.txt ⑤
> ls -l
-r--------. 1 somegroup webserveruser    0 Mar  8 17:23 file1
```

① The `file1` is owned by the `group` group and `user` user, allowing read and write access for the owner and read access for everyone else.

② The owner is changed to `webserveruser`.

③ The group owner is changed to `somegroup`.

④ The changes are reflected in the `ls -l` output.

⑤ Now, we redefine the allowed operations. Only the user owner, in our case `webserveruser`, will have the read access. All other users won't be able to do anything with the file.

There are other ways to change the permissions than using the octal notation, as we will see in the section for writing Bash scripts.

Package Management

Operating systems and programming languages feature package managers to install programs and software libraries. We can use them to install software on workstations and servers programmatically. Usually, the software would be packaged and available for our operating system, but sometimes we need to use a different package manager if the application is not available there. For example, the program `thefuck` is written in Python and can be installed with `pip install thefuck` if it is not in the package manager of the operating system. Of course, the package manager `pip` would have to be installed first.

To install software, we will typically need to use `sudo`, as in `sudo dnf install <package>` on Fedora. Some package managers like pip make it possible to install packages only for the current user without sudo, e.g., `pip install --user <package>` instead of performing the system-wide installation with `sudo pip install <package>`.

Table 9. Package managers

Name	Description
dnf[1]	Package manager for Fedora and Red Hat operating systems
apt[2]	Package manager for Debian and Ubuntu operating systems
brew[3]	Package manager for macOS operating system
npm[4]	Package manager for JavaScript
pip[5]	Package manager for Python

Some software might not be distributed through any package manager. In that case, we might need to download it directly or sometimes even build it from the source code.

Disk Usage

There are situations when we need to extract information about the disk space on a command line. For instance, if a server is not responding to requests, it might be because its disk is out of space. Although there are better ways to monitor such things at scale, it is always good to know how to quickly check the disk usage.

To get a nice colorized report of disk space available and used on mounted file systems, we can use pydf[6] which is a variation of the more standard utility `df`. The same information, but with even nicer output, can be displayed by duf[7]. These programs run fast but only display information from the used disk blocks in the file system metadata. What if we are interested in how much space the folders and files take inside a file system?

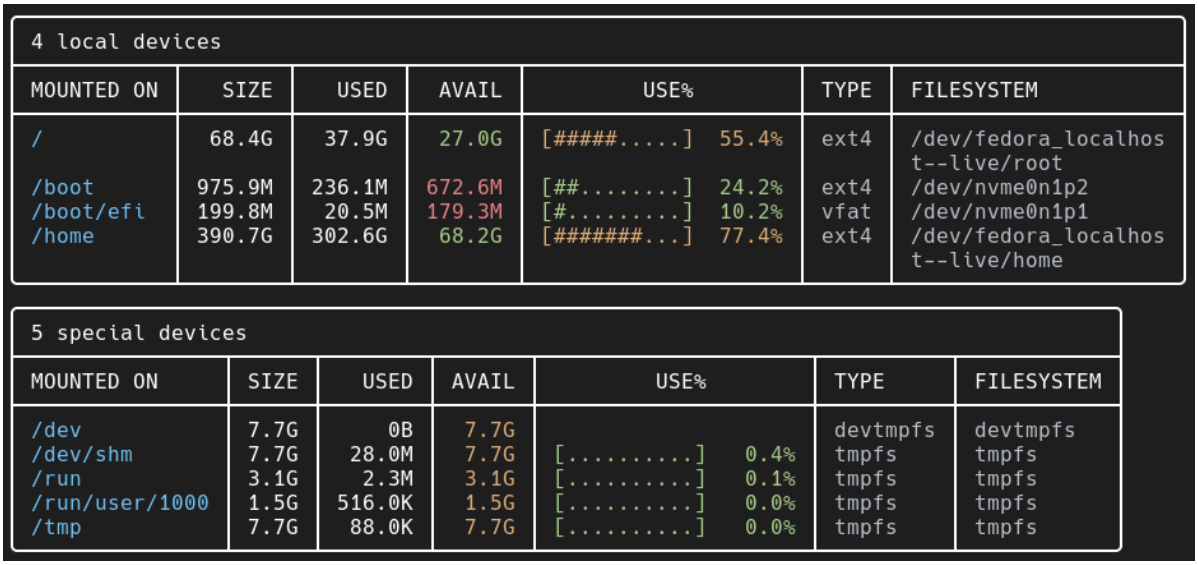


Figure 1. Colorized output of duf, disk usage utility.

To quickly check disk usage in a particular folder, we can use the utility called `du` (disk usage). It will go through the directory tree, examining the sizes of files and summing them up. Run it with `du -h --max-depth=1` to see the overview of just a single directory with human-readable file sizes. Even better than `du` is ncdu[8] (NCurses Disk Usage). It allows comfortable, interactive browsing through the file system, displaying the file and folder sizes in a readable graphical way.

System Monitoring

Top (`top`) is a system monitor that prints information about running processes and their CPU and memory usage so that we can see what is happening on a system. Pressing `H` when `top` is running will display instructions on how to customize its output or use it to kill processes.

While `top` is designed for interactive use, `ps` (process status) can be used to get an overview in a single system snapshot. Because of that, `ps` is a good alternative to use in scripts. `ps` is often run as `ps -aux` where `a` shows processes for all users, `u` displays the process's owner, and `x` also includes processes that are not attached to a terminal.

Instead of using this standard utility, we can install a more powerful alternative called htop[9] with colorized output. If the information displayed is not enough, programs like

tiptop[10] and glances[11] displays additional monitoring information like disk usage or network statistics. WTF[12] goes even further, making it possible to create a terminal dashboard bringing information about practically anything via installable modules.

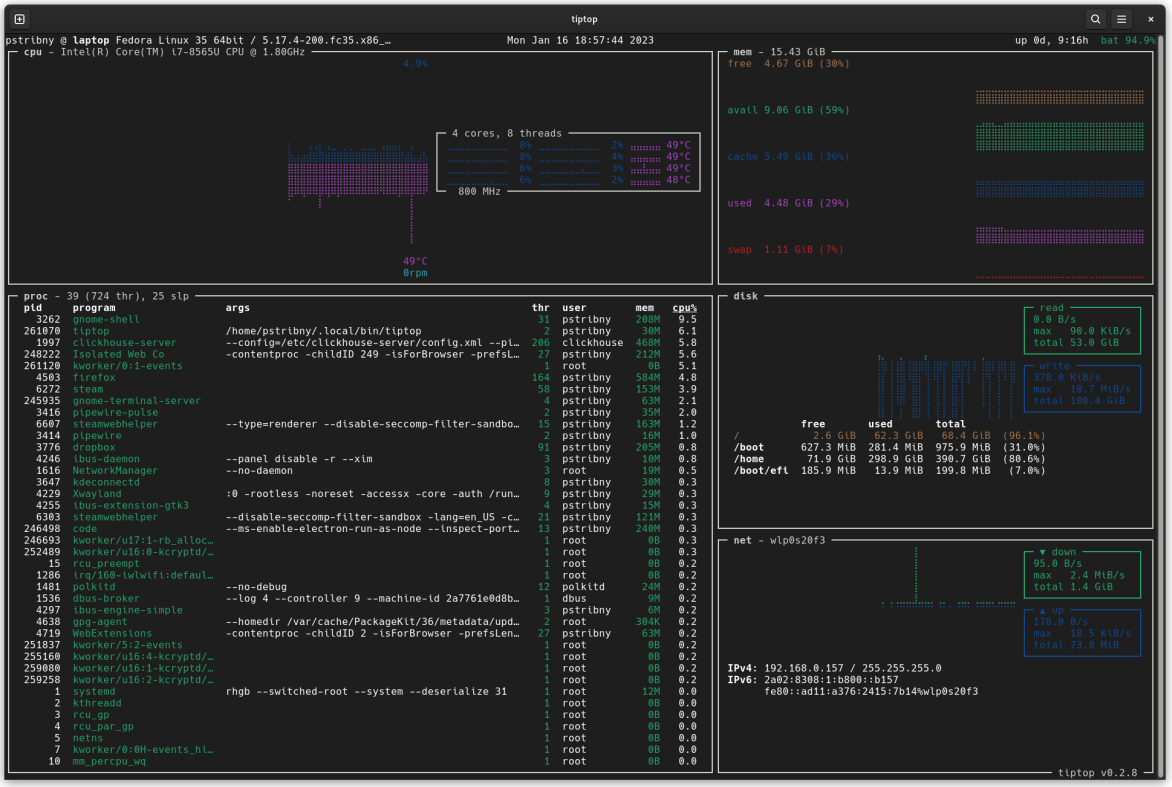


Figure 2. System monitoring with tiptop.

This a reminder that plenty of TUI applications like `htop` can also be controlled with a mouse! There is only sometimes a need for keyboard shortcuts and memorization.

[1] https://en.wikipedia.org/wiki/DNF_(software)

[2] https://en.wikipedia.org/wiki/APT_(software)

[3] https://brew.sh/

[4] https://www.npmjs.com/

[5] https://pip.pypa.io/en/stable/

[6] https://linux.die.net/man/1/pydf

[7] https://github.com/muesli/duf

[8] https://dev.yorhel.nl/ncdu

[9] https://htop.dev/

[10] https://github.com/nschloe/tiptop

[11] https://nicolargo.github.io/glances/
[12] https://wtfutil.com/

File Management

New directories can be created with `mkdir` (make directories). `mkdir -p` can even create a folder in a non-existing location. This option is useful when we want to nest the new folder under a structure that doesn't exist yet. `mkdir` can take multiple directory paths to create multiple folders at once.

Files and directories can be moved and renamed with `mv` (move files), and copied with `cp` (copy files). To delete files, use `rm` (remove files). The most important options are `-i` to prompt for confirmation before deleting files (instrumental when using globbing to prevent accidents), `-r` to delete files recursively, allowing rm also to delete directories, and `-f` never to prompt the user about anything.

Because directories are trees, it can be useful to display them as such. The basic program for that is called `tree`, and we can control how deep the output of it will be with `-L` option, e.g. `tree -L 2`.

Working with folders

```
> mkdir -p work/notes ①
> cp -r work school ②
> tree ③
.
├── school
│   └── notes
└── work
    └── notes
```

```
4 directories, 0 files
> rm -r work school ④
```

① A new `work` directory is created, with another directory `notes` inside it.

② To copy a directory, `cp` requires `-r` (recursive) option, followed by the source and target directory paths. A new directory `school` is created as a copy of `work`.

③ We check that the new folder structure is how we wanted with `tree`.

④ If we don't like it, we can delete both folders at once.

While the commands above for manipulating files and directories are useful and great for scripts, consider using file managers for interactive use instead. Midnight Commander[1] is a two-pane file manager, useful for moving files between two folders or two computers easily. Ranger[2] is a popular file manager with vim-like control, tabs, bookmarks, mouse support, and file previews. There are many new programs nowadays, but both Midnight Commander and Ranger are great choices to start with.

A solid alternative to plain `ls` is eza[3], offering colorized output and many other enhancements.

Viewing Files

The basic program for viewing text files is `cat`. One interesting option when using cat is the ability to print line numbers for the output with `cat -n`. Alternatively use a program called bat[4] which displays line numbers automatically and offers beautiful syntax highlighting for common file types.

Colorized man pages

`man` program for viewing documentation can be configured to display manual pages in color when `bat` is installed.

Configuration for the shell startup script

```
export MANPAGER="sh -c 'col -bx | bat -l man -p'"
```

```
export MANROFFOPT="-c"
```

When dealing with large text files like logs or data sets, we can peak in to see the first few lines with the program `head`, or the last few lines with `tail` (with `-n NUM` as the option for how many lines to show). Since log files can grow continuously, we might want to keep reading the updating log with `tail --follow` or `tail -f` for short. This way, new log messages are displayed as they come in. When the text output is too wide, use `fold` to reduce the number of characters printed per line. It will output the whole text in a more readable way.

Working with datasets like large CSV files, Excel spreadsheets, or databases, can be done in a neat CLI tool VisiData[5].

To open a file in a default system application from a command line, use the program `xdg-open`. It is helpful for images, web pages, PDFs, or other types of files that we don't want to look at inside a terminal window. I recommend setting up an alias (`alias open="xdg-open"`) so that it is possible to just type `open <file or folder>`.

Comparing files

Comparing two files to each other can be achieved with `diff <file1> <file2>`. It can also compare directories with `diff -r` option. Try icdiff[6] as an alternative that uses two-column colored output by default.

Developers can try difftastic[7], which compares code files structurally, not line by line as other tools.

Editing Text Files

Today, many terminal text editors offer screens split into panes, file navigation, the ability to use language servers for intelligent code completion, themes, plugins, etc. They can completely replace GUI editors for many people, and it is helpful to learn at least one

to perform small editing tasks on the command line.

As discussed in the *Configuration* section, set your favorite editor as the default using the `EDITOR` environment variable. Plenty of command line tools open text for editing in the default editor.

The most famous terminal editor is Vim[8]. Its focus is on efficient ways of editing text using multiple modes, one for writing and others for editing text using key sequences. Modern terminal editors like Neovim[9] and Helix[10] try to improve upon the concept with better support for programming tasks.

As these editors can be a bit complicated for first-timers, they all offer a "tutor". Vim tutor can be launched directly on the command line with `vimtutor`. In Neovim, type `:Tutor` after opening the editor. In Helix, type `:tutor`, hit `Enter`, and follow the instructions on the screen.

Different modes make these editors more difficult to use. If working with modal editors is too complicated, you can use more straightforward and user-friendly Nano[11] or micro[12]. Nano is often installed in Linux distributions by default.

Opening a file for editing is as simple as passing the file path as an argument:

```
> vim ~/notes.txt ①
> nano ~/notes.txt
```

① It is okay to pass a file path to a file that doesn't exist yet. You will be able to save it as a new file later.

Archiving and Compression

It is common to work with compressed and archived files, usually `.tar` and `.zip` files. TAR stands for *Tape ARchive*, created initially to store files efficiently on magnetic tapes. It is a file format and a utility of the same name that preserves Unix file permissions and the directory structure. The files themselves are often called *tarballs*.

Tar itself is not used for compression but only for creating one file from multiple ones. That's why we often see file types like `.tar.gz`, which refer to files that are archived and

compressed afterward with Gzip. Files can be compressed and uncompressed with `gzip` and `gunzip` utilities. Since the archive is typically compressed as one file, it is impossible to modify its contents afterward.

```
> tar -cf myfolder.tar myfolder ①
> gzip myfolder.tar ②
> ls
myfolder  myfolder.tar.gz ③
> rm -r myfolder ④
> gzip -d myfolder.tar.gz ⑤
> gunzip myfolder.tar.gz ⑥
> tar -xf myfolder.tar ⑦
```

① Creating an archive with `tar` requires `-c` to create an archive and `-f` option to write it to a file. The first argument is the name of the desired archive file, and the second argument is a path to files that we want to put into the archive.

② Using `gzip` is simple. Just tell it what file should be compressed.

③ The result is our original folder and the same folder compressed in a tarball.

④ Our folder is archived and compressed for, e.g., a backup, so now we can remove the original folder.

⑤ `gzip` can decompress a file with `-d` option.

⑥ Alternatively, `gunzip` can be used without any options to do the same.

⑦ Finally, we extract the files from the tarball using `-x` option for `tar`.

The process of unpacking a Gzip compressed archive is so common that the `tar` program itself supports doing so in one go with `tar -xvzf myfolder.tar.gz`. The reverse can be done too with `tar -cvzf myfolder.tar myfolder`.

Modern multi-core and multi-processor machines can take advantage of parallel Gzip compression with pigz[13]. There are similar tools for other formats and/or decompression. Look for them in case the compression is too slow for you.

When we need to work with Zip format, commonly used on Windows systems, we can use programs `zip` and `unzip`. Zip was created for MSDOS, so, unfortunately, Zip doesn't preserve Unix file attributes, e.g., a file won't be marked as executable after decompressing. Zip archives are already compressed, and the files are compressed

individually, in contrast to `.tar.gz` files. `zipinfo` command can be used to inspect an archive without decompressing it.

Other compression formats and tools have different tradeoffs regarding the compression and decompression speed, compression size, or the general availability of such tools on different systems. For instance, `xz` is a modern compression tool that is efficient regarding the final file size.

[1] https://midnight-commander.org/
[2] https://github.com/ranger/ranger
[3] https://eza.rocks/
[4] https://github.com/sharkdp/bat
[5] https://www.visidata.org/
[6] https://www.jefftk.com/icdiff
[7] https://github.com/Wilfred/difftastic
[8] https://www.vim.org
[9] https://neovim.io/
[10] https://github.com/helix-editor/helix
[11] https://micro-editor.github.io
[12] https://micro-editor.github.io
[13] https://zlib.net/pigz/

Search

Searching for Files

The standard programs to search for files by their file names are `find` and `locate`. The main difference between them is that `locate` is much faster but can be outdated as it uses an internal database that has to be regularly reindexed. fd[1] is an improvement over find that has sane defaults and uses colorized output.

Using find and locate

```
> touch file1.txt
> find . -name 'file1.txt' ①
./file1.txt ②
> locate 'file1.txt' ③
> sudo updatedb ④
```

① `find` will look for files in the directory specified as its first argument (`.` referring to the current folder).

② `file1.txt` is found.

③ `locate` will not be able to find a recently created file.

④ We can, however, rebuild the index with `updatedb`, and try again.

Command-line fuzzy finder fzf[2] is an interactive tool that can be used to find not only files but also search within any list that can be provided on the standard input. For instance, we can get an interactive search for our history with `history | fzf`.

We can tell fzf to preview files using our chosen tool with `--preview`, e.g. `fzf --preview 'cat {}'` will display any selected result using `cat`.

Searching within Files

Many times, searching files by their file name is not enough. That's where standard `grep` and its alternatives ripgrep[3] and ripgrep-all[4] come handy. They search for patterns inside files to find the exact matching lines.

Grep searches files (or the standard input) by patterns. It will print each line that matches the pattern, along with the name of the file where it was found. By default, the pattern is assumed to be a regular expression. Positional arguments after the pattern tell grep which files or directories to search.

Grep examples

```
> grep -r '\# [A-Z]' ~/work ~/uni ①
/home/username/work/cli.md:## Terminal
/home/username/work/cli.md:### Shells
...
> grep -Ern 'def |class ' . ②
./app/models.py:8:class User(db.Model):
./app/models.py:22:    def __init__(self, username, email, password,
created_on):
...
```

① Assuming there is a bunch of Markdown notes in folders `work` and `uni`, this prints all headings (lines where a capital letter follows `#`). `-r` is for recursive search to search within folders.

② We search the current directory for all Python function and class definitions. `-n` prints the line numbers, while `-E` makes it possible to use extended regular expressions.

Searching the standard input

Grep is often used to perform a search on the standard input. Typically, text lines are collected from the output of a program and passed to grep using the pipe operator.

```
> history | grep -w grep ①
  320  grep -Ern 'def |class ' .
  321  history | grep -w lsof
...
> compgen -c | grep mysql ②
mysqlcheck
mysql_install_db
mysql_upgrade
mysql_plugin
mysqldump
...
```

① We search the command history for all commands that were executed before. `-w` will match only exact words so that only commands that have a standalone "grep" string are matched. If we don't remember what we did, this is an excellent way to recover some lost knowledge.

② The `compgen` utility lists possible command auto-complete matches. We can use its database to find executables with a particular name. In this case, we look for all utilities that have "mysql" in the name.

Ripgrep is a fast alternative to grep with better defaults. It automatically shows line numbers, ignores files from `.gitignore` files, and skips hidden and binary files. It can also search compressed files.

Ripgrep usage

```
> rg PATTERN [PATH] ①
> rg -F EXACT_STRING ②
> rg -tpy PATTERN ③
> rg --hidden PATTERN ④
> rg --sort path PATTERN ⑤
```

```
> rg --context X PATTERN ⑥
```

① Ripgrep is invoked with `rg` and will automatically search for a regex pattern in the current working directory unless an optional path is provided.

② Exact strings can be matched the same way as in grep with `-F`. This is useful when we need to use characters like `(`, `)`, `[`, `]`, `$` that have special meaning in regular expressions.

③ `-t` option can be used to restrict the search to a particular type of file. In this case, the search will be performed only in Python files. Capital `-T` will do the opposite: it will exclude a specific file extension from the search.

④ To search inside hidden files, use the `--hidden` argument.

⑤ Ripgrep uses parallelism to achieve better speed. If we need a consistent output (for instance, when we want to process it by another tool), we can add `--sort path` option to make the order consistent. This option will, however, disable parallelism.

⑥ We can extend the output of Ripgrep to show surrounding lines with `--context X`, e.g., `--context 20` would show 20 lines around the matched term.

It is a good idea to put a common flag configuration to a `~/.ripgreprc` config file. For instance, a color configuration can be placed here if we don't like the defaults.

For searching within special file types, use ripgrep-all. It can read many different files like PDFs, e-books, office documents, zip files, and even SQLite database files.

[1] https://github.com/sharkdp/fd

[2] https://github.com/junegunn/fzf

[3] https://github.com/BurntSushi/ripgrep/#quick-examples-comparing-tools

[4] https://github.com/phiresky/ripgrep-all

Text Data Wrangling

Working with text data, both structured and unstructured, is an everyday activity. It is handy to make some repetitive text transformations and edits quickly from the command line. This includes tasks such as:

- Connecting outputs and inputs of programs when they don't match exactly
- Replacing values in text-based configuration files. For instance, updating configuration for cron or a database, replacing placeholders for application deployments, and so on.
- Cleaning data for further processing by manipulating CSV, JSON, and other formats
- Converting from one format to another
- Extracting information from log files, including logs from web servers, databases, systemd services, and other applications
- Extracting information from dumps, e.g., from profilers

Working with Sed

Sed is, from its name, a stream editor. It allows us to find and then replace, insert, or delete text from the standard input or a file and put the result to the standard output. If the input is a file, sed can also change it in place.

By default, sed processes text input line by line and performs a specified operation on the text matched by a regular expression. The basic operations are:

- `s` for replacing the matched string
- `c` for replacing the whole line
- `a` for adding a line before the matched search
- `i` for adding a line after the matched search
- `d` for deleting lines

Although we will go through a couple of examples, complete documentation can be found in man pages or online in the sed manual[1].

Replacing Text

We will build upon our previous example of replacing text, but this time we will modify an existing file. The syntax for the replacement operation is `s/pattern/replacement/flags`.

`s` is the operation, `pattern` is the regex to match the string we are looking for, while `flags` control the behavior. Let's now modify a configuration file `.env` with placeholders that we want to replace:

.env file

```
USER_NAME=USER_PLACEHOLDER
USER_PASSWD=USER_PASSWD_PLACEHOLDER
```

Replacing placeholders in a file

```
> sed -i 's/USER_PLACEHOLDER/petr/' .env ①
> sed -i -e 's/USER_PLACEHOLDER/petr/; s/USER_PASSWD_PLACEHOLDER/passwd/'
.env ②
> sed 's/USER_PLACEHOLDER/petr/' .env > .env2 ③
```

① `-i` option can be used to replace a file in place. If we specify a file or files at the end, sed will read them instead of reading the standard input.

② We can perform multiple operations at once by using `-e` option. The operations have to be separated with a semicolon. In this case, we replace both placeholders.

③ Instead of modifying the original file, we redirect the output to the file `.env2`.

One helpful flag is `g`, meaning global. By default, sed will only replace the first occurrence of a match at every line, so we have to use `g` to replace all matches. To replace the word "Unix" with "Linux", the command would look like `echo "Unix is the best OS. I love Unix!" | sed 's/Unix/Linux/g'`.

In the following example, we will look at how to replace tabs with spaces to settle the war between the two once and for all:

```
> sed 's/\t/    /g' tabs.txt > spaces.txt ①
```

① Sed will replace all tabs with four spaces in the file `tabs.txt`. We redirect the output to a new file with `> spaces.txt`.

Sed and Regular Expressions

So far, we have always replaced a plain string with another plain string. The pattern, however, can be a regular expression capturing groups, while the replacement string can include the matched parts.

Consider that we have a text file with full names and ages like this:

people.txt

```
John Doe, 30
Jane Smith, 25
Alice Johnson, 35
```

We can use `sed` with regular expressions and capture groups to swap the first and last names.

```
> sed -E 's/([A-Za-z]+) ([A-Za-z]+)/\2 \1/' people.txt ①
Doe John, 30 ②
Smith Jane, 25
Johnson Alice, 35
```

① Two capture groups are defined in the same way (`([A-Za-z]+)`) to capture two strings with a space in between. The matched parts are then referenced using their numerical positions `\1` and `\2`.

② Without the `-i` option the result is printed on the standard output.

Sed uses POSIX BRE (Basic Regular Expression) syntax. Extended regular expressions can be enabled by `-r` or `-E` option depending on the platform.

Other Sed Operations

Other operations like `c`, `d`, or `a` are written after the matching slash: `/pattern/c`. Let's now introduce the "replace line" operation `c`, implementing the same placeholder replacement as before. Because we are replacing the whole line, we need to specify how the whole new line should look like:

```
> sed -i '/USER_PLACEHOLDER/USER_NAME=petr/c' .env
```

Inserting lines with `a` and `i` operations follow the same syntax, only the text is appended after or prepended before the match. The delete operation `d` can be useful to clean up a file or hide certain lines:

```
> sed -e '/^#/d' /etc/services | more ①
```

① We remove all comment lines that start with `#`. `/etc/services` and many other Linux/Unix locations and configuration files contain such comments, so the output can be cleared this way. Passing the output to `more` command will paginate the results.

Matching Lines With Separate Patterns

Sed also has the ability to match a line first by a separate pattern, match a line between two patterns, and allows specifying on which lines the operation should occur. We can write `/pattern/`, `/beginpattern/,/endpattern/`, `NUMBER` or `NUMBER,NUMBER` in front of the command to specify the lines we want to process, and sed will only execute the command on those lines.

```
> sed -i '/=/ s/USER_PLACEHOLDER/petr/' .env ①
> sed -i '1 s/USER_PLACEHOLDER/petr/' .env ②
> sed -i '1,2 s/USER_PLACEHOLDER/petr/' .env ③
```

① This is the same placeholder replacement as before, but we tell sed to only make the replacement on lines that contain the equality character (`=`).

② The same, but telling sed to only make the replacement on the first line.

③ The same, but telling sed to only make the replacement on the first two lines.

Changing the Separator

Sometimes we need to use slashes in the sed command, e.g., when working with paths. Instead of escaping all the slashes, we can use another separator like `|` or `_`.

Using a different separator in sed

```
> echo '/usr/local/bin' | sed 's|/usr/local/bin|/home/username/bin|'
/home/username/bin
```

A useful flag for experimenting with sed is `--debug`, which will print some debugging information to explain what is happening. If we have multiple sed commands we want to run at once, we can also store them in a file and load the file with the `-f` option.

Working with Records

Awk is another tool to process text files line by line, but `awk` focuses on working with lines as records, dividing each line into separate records based on a separator. It is also a complete programming language with conditionals, loops, variable assignments, arithmetics, etc. There is no possible space to go into the details of Awk, so we will only

look at some examples that are easier done with Awk than sed.

By default, Awk parses each line automatically using any whitespace as a delimiter and exposes the individual records as variables `$1`, `$2`, `$3`, and so on. Note that it is better to always quote Awk script with single quotes because of the dollar symbols so that the shell won't expand them to variables. Leading and trailing whitespace is ignored. Other than that, the simple usage of Awk is like sed: we match lines that we want to work with using a pattern and then perform some action with that line.

A good example might be getting the list of Linux users from `/etc/passwd`. This file lists each user per line, together with some additional information where each record is separated by a colon. The user name is at the beginning:

/etc/passwd

```
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
pstribny:x:1000:1000:Petr Stribny:/home/pstribny:/usr/bin/zsh
```

Processing /etc/passwd with awk

```
> awk -F':' '{print $1}' /etc/passwd ①
root ②
bin
daemon
pstribny
> awk -F':' '$7 == "/usr/bin/zsh" {print $1}' /etc/passwd ③
pstribny
```

① We need to change the default separator from whitespace to a colon using `-F` option. It can take a regular expression, but in this case, our single character `:` is all we need. The block `{ }` is the script that should be executed on each line. The first record will be parsed and available automatically via `$1` and printed.

② The result is the list of all users on separate lines.

③ We can limit what gets processed by a pattern or a conditional. The last part of `/etc/passwd` file points to the shell the user is using. So, in this case, we get all user names of people who have their shell set to Zsh.

While working with CSV files in Awk is tempting, parsing CSV files is much more difficult as the separators are not always clear (the same character might appear escaped or quoted inside records). For CSV it is best to use specialized tools.

`cut` is another utility that can perform similar tasks as awk. It is not a complete language like Awk, but some people might prefer its interface for manipulating record-type lines.

A note on regular expressions

We have seen that programs, like grep and sed, operate with two types of regular expressions that are part of the POSIX standard: Basic Regular Expressions (BRE)[2] and Extended Regular Expressions (ERE)[3]. The syntax of regular expressions was standardized to make some basic system utilities consistent.

We can switch between these two options in those programs, typically using `-E` to enable the extended version. Awk behaves differently, using the extended version by default. Note that other programs or libraries might have their implementation, behaving differently.

Working with Other Formats

While `sed` is helpful for general regex style text manipulation and `awk` for manipulating tabular data, sometimes we want to work with specific file formats. There are many tools available nowadays that focus on specific formats or work with multiple formats at once.

JSON is one of the most commonplace data exchange formats, so it can be useful to process and manipulate it on the command line. A popular modern JSON manipulation tool is jq[4].

Let's say we have a JSON file with the current job offers on the market:

job_offers.json

```
{
```

```
  "jobOffers": [
    {
      "position": "Python developer at Snakes, Inc.",
      "language": "Python"
    },
    {
      "position": "Full-stack developer at WeDoEverythingOurselves",
      "language": "JavaScript"
    },
    {
      "position": "Superstar at GalaxyWorks",
      "language": "Python"
    }
  ]
}
```

Using `jq`, we will extract the popular programming languages people are hiring for:

```
> cat job_offers.json | jq --raw-output '.jobOffers[].language' ①
Python
JavaScript
Python
```

① We feed the JSON file to `jq` using the pipe operator. `--raw-output` will allow us to get a non-JSON string as the result of our query (otherwise, jq builds a JSON object by default - in this case, the strings would have quotes around them). The last part is the jq's internal query language that allowed us to extract the information without much effort.

Wrapping Up

Finally, this section would not be complete without mentioning these commonly used utilities for data wrangling: `sort` for sorting text lines, `uniq` for removing duplicates, and `wc` for word and character counts.

```
> cat job_offers.json | jq --raw-output '.jobOffers[].language' | sort | uniq
-c ①
```

```
      1 JavaScript  ②
      2 Python
```

① `uniq -c` will merge duplicates and print the number of occurences. It expects that the same lines are next to each other, therefore we need to pass the text through `sort` first.

② We get the counts together with the line string. It seems that Python is in demand.

[1] https://www.gnu.org/software/sed/manual/sed.html

[2] https://en.wikibooks.org/wiki/Regular_Expressions/POSIX_Basic_Regular_Expressions

[3] https://en.wikibooks.org/wiki/Regular_Expressions/POSIX-Extended_Regular_Expressions

[4] https://stedolan.github.io/jq/

Basic Networking

`ping` is a famous command to send repeated ICMP[1] echo requests to network hosts. It is used to find out if a remote computer, server, or device is reachable over the network. To use it, follow the ping command with a positional argument representing IP version 4 or 6 address or with a domain name pointing to such address. gping[2] does the same but draws a chart with response times rather than printing the results line by line.

Domain Name System (DNS) can be queried using dog[3]. It is useful for getting information about domain names, e.g., to determine if a performed change in a DNS entry has been propagated already.

Making HTTP Requests

The most common HTTP client for the command line is without a doubt curl[4], appearing in many documentation and testing examples. Curl comes preinstalled on many systems and understands many protocols, not just HTTP. When we need something more user-friendly with fancy syntax highlighting, HTTPie[5] is a great curl replacement for many tasks.

Let's see how we can check the EUR/USD rate by fetching the rates from an HTTP endpoint and processing it on the command line. Let's assume that the online exchange service returns a response in JSON like this:

```
{"amount":1.0,"base":"EUR","date":"2025-04-30","rates":{"AUD":1.7798,"BGN"
:1.9558,"BRL":6.3839,"CAD":1.5728,"CHF":0.9389,"CNY":8.2635,"CZK":24.92,"DKK"
:7.4636,"GBP":0.8518,"HKD":8.8214,"HUF":404.08,"IDR":18902,"ILS":4.1231,"INR"
:96.14,"ISK":145.9,"JPY":162.68,"KRW":1618.39,"MXN":22.202,"MYR":4.9074,"NOK"
:11.809,"NZD":1.9219,"PHP":63.538,"PLN":4.2753,"RON":4.9782,"SEK":10.9715,"SG
D":1.4859,"THB":38.009,"TRY":43.757,"USD":1.1373,"ZAR":21.11}}
```

Combining `curl`, `jq`, and `grep` we can extract what we need:

```
> curl https://api.frankfurter.dev/v1/latest --silent | jq | grep USD ①
    "USD": 1.1926, ②
```

① `curl` takes the URL as a positional argument. `--silent` option will suppress additional curl output about the transfer, outputting only the response text. Because the JSON is not formatted, we pass the output through `jq`, which will place all JSON fields on separate lines. Finally, we will filter the results by the currency we are interested in.

② We get the line that contains "USD". We know that we can exchange 1 EUR for 1.1926 dollars.

The advantage of `curl` is that its syntax is supported across developer tools. For example, GUI HTTP clients can accept and convert curl syntax and export requests as curl commands. Web browser tools can also convert past requests to curl commands for further debugging.

Let's look at the other tool, HTTPie. By default, it is invoked with the easy-to-remember command `http`. In the next example, we will perform two requests. The first request will authenticate us against an endpoint that takes a username and password as JSON and returns a token as a top-level JSON property that we will use to retrieve data from a protected resource in the second request.

The following example references a hypothetical web application running on `localhost:8000`. It will not work on your computer out of the box.

```
> USER=petr@example.com ①
> PASSWD=test
> JWT=$(http POST http://localhost:8000/api/user/token-auth/ username=$USER
```

```
password=$PASSWD --print b | jq '.token' --raw-output) ②
> http http://localhost:8000/api/groups/ authorization:"JWT $JWT" ③
```

① Let's set some variables that we will use later.

② We can tell HTTPie to output only the response body without the HTTP headers (`--print b`), pass the output to the JSON command-line parsing tool `jq` using the pipe (`|`), and extract the top-level token property in the JSON response. `--raw-output` will discard the quotation marks around the `token` property. Finally, the special syntax `$( )` will ensure that the whole command inside it is run first, and its result is passed to the top-level command that assigns it to a shell variable.

③ The `JWT` shell variable (token) is used to define an HTTP authorization header for the subsequent request.

The wget[6] utility can be used to download files, pages, and whole websites since it can follow links on web pages. It is a simple web crawling tool.

SSH and SCP

SSH stands for *Secure Shell*. It is a protocol that we can use to operate remote machines. It is a client-server protocol, meaning that the remote machine has to be configured to support it, while we need one of many client-side applications that can speak the protocol. SSH supports executing commands remotely, port forwarding, file transfer, and other things. The most important applications that we might want to use are `ssh` for executing commands and `scp` or `rsync` for file transfers. We will see them in action shortly.

The authentication with the server can be handled with just a password or a private-public key pair. When we create a new server instance (using a cloud provider like Digital Ocean or AWS), we will be given a machine with a root user and password. Some providers like Digital Ocean can insert our public key on the server automatically. In other instances, we will have to log in to the machine and place our key in `~/.ssh/authorized_keys` file, which is where authorization keys are typically stored in Unix-like systems. We can also use `ssh-copy-id` utility to copy the key for us. Once our key is configured on the remote server and we can authenticate using it, we might want to turn off password-based authentication to increase security.

Creating a private-public key pair

If we don't have our cryptographic key pair yet, we can generate one using a utility called `ssh-keygen`. There are two essential choices to make. The first one is to decide what crypto algorithm we want to use to sign and verify the keys. Today, the most common ones are RSA, with a recommended size of at least 3072 bits, and Ed25519, with a length of at least 256 or 512 bits.

The second one is the choice of whether we want to have a passphrase associated with our private key, effectively protecting it with a password. It makes it more difficult for other parties to use our key when stolen. Some people prefer not to set a passphrase because it makes using the key simpler. But we can set a passphrase and avoid having to type the key password again and again on every use by utilizing `ssh-agent` or `gpg-agent` utilities. This approach gives us a more secure private key but doesn't hinder everyday interactions with SSH-based tools.

How to generate Ed25519 key pair with ssh-keygen

```
> ssh-keygen -o -a 100 -t ed25519 -f ~/.ssh/id_ed25519
Generating public/private ed25519 key pair.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/username/.ssh/id_ed25519.
Your public key has been saved in /home/username/.ssh/id_ed25519.pub.
The key fingerprint is:
SHA256:PNGnePjvoVL/nVDPq9HrBsF/tYo5HxVoull+FK3LMPs username@hostname
...
```

Check the man page `man ssh-keygen` for details.

Based on the type of key we decided to use, we should have our key pair stored as `~/.ssh/id_rsa` (the private RSA key) and `~/.ssh/id_rsa.pub` (the public RSA key), or `~/.ssh/id_ed25519` (the private Ed25519 key) and `~/.ssh/id_ed25519.pub` (the public Ed25519 key). The content of the `*.pub` files is what we need to get in the `~/.ssh/authorized_keys` on the server. The `~/` location has to match our intended remote user, e.g. `/home/root`.

Let's now see how we can log into a remote machine using `ssh` and execute commands in a remote session:

Connecting to a server on 10.5.6.22 as username via SSH

```
> ssh username@10.5.6.22 ①
username@server> ②
```

① If our key is not recognized on the server, we will be asked for a password. If we generated a private key with a passphrase, we would have to enter that passphrase now.

② If the authentication succeeds, we are given access to a remote shell session to execute commands on the remote machine. Now is the time when we might want to employ job control techniques or terminal multiplexers to run long-running programs so that we don't have to maintain a permanent SSH connection.

Another way is to execute remote commands in our local shell session. We can do so by simply placing the command as another positional argument to `ssh`:

```
> ssh username@10.5.6.22 "ls ~/"
```

Note that we need to quote the command if there are any spaces, otherwise it would not be recognized as one positional argument. We can also issue multiple commands at once and use other techniques that we saw earlier, like piping.

SSH Configuration

On our local machine, SSH is configured using `~/.ssh/config` file. Here we can predefine our remote connections. This file is useful for aliasing remote servers or specifying various configurations about the remote servers, e.g., when we need to use different key pairs or need to establish a tunnel connection through a bastion host (a server that can connect to other servers to prevent internal servers from being exposed to the internet).

SSH configuration example

```
Host myserver ①
       Hostname 10.5.6.22
       User username
       IdentitiesOnly yes ②
       IdentityFile ~/.ssh/id_ed25519 ③
Include work_hosts/* ④
```

① We name the remote server. We can then refer to it when using SSH-based tools with that name, e.g., `ssh myserver` instead of `ssh username@10.5.6.22`.

② IdentitiesOnly will only send our specified identity (`~/.ssh/id_ed25519`) to the server for authentication. If not set, SSH will send all our local identities for the server to pick the correct one.

③ This is the path to our private key that we want to use to authenticate with this remote server.

④ We can use `Include` to include other files in the configuration. This directive is useful when we want to source the configuration from our dotfiles elsewhere or keep hosts configurations separate in individual files. Globbing is utilized to include multiple matching files, in this case, all files found in the `work_hosts` directory.

Dealing with Long Running Programs

When running programs over SSH, we should ensure that a long-running program will finish even if we get logged out of the system or get disconnected from the running shell. That's what `nohup` command is for. The usage is simple. Just append the command to execute with its arguments after `nohup` like `nohup [command] [arguments]`:

```
> nohup ping google.com >> ping.txt & ①
[1] 272502 ②
> exit ③
```

① It is a good idea to append `&` at the end to start the program executing in the background (otherwise, it is impossible to continue using the current shell). We redirect the command's output to a new file, `ping.txt`. If we don't, `nohup` will store the output in the default file `nohup.out`.

② The command is sent to the background. See the *Job Control* section on how to

manipulate background jobs.

③ We can safely exit the shell. Once this is done, open a new shell and look at the `ping.txt` file. It will still grow continuously since the ping command wasn't killed.

A complete replacement for interactive SSH sessions is Mosh[7], a mobile shell that supports roaming. The problem with SSH sessions is that they require a permanent connection to the remote machine. With Mosh, we can lose connection on a train, put our laptops to sleep, and continue the work without starting a new session.

File Transfers

Copying files from and to servers is very useful for managing configuration files or doing backups. `scp` is a program for secure file transfers over SSH. Let's have a look at how to use it to transfer a local file to a remote host:

Transfering files with SCP

```
> ls
dir1 file1.txt ①
> scp file1 username@10.5.6.22:~/file1 ②
file1.txt        100%    0     0.0KB/s   00:00 ③
> scp -r dir1 myserver:~/dir1 ④
> scp username@10.5.6.22:~/file1 file1 ⑤
```

① Let's assume we have a file `file1.txt` and a directory `dir1` that we want to move to our remote server.

② `scp` expects the source location first, followed by the target location. We use `username@10.5.6.22` to specify our target server, the same as with SSH. The target file path is separated by a colon.

③ SCP will output basic statistics about the state of the transfer. This was a transfer of an empty file that was very quick, hence the zeros.

④ To transfer directories, we need to use `-r` (recursive) option, as with many other standard commands. In this case, we are referring to our server as `myserver`, thanks to the SSH configuration we did before.

⑤ To download the file back from the server, we just need to switch the arguments so that the remote location is our source path.

`rsync` is a great file transfer utility that improves upon scp by copying only necessary files

to perform a synchronization. The basic usage is similar to `scp`, but it can be enhanced with `rsync -av --progress`. This way rsync will preserve things like file permissions and output more details during the transfer.

`rsync` can be used locally as well. It is possible to synchronize directories on a local file system for backups or use it on the remote server to synchronize a Git repository with a deployment folder.

[1] https://en.wikipedia.org/wiki/Internet_Control_Message_Protocol

[2] https://github.com/orf/gping

[3] https://github.com/ogham/dog

[4] https://curl.se/

[5] https://httpie.io/cli

[6] https://www.gnu.org/software/wget/manual/wget.html

[7] https://mosh.org

Terminal Multiplexers

If your terminal works as a tiling terminal emulator, you might already use multiple panes with separate shell processes. Having multiple terminals on the screen within one window can be very useful. However, terminal multiplexers have a concept of sessions that can be attached and detached as necessary, making it possible to leave work and return to it later. This feature is useful when working with remote servers, as it means that we can start multiple shells and long-running programs in them, disconnect and connect to them later.

Consider whether you really need a terminal multiplexer. Terminal multiplexers implement their keyboard shortcuts and scrollback buffer (among other things), potentially changing the shell interactions you will have inside them. They are a good addition to the toolbox of advanced users that live on the command line, or for people administrating remote servers via SSH.

Tmux[1] is a popular terminal multiplexer that can open multiple sessions, multiple windows (screens) in each of the session, and split the screen into various panes for each window. It is easy to switch between panes, windows, and sessions, and sessions can also have custom names, making it easier to switch between them or come back to them another time.

Because of a client-server architecture, we interact only with the client, and the server can continuously run in the background, which will preserve our shells when we need to disconnect from a remote server. Another advantage is that Tmux will run in any supported shell, whereas tiling terminals are usually platform specific. That means we can learn something once and use it in many places.

Tmux client can be started by typing `tmux`. If we want to have multiple sessions with their own windows and panes, we can create a second (third) session with `tmux new -s <sessionname>`, or re-enter a session with `tmux attach -t <sessionname>`. The name is optional. Tmux will assign a number to the session so they can be referred. We can provide the session number instead of the name (`tmux attach -t 0`) when attaching.

A session will be killed with `session kill-session -t <sessionname>`. Otherwise, it will cease on system reboot or when the Tmux server process ends. Listing all current sessions is available via `tmux ls`.

Once Tmux is open, we are given a standard terminal shell, but with the ability to issue special commands that will manage panes, windows and the ability to switch between them. Also, a status bar with the list of windows will be displayed below. Any pane or window can be discarded by typing `exit`.

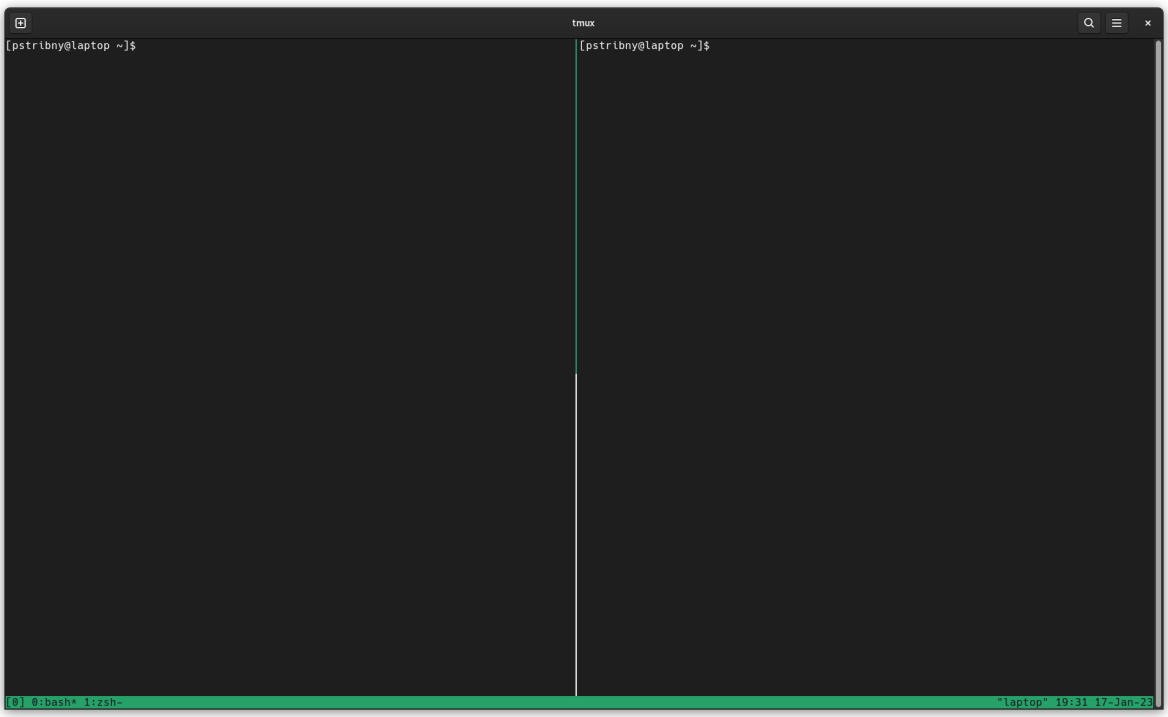


Figure 3. Tmux in action with two windows and the selected window split into two panes.

Let's have a look at an example session:

```
> tmux ①
> ②
> ③
> tmux ls ④
0: 1 windows (created Mon Jan  9 20:24:30 2023)
> tmux attach -t 0 ⑤
> ⑥
> ⑦
> ⑧
> ⑨
> session kill-session -t 0 ⑩
```

① Tmux is started with an implicit first session and the first window.

② Pressing `Ctrl` + `B`, then `"` will instruct Tmux to split the window into two panes.

③ Pressing `Ctrl` + `b`, then `D` will detach the session and return us to our original shell.

④ We can list all sessions. Currently, there is one session that we opened with one window and two panes.

⑤ The session will be reattached, and our two panes reappear.

⑥ Pressing `Ctrl` + `B`, then `C` will open a new window.

⑦ Pressing `Ctrl` + `B`, then `0` switches to the previous two-pane window.

⑧ Pressing `Ctrl` + `B`, then `Left` will switch the focus on the left pane.

⑨ Pressing `Ctrl` + `b`, then `D` will detach the session and return us to our original shell.

⑩ Optionally, we can delete the session.

Table 10. Operations in a Tmux session

Operation	**Shortcut**
Open a new window	`Ctrl` + `B`, then `C`
Switch to the window number 0 (or other)	`Ctrl` + `B`, then `0` (the window number)
Switch to the next window	`Ctrl` + `B`, then `N`
Switch to the previous window	`Ctrl` + `B`, then `P`

Operation	Shortcut
Split the window into panes vertically	`Ctrl` + `B`, then `"`
Split the window into panes horizontally	`Ctrl` + `B`, then `%`
Select the pane on the right	`Ctrl` + `B`, then `Right`
Select the pane above	`Ctrl` + `B`, then `Up`
Select the pane below	`Ctrl` + `B`, then `Down`
Select the pane on the left	`Ctrl` + `B`, then `Left`
Select the pane on the right	`Ctrl` + `B`, then `Right`
Expand the current pane upwards	`Ctrl` + `B` (holding), then `Up` (repeatedly)
Expand the current pane downwards	`Ctrl` + `B` (holding), then `Down` (repeatedly)
Expand the current pane leftwards	`Ctrl` + `B` (holding), then `Left` (repeatedly)
Expand the current pane rightwards	`Ctrl` + `B` (holding), then `Right` (repeatedly)
Close the current pane	`exit` or `Ctrl` + `D`
Detach from the session	`Ctrl` + `b`, then `D`
List available commands	`Ctrl` + `b`, then `?`

It can be helpful to configure Tmux further. Besides themes that can modify the status bar, we can configure different shortcuts (especially replacing unintuitive `"` and `%` with `|` and `-` for splitting panes) or enable mouse control. Mouse control will allow us to point and click into a pane to select it and resize panes with mouse dragging.

The config file might be at `~/.tmux.conf`.

Tmux configuration

```
# split panes using | and -
bind | split-window -h
bind - split-window -v
unbind '"'
unbind %
# enable mouse control (clickable windows, panes, resizable panes)
set -g mouse on
```

tmuxinator[2] is an exciting project for every serious Tmux user. It can keep per-project configurations for the windows and panes we need, including what command should be started in each shell.

Zellij[3] is a great Tmux alternative. Although not being around as long, it offers user-friendly interface and modern features.

[1] https://github.com/tmux/tmux/wiki

[2] https://github.com/tmuxinator/tmuxinator

[3] https://zellij.dev

Better Command Line Experience

Better completions

Both Bash and Zsh offer autocompletion for basic commands, arguments, and file paths accessible through the `Tab` key. Zsh is better out of the box as it cycles through the options when `Tab` is pressed repeatedly. We can achieve similar behavior in Bash with a little bit of configuration:

Improving Tab completions in Bash

```
bind 'TAB:menu-complete'
bind 'set show-all-if-ambiguous on'
```

Zsh autocompletion is still better as it shows descriptions for command arguments.

To improve the completions database in the shell, consider installing shell plugins bash-completion[1] for Bash or zsh-completion[2] for Zsh. Once installed, we will be able to

complete most of the things on the command line.

Although rarely needed, it is possible to define custom completions. This might come in handy if you want completion support for your own scripts and programs.

One important thing that I recommend to configure is case-insensitive completions, because it can be annoying not to be able to autocomplete file paths just because of uppercase letters. Let's see how to do it:

Example autocomplete setting to ignore case sensitivity

```
# ZSH
autoload -U compinit && compinit
zstyle ':completion:*' matcher-list 'm:{a-z}={A-Za-z}'

# Bash
bind 'set completion-ignore-case on'
```

When completing file paths specifically, we can use the command-line fuzzy finder fzf[3] that will fuzzy search files and directories, invoked with `Ctrl` + `T` by default. We can start an fzf search in any place where a file path is expected and also use it to provide multiple file paths at the same time using `Shift` + `Tab` (once the list of options is on the screen). There are also shell plugins to use fzf for shell command completions if you don't like the default completion experience in your shell.

Faster Navigation

There are many ways to improve navigation (switching directories) on the command line. We will focus on three approaches that are the most impactful:

- Using file managers and interactive tools to comfortably navigate between locations.
- Using specialized programs to jump to locations.
- Leveraging the `CDPATH` variable to extend the location search when using the `cd` command.

Using interactive tools

Command-line fuzzy finder fzf[4] has built-in support for navigating subdirectories. The default key binding to invoke it is `Alt` + `C`. It works great for quickly accessing any subdirectory, the disadvantage is that we cannot search in a different location by default.

Terminal file managers provide powerful tools for navigating the file system. While they often cannot directly change the current working directory, scripts are available to configure shells to switch to the last navigated directory upon exiting the file manager. The setup process varies depending on the combination of shell and file manager, so you will need to find the appropriate script for your specific tools. Despite this limitation, using file managers for directory navigation can be highly comfortable and worth considering.

Using jump tools

There are several "jump tools" that can change the current working directory to any location in the file system. One of the most popular ones is zoxide[5]. It remembers which directories we use and uses fuzzy matching to determine the most likely directory to jump to. Once installed, it can be invoked with the `z` command where its arguments describe the location we want. Beyond fuzzy matching, `z` also works as `cd` replacement with absolute and relative paths. Zoxide and fzf can work together to provide interactive search over the most used directories.

The following table shows how it works:

Table 11. zoxide usage examples

Command	Description
`z proj`	changes `$PWD` to the dir matching `proj`, such as `/home/user/projects`
`z p myp`	changes `$PWD` to the dir matching both `p` and `myp`, such as `/home/user/projects/myproject`
`zi`	opens interactive selection using fzf
`zi proj`	opens interactive selection using fzf for locations matching `proj`

Using CDPATH

We can leverage `CDPATH` to provide locations to be searched when using the `cd` command. `CDPATH` is a colon-separated list of directories in which `cd` looks for destinations, similar to the `PATH` variable shells use to find executables. When configured, `cd` will look for locations in `CDPATH` if the typed folder is not in the current working directory (`$PWD`).

Using CDPATH

```
> export CDPATH="/home/user/projects:$CDPATH" ①
> cd ②
> echo $PWD
/home/user
> cd myproject ③
> echo $PWD
/home/user/projects/myproject
```

① We set `CDPATH` to the `projects` folder inside the user's home directory.

② Plain `cd` without arguments navigates to the home directory by default.

③ With the new configuration, `cd` will change the current working directory to `myproject` inside `projects` folder even though `projects` folder is not the current working directory.

If you want the setting to persist, you will need to export `CDPATH` in your shell configuration file.

Improved History

To get the most out of the shell history feature, we need to configure it; we should make sure that enough history entries are preserved, that history is saved to a file, and that correct commands or command metadata are saved. There are a number of different options available for Bash and Zsh; we will have a look at some sensible options to set for both. Feel free to grab only the setting that works for you.

History configuration for Bash

```
HISTFILE=~/.bash_history # Pick a file where to store the history
HISTSIZE=100000  # Number of entries kept in memory
HISTFILESIZE=100000  # Number of entries kept in the file
HISTTIMEFORMAT='%F %T '  # Entries will be stored according to a pattern, in
this case with date and time
HISTCONTROL=ignoreboth  # Don't store duplicates and entries starting with a
space
shopt -s histappend  # Append history entries instead of overwriting them.
Useful when working with multiple shell sessions that would otherwise
overwrite the history file with their own entries only
```

History configuration for Zsh

```
HISTFILE=~/.zsh_history # Pick a file where to store the history
HISTSIZE=100000  # Number of entries kept in memory
SAVEHIST=$HISTSIZE  # Number of entries kept in the file
setopt EXTENDED_HISTORY  # Logs the start and elapsed time
setopt INC_APPEND_HISTORY  # Add entries immediately, don't wait until the
end of session
setopt SHARE_HISTORY  # History entries will be the same for all started
shells
setopt HIST_IGNORE_DUPS  # Don't store duplicates for previous commands
setopt HIST_IGNORE_ALL_DUPS  # Don't store duplicates for all commands
setopt HIST_FIND_NO_DUPS  # Don't offer duplicate entries on interactive
search (Ctrl+R)
setopt HIST_IGNORE_SPACE  # Don't store entries starting with a space
```

By now, we have already covered the basic history navigation, configuration, and using `grep`, but the search within history is still not that practical. The simplest way to improve

it is to use, again, the command-line fuzzy finder fzf[6]. When set up properly, it will replace the standard `Ctrl` + `R` shortcut with its superior interface.

There are many other interesting ways to improve history with various shell plugins. One of my favorites, for example, is zsh-history-substring-search[7] for Zsh that improves the `Up` and `Down` experience, cycling only through commands that have a partial match with the text on the command line. It provides a bit faster experience for situations where we are interested in the most recent invocations.

Last but not least, specialized programs can be used to replace shell history entirely, like McFly[8] or Atuin[9]. These utilities try to be a bit smarter in the suggestions, although you might not need it. Atuin can be used if we need to sync history between multiple computers.

[1] https://github.com/scop/bash-completion

[2] https://github.com/zsh-users/zsh-completions

[3] https://github.com/junegunn/fzf

[4] https://github.com/junegunn/fzf

[5] https://github.com/ajeetdsouza/zoxide

[6] https://github.com/junegunn/fzf

[7] https://github.com/zsh-users/zsh-history-substring-search

[8] https://github.com/cantino/mcfly

[9] https://github.com/atuinsh/atuin

Task Runners and Build Tools

When working on multiple projects, using global aliases or just a collection of shell scripts don't cut it. We need to get organized a provide a good entry point to a project's folder with common tasks that can be quickly listed and executed. For that, we are going to use GNU Make[1].

Using Make, we get a plain task runner that can run commands specific to a particular directory and a complete build system designed for compiling applications from source files. This feature makes it possible to model dependencies between tasks (chain them) and to skip steps in our defined workflow if it is not necessary to rerun a particular task.

The tasks that we want to run are called "targets". A target can be just a name of a task or the name of a file that the task will build (create). If a particular file exists, it means it has already been built, and the task won't be executed again.

Make targets live in a file called `Makefile` that we can place in any folder where project's files, scripts, and other artifacts will be located. Let's have a look at a simple `Makefile` that can build a `file.txt` and offers a `clean` task (target) to delete it:

```
SHELL := bash ①
.ONESHELL: ②
```

```
.SHELLFLAGS := -eu -o pipefail -c ③

.PHONY: clean ④

all: file.txt ⑤

file.txt: ⑥
	touch file.txt ⑦
	echo "Hello from Make" >> file.txt

clean: ⑧
	rm file.txt
```

① Tell Make to use Bash as the language of choice in the targets.

② `.ONESHELL:` configuration line might not always be necessary. It instructs Make to reuse a single shell session for running targets instead of running them in separate shells.

③ We pass our choice of flags to the executing shell. These flags are commonly used in Bash scripts; we will learn more about them later.

④ The list of targets separated by a space should contain all targets that don't use the dependency system and should always be executed when called.

⑤ `all` is a special target because it is defined first (we can name it however we want). It specifies the list of targets that should be called by default when running Make without any parameters.

⑥ Our first custom target is `file.txt` because it will create (build) a file called `file.txt`. If the target is called multiple times, the file will be created only once since Make can detect the file already exists.

⑦ We can place commands on separate lines that should be executed one by one. Note that the space at the beginning is a tab character (this is important).

⑧ Tasks (targets) don't have to follow the output file names, especially not tasks that don't specifically build a particular file.

We can now execute the tasks!

```
> ls
Makefile ①
> make ②
touch file.txt ③
echo "Hello from Make" >> file.txt
> make ④
make: Nothing to be done for 'all'.
> make clean ⑤
rm file.txt
> make file.txt ⑥
touch file.txt
echo "Hello from Make" >> file.txt
```

① The Makefile is created, and we can call `make`.

② By typing `make`, all targets (tasks) from `all:` are run.

③ Make prints the commands while executing them.

④ Running the targets again doesn't recreate the file as it already exists.

⑤ We can run individual targets with `make [target]` like `make clean`.

⑥ After the file is deleted, the target will "build" the file again.

For our second example, let's imagine we want a project that downloads a potentially large data file from a server, modifies it, and then plots it on the command line. If the file is already downloaded, we don't want to redownload it again. If it is already processed, we don't want to process it again. To create a plot, we are going to use a program called YouPlot[2]. Let's jump straight to the new `Makefile`:

```
SHELL := bash
.ONESHELL:
.SHELLFLAGS := -eu -o pipefail -c
.SILENT: ①
.PHONY: clean

all: report

data/archive.zip:
	mkdir data
```

```
            wget https://github.com/stribny/public/raw/master/command-line-
book/imdb.zip -O $@ ②

data/movies.csv: data/archive.zip ③
            unzip data/archive.zip -d data
            touch $@ ④

report: data/movies.csv ⑤
            csvcut -c year_of_release data/movies.csv \
            | sed 1d | grep -Eo '[0-9]{4}' | sort -nrk1 | uniq -c \
            | awk '{print $$2, $$1}' \ ⑥
            | uplot bar -d ' ' -t "Year" -c blue

clean:
            rm -r data
```

① Prevents Make from outputting every command.

② Special variable `$@` can be used as the target's name. Here it means `data/archive.zip`. With `wget -O`, we can download a file to our desired location.

③ After the target's name, we can list all dependencies. In this case, the file `data/movies.csv` can be created only if `data/archive.zip` already exists. If it doesn't the file will be downloaded first.

④ For some reason, Make could not properly recognize the output from `unzip` as already-built target. `touch` will update the file's timestamp and the file will be now recognized as built when running the target multiple times.

⑤ Our main target is called `report`. It uses tools like `csvcut` from csvkit, `sed`, `grep`, `sort`, `uniq`, and `awk` to prepare tabular data as the input for `uplot`.

⑥ Beware that using the double dollar sign `$$` is necessary instead of one. That's because of the unusual behavior of Make.

The downloaded `imdb.zip` file contains a CSV file with popular movies and their release dates (among other things). The `report` target will efficiently make a bar chart of different years with the number of popular released movies each year. It is done through several steps:

- `csvcut -c year_of_release` takes the `year_of_release` CSV column
- `sed 1d` removes the header line from the CSV data

- `grep -Eo '[0-9]{4}'` takes only a 4-digit number from the original input which is in the format `(1989)`
- `sort -nrk1 | uniq -c` sorts the result and adds a count for each year
- `awk '{print 2, 1}'` reverses the columns
- `uplot bar -d ' ' -t "Year" -c blue` creates a blue bar chart, with the added label

The result can be seen in the picture below:

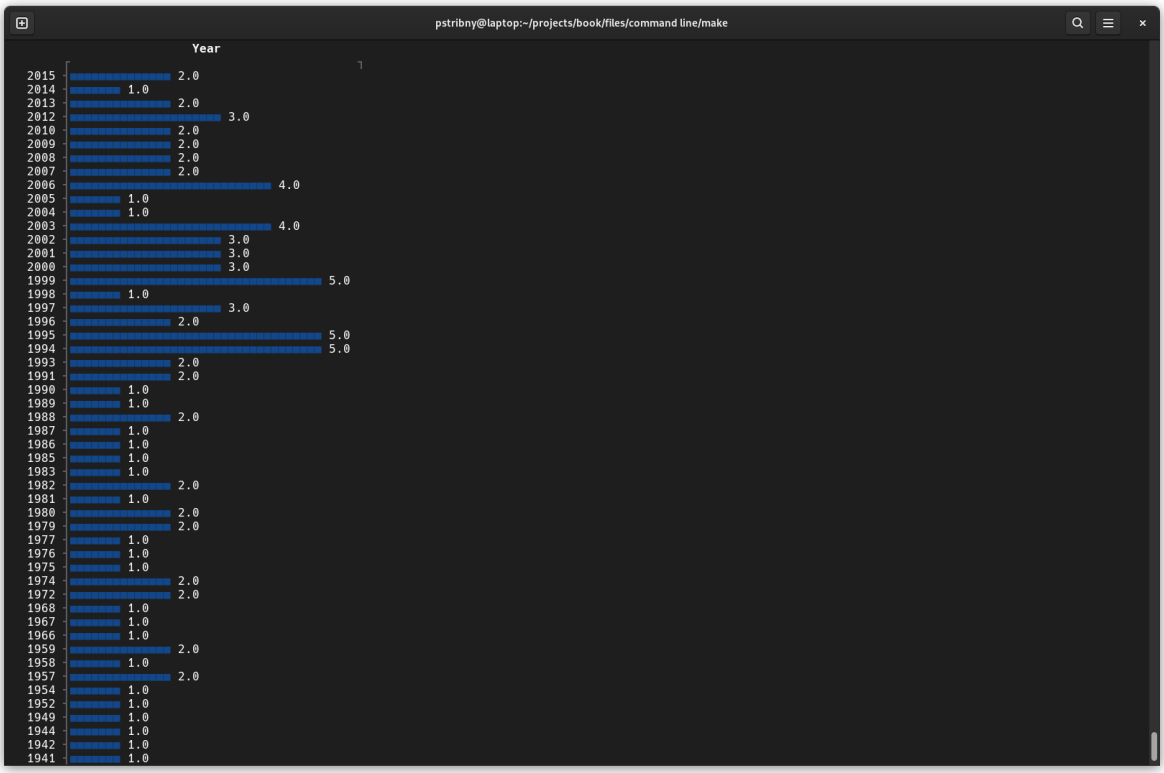


Figure 4. Creating a bar chart with a Makefile

If the building capabilities of Make are not necessary, we can opt for a pure task runner like just[3], where we can also reuse our Bash knowledge or use task runners designed to run tasks in different programming languages.

[1] https://www.gnu.org/software/make/

[2] https://github.com/red-data-tools/YouPlot

[3] https://github.com/casey/just

Shell Scripts

A shell script is a type of executable text file that is understood by the shell. It can execute just one command or group many commands with conditions, loops, functions, and other control mechanisms.

We use shell scripts as installation and configuration scripts that set up an environment for programs to run, as startup scripts that are executed after the computer is started, and for automating tasks where it would be tedious and error-prone to type the sequence of commands manually.

To avoid problems with compatibility between different shells, we will only discuss writing Bash scripts. Even if we use a different interactive shell on the command line, there is no problem in running scripts written for a different shell, as we will see shortly.

Our First Shell Script

To create a script, we need to:

- Tell Bash and other shells how to run our text file using a special first line
- Change the file permissions to make the file executable

Let's create a simple script `print.sh` that prints text to the standard output (the file

extension is optional):

print.sh

```
#!/usr/bin/env bash ①
echo "Robots are humans' best friends." ②
```

① The first two characters at the beginning are called *shebang*. Following the shebang, we specify which program should execute our file. It is important because we might want to run the script from Zsh or another shell. It could be enough to specify directly `#!/bin/bash`, but using a program called `env` will make the script more portable to other environments as it will find the correct binary automatically if it is installed in a different location.

② We can put any valid Bash code in the file.

Before we can run it, we need to make this text file executable. We will do it with the command called `chmod`, as described in the *Accounts and Permissions* section. This time, we will not set all permissions at once but only modify them.

```
> chmod u+x print.sh ①
```

① We only allow this file to be executed by the owner of the file with `u+x`. We could also make the file executable by anyone with just `+x`.

To run the script, all we need to do is to type its path:

```
> ./print.sh ①
Robots are humans' best friends.
> bash ./print.sh ②
Robots are humans' best friends.
```

① We prefix our program with a relative path to the current directory (`./`). This way, the shell executes our script and not another program named `print.sh` that might be on the PATH.

② Alternatively, we can pass the script path as an argument to `bash`. In this case, using the shebang inside the script is not required.

Programming in Bash

We have already seen some aspects of Bash programming like the use of variables, short-circuiting operators like `&&`, or conditionals with `if ⋯ fi`. We will now expand this knowledge and see how to create Bash scripts in more detail.

Functions and Arguments

Bash functions make it easy to group commands under one name. They can be defined and called inside our scripts as well as on the command line. They are in some ways similar to aliases, as we will see in the next example. The following alias `homedir` and the function `homedir` do the same:

```
# inside a Bash script, e.g. inside a shell startup script

# alias
alias homedir="cd ~/ ; ls"

# function defined on multiple lines
homedir () { ①
    cd ~/ ②
    ls
}

# function defined on one line
homedir () { cd ~/; ls; } ③
```

① Note the spaces around the parentheses and the curly braces.

② We can place commands one after another on new lines inside the function body.

③ If we want to define a function on one line, we need to separate commands with a semicolon, and also place a semicolon after the last command.

The function will be defined and available in our shell session once we execute the code for its definition, e.g., by putting it in a script file and running it as we saw before. Placing it in the shell startup script will make it available every time a shell session is started.

We can execute the alias or the function simply by its name `homedir`. In this case, the function would change the current working directory to our home directory and print its contents.

Of course, functions can do more than aliases. For instance, functions can take positional arguments that become available through special variables `$1`..`$2`..`$9` based on their position. Let's now modify our function to take a path to the directory instead:

```
anydir () {
    cd $1 ①
    ls
}
```

① We will expect the path to be the first positional argument of the function.

A function with positional arguments can be called the same way as any other program:

```
> anydir /folder/with/files
a.png  a.txt  b.png  b.txt ①
```

① This will be the output of the `ls` command. The current directory will be also changed.

Bash functions don't and can't return values like other programming languages. They can only end with an *exit status* using the `return` statement. Programs and functions should return 0 if all goes well and a non-zero number between 1 and 255 if not.

Let's modify the `anydir` function to return an error status code in case the directory doesn't exist:

```
anydir () {
    if [ ! -d $1 ]; then ①
      echo "$1 not a directory"
      return 1 ②
    fi
    cd $1
    ls ③
}
```

① We use `if` block to check whether the directory exists or not. `-d` means "check if the following directory exists". The `!` operator reverses the condition.

② We return the error exit status 1.

③ If the exit status is not provided by the `return` statement, the exit status of the last executed command will be used.

Execution of the anydir function

```
> anydir /not/existing/dir && echo "Directory exists"
/not/existing/dir not a directory
```

There is also the `exit` statement that can be used like `return`. It can be used outside of functions and will terminate the entire script where it is used, whereas `return` will only end the function, and a script that uses that function can continue to run.

Local Variables

Variables inside functions can be marked as local to limit their scope. It is a good practice to mark variables as local when we need them only within the function.

```
# inside a Bash script
my_func () {
    local my_var="value"
    echo my_var # the value is available
}
echo my_var # the value is not available
```

Conditionals

Table 12. Conditional operators in Bash (< > characters indicate placeholders)

Operator	Checks
`<num> -eq <num2>`	Check if two numbers are equal
`<num> -lt <num2>`	Check if the first number is lower than the second
`<num> -gt <num2>`	Check if the first number is higher than the second
`<str> == <str2>`	Check if two strings are equal
`<str> != <str2>`	Check if two strings are different
`<str> =~ <pattern>`	Check if a string matches the regular expression pattern
`! <expression>`	Can be placed in front of an expression to reverse the condition
`-d <path>`	Check if the directory exists
`-e <path>`	Check if the file exists
`-r <path>`	Check if the file exists and we have permissions to read it
`-w <path>`	Check if the file exists and we have permissions to write to it

Operator	Checks
`-x <path>`	Check if the file exists and we have permissions to execute it

The table lists the most useful conditional operators used in conditional statements. They are used inside single or double brackets. Single brackets execute the inside expression literally, while double brackets allow for more logical comparisons and other features. Double brackets are usually recommended for preventing unexpected behavior.

Basic usage on the command line

```
> [[ 1 -eq 1 ]] && echo "This is equal"
This is equal
> [[ "str1" == "str2" ]] || echo "This is not equal"
This is not equal
```

Basic usage in a script

```
# template
if [[ EXPRESSION ]]; then
    # commands
fi

# example
if [[ 1 -eq 1 ]]; then
    echo "This is equals"
fi
```

Loops

Bash supports both `for` and `while` loops. In the next example, we will create a script that will accept a list of directories as positional arguments and list all files inside them with the `.sh` file extension. To do that, we are going to iterate over all arguments using the `for` loop.

All positional arguments passed to a script or a function are exposed together in a Bash array that we can access with `$@`. Working with arguments in a script file is the same as in functions.

listscripts.sh

```
#!/usr/bin/env bash

function list_scripts_in_dir() { ①
  ls $1 | grep ".*\.sh$"
}

for dir in "$@" ②
do ③
    echo "Scripts in dir $dir:"
    list_scripts_in_dir $dir ④
done
```

① At the beginning, we define a helper function that will list all `.sh` files inside a single folder. We offload the heavy lifting to `ls` and `grep` programs.

② We iterate over the array containing all positional arguments.

③ The `for` expression is followed by the `do`..`done` block that can contain commands as usual.

④ The `dir` variable now stores the current argument at each iteration. We can pass it as the positional argument to our helper function.

Once the script is saved in a file, we can run it:

```
> ./listscripts.sh ~/folder1 ~/folder2
Scripts in dir ~/folder1:
listscripts.sh
print.sh
Scripts in dir ~/folder2:
anotherscript.sh
```

Other available information

Besides accessing positional arguments with `$n` and `$@`, other information is exposed automatically to the script or function. The table below lists some examples of that.

Placeholder	Meaning
`$0`	The name of the script
`$#`	The number of arguments
`$?`	The return code of the previous command
`$$`	The PID of the process running the script

Reading User's Input

If the script is supposed to be interactive, one may want to prompt users for input. We can do so with `read -r`, followed by names of the variables that should store the input.

prompt.sh

```
# inside a Bash script
echo "What is your name and surname?"
read -r name surname
echo "Hello $name, Your surname is $surname"
```

Invoking a script with prompt

```
> ./prompt.sh
What is your name and surname?
> ①
> Petr Stribny ②
Hello Petr, Your surname is Stribny ③
```

① The script prompts the user. The execution of the script is paused until `Enter` is pressed to confirm the input.

② Example of the input given to the script.

Bash Scripting Practices

There are three main practices that we should consider when creating Bash scripts:

- Offering help to the users that will use them (similarly as other tools do)
- Making their execution safer to avoid unintended consequences
- Learning how to debug them

Offering help

One of the most important things to do is to document how a particular script should be used. Providing help is typically done via help options like `-h` or `--help`. Let's see how to output a help text in our script:

script.sh

```
#!/usr/bin/env bash

if [[ "${1-}" =~ ^-*h(elp)?$ ]]; then ①
    echo 'Usage: ./script.sh -h  Shows help'  ②
    exit ③
fi
```

① Check that the first argument passed to the script (`${1-}`) is an option that starts with one or multiple `-` characters and is followed by `h` or `help`.

② Print any necessary information. The following section will detail the special syntax used to retrieve the argument (`${1-}`).

③ Exit the script if help is accessed. We don't want to execute anything else.

When dealing with options as arguments, it is much better to use an option parser like getoptions[1].

Safer scripts

In the previous example, we retrieved the first script argument with `${1-}`, a syntax that can be used to define a default value if the retrieving argument or variable is not set or provided. Any variable can be defined like that:

```
NUM_WORDS="${1:-10}" ①
if [[ "${NAME:-Kevin}" == "Kevin" ]]; then ②
    echo "It is Kevin!"
fi
```

① Defines `NUM_WORDS` as the first function or script argument if it is set. Otherwise, default to `10`.

② For the `if` condition, use a variable called `NAME`, or use the value `Kevin` if it is not defined.

There are also several valuable settings we can opt for in the execution of a Bash script. Use:

- `set -u` to detect unset variable usage. It will exit the script if a variable is undefined, which helps avoid typos in variable names.
- `set -e` will exit the script if a subprocess fails
- `set -o pipefail` will exit the script if piped subprocess fails

Combine them in `set -euo pipefail` and place the line at the start of the script file.

A helpful tool called Shellcheck[2] can be used to automatically find problems and bugs in shell scripts. You can use it online, on the command line, or use one of the editor integrations.

Debugging

It is possible to instruct Bash to output each command during the execution of a script. `set -x` and `set -v` will both output the line or a block of code before it is executed. While `set -v` outputs the code exactly as it is written, `set -x` outputs commands after they have been expanded (e.g., variables replaced with their values).

Let's see it in a simple script that will print a variable:

script.sh

```
#!/usr/bin/env bash
set -x

# comment

NUM_WORDS="${1:-10}"
echo "${NUM_WORDS}"
```

The output of running such script will be:

Running with set -x

```
> ./script.sh
+ NUM_WORDS=10 ①
+ echo 10
10
```

① The variable was expanded. We can see the default value `10` to be used.

It is also possible to use the `-x` debugging option without modifying the script itself. We can do so by invoking `bash` directly with that option instead (`bash -x ./script.sh`).

While `set -v` offers a bit different output:

Running with set -v

```
> ./script.sh
# comment ①

NUM_WORDS="${1:-10}" ②
echo "${NUM_WORDS}"
10
```

① Comments are printed.

② Code lines are printed as they were written.

Using the `-x` option for the entire script might end up generating very noisy output, but it

is possible to set and unset it only for the lines of code that we are interested in. Unsetting the option is done with `set +x`.

Debugging lines of code selectively

```
#!/usr/bin/env bash
ABC="ABC"

# start debugging
set -x
ZXY="ZXY"

# end debugging
set +x

ABC="ABC"
```

Shell built-in `trap` can be used to set up simple error reporting. It is used to catch signals and execute specific commands when those signals are received. Let's see a basic example of setting up `trap` to print an error in a Bash script:

script.sh

```
#!/usr/bin/env bash
set -euo pipefail

handle_error() { ①
    echo "Error on line $1"
}

trap 'handle_error $LINENO' ERR ②

echo $undefined_var
```

① Although not required, it might be useful to define our own error handler function so that we can execute multiple commands when an error occurs.

② `trap` is instructed to react to the `ERR` signal and call the function `handle_error()`. We could also use `echo` directly here instead if a custom error handler is not required.

When executed, we get a nice error output pointing to the problematic line:

```
> ./script.sh
./script.sh: line 10: undefined_var: unbound variable
```

Final Script Template

```
#!/usr/bin/env bash
set -euo pipefail

handle_error() {
    echo "Error on line $1"
}

trap 'handle_error $LINENO' ERR

if [[ "${TRACE-0}" == "1" ]]; then ①
    set -x
fi

if [[ "${1-}" =~ ^-*h(elp)?$ ]]; then
    echo 'Usage: ./script.sh -h  Shows help'
    echo '    Set TRACE=1 to see the debug output'
    exit
fi

main() { ②
    echo "Script started"
}

main "$@" ③
```

① Let the script users decide if they want to see debug output. With such a check, users can enable tracing with `TRACE=1 ./script.sh` when running the script.

② An optional separate `main` function allows using local variables that don't leak elsewhere.

③ If using the `main` function, call it at the start of the script, passing it all positional arguments that were passed to the script.

Scripts in Other Languages

Programming in Bash is as complex as programming in any other language. It is, therefore, impossible to squeeze it all into this book. However, the main concepts like scripts, functions, basic operations, safer script execution, or debugging should now be

clear.

Many programming languages like Python, Ruby, or Perl can be used to write single-file executable scripts if they are installed on the system. This choice of another language is often desirable as many developers are not familiar with Bash syntax, and the code can become quite unreadable.

To use a different language in a script, we need to change the shebang line to point to the language executable.

```
#!/usr/bin/env python
print("Robots are humans' best friends.")
```

Scheduling

The traditional Linux utility for scheduling jobs is cron[3]. It is a daemon process that runs continuously in the background and executes tasks scheduled using a file called *crontab*. It uses a special syntax for specifying execution times, but utilities like crontab guru[4] help us with that.

Cron is meant to be run on servers that are always powered on. So if a computer is not running at the right time, the scheduled job won't be executed. In this situation, we can use anacron[5], which is suitable for machines that are not always on.

A good use case for anacron is scheduling regular file backups on a personal computer. The *anacrontab* file in `/etc/anacrontab` reveals that some daily and weekly events are already configured to be run:

An example of anacrontab file

```
1   5   cron.daily      nice run-parts /etc/cron.daily ①
7   25  cron.weekly     nice run-parts /etc/cron.weekly
```

① The folder `/etc/cron.daily` contains scripts that will be executed daily. We only need to place our backup script there, and it will automatically run every day. Please note that this configuration is taken from a Fedora Workstation system, so that it might differ from yours.

[1] https://github.com/ko1nksm/getoptions

[2] https://www.shellcheck.net/

[3] https://en.wikipedia.org/wiki/Cron

[4] https://crontab.guru/

[5] https://en.wikipedia.org/wiki/Anacron

Where Next?

If you have made it this far, the next journey will highly depend on your needs. I recommend you check out the manual of your terminal application, the manual of your shell, and scan various GitHub repositories that list modern command-line applications.

You can share your feedback about the book here: https://docs.google.com/forms/d/e/1FAIpQLSfFCpRqP0HADXELZPZYwDK2IDq1xN44vIymQMPZHp6-_LrN9A/viewform?usp=header. This will help me improve the book in the future.

If you are interested in more command line material from me, I occasionally write about the command line in my Software Development Tips[1] newsletter. You can also find me on stribny.name[2].

[1] https://devtips.stribny.name

[2] https://stribny.name