

---

# Shells and Scripting

In this chapter, we'll focus on interacting with Linux on the terminal, that is, via the shell that exposes a command-line interface (CLI). It is vitally important to be able to use the shell effectively to accomplish everyday tasks, and to that end we focus on usability here.

First, we review some terminology and provide a gentle and concise introduction to shell basics. Then we have a look at modern, human-friendly shells, such as the Fish shell. We'll also look at configuration and common tasks in the shell. Then, we move on to the topic of how to effectively work on the CLI using a terminal multiplexer, enabling you to work with multiple sessions, local or remote. In the last part of this chapter, we switch gears and focus on automating tasks in the shell using scripts, including best practices for writing scripts in a safe, secure, and portable manner and also how to lint and test scripts.

There are two major ways to interact with Linux, from a CLI perspective. The first way is manually—that is, a human user sits in front of the terminal, interactively typing commands and consuming the output. This ad-hoc interaction works for most of the things you want to do in the shell on a day-to-day basis, including the following:

- Listing directories, finding files, or looking for content in files
- Copying files between directories or to remote machines
- Reading emails or the news or sending a Tweet from the terminal

Further, we'll learn how to conveniently and efficiently work with multiple shell sessions at the same time.

The other mode of operation is the automated processing of a series of commands in a special kind of file that the shell interprets for you and in turn executes. This mode is usually called *shell scripting* or just *scripting*. You typically want to use a script rather

than manually repeating certain tasks. Also, scripts are the basis of many config and install systems. Scripts are indeed very convenient. However, they can also pose a danger if used without precautions. So, whenever you think about writing a script, keep the XKCD web comic shown in [Figure 3-1](#) in mind.

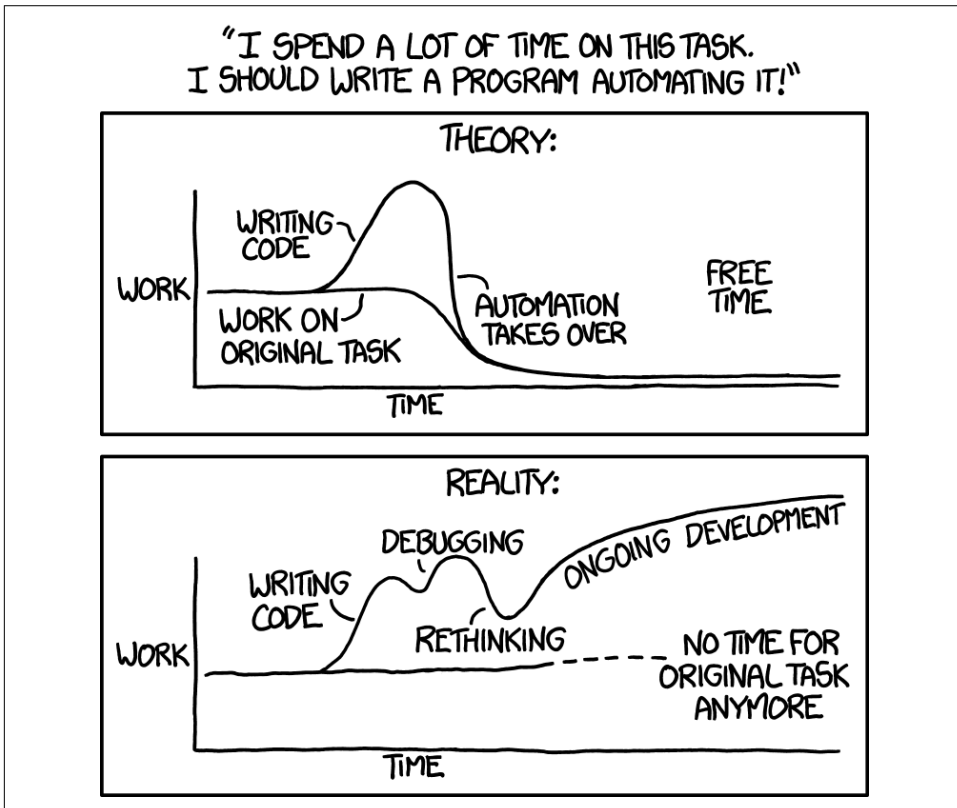


Figure 3-1. XKCD on *automation*. Credit: Randall Munroe (shared under CC BY-NC 2.5 license)

I strongly recommend that you have a Linux environment available and try out the examples shown here right away. With that, are you ready for some (inter)action? If so, then let's start with some terminology and basic shell usage.

## Basics

Before we get into different options and configurations, let's focus on some basic terms such as *terminal* and *shell*. In this section I'll define the terminology and show you how to accomplish everyday tasks in the shell. We'll also review modern commands and see them in action.

## Terminals

We start with the terminal, or terminal emulator, or soft terminal, all of which refer to the same thing: a *terminal* is a program that provides a textual user interface. That is, a terminal supports reading characters from the keyboard and displaying them on the screen. Many years ago, these used to be integrated devices (keyboard and screen together), but nowadays terminals are simply apps.

In addition to the basic character-oriented input and output, terminals support so-called *escape sequences, or escape codes*, for cursor and screen handling and potentially support for colors. For example, pressing Ctrl+H causes a backspace, which deletes the character to the left of the cursor.

The environment variable TERM has the terminal emulator in use, and its configuration is available via `infocmp` as follows (note that the output has been shortened):

```
$ infocmp ❶
#       Reconstructed via infocmp from file: /lib/terminfo/s/screen-256color
screen-256color|GNU Screen with 256 colors,
    am, km, mir, msgr, xenl,
    colors#0x100, cols#80, it#8, lines#24, pairs#0x10000,
    acsc=++\,\,--..00`aaffgghhiijjkkllmmnnooppqrrssttuuvvwxxyyz{{|}}~~,
    bel=^G, blink=\E[5m, bold=\E[1m, cbt=\E[Z, civis=\E[?25l,
    clear=\E[H\E[J, cnorm=\E[34h\E[?25h, cr=\r,
    ...
```

- ❶ The output of `infocmp` is not easy to digest. If you want to learn about the capabilities in detail, consult the **terminfo** database. For example, in my concrete output, the terminal supports 80 columns (`cols#80`) and 24 lines (`lines#24`) for output as well as 256 colors (`colors#0x100`, in hexadecimal notation).

Examples of terminals include not only `xterm`, `rxvt`, and the Gnome terminator but also new generation ones that utilize the GPU, such as **Alacritty**, **kitty**, and **warp**.

In “**Terminal Multiplexer**” on page 55, we will come back to the topic of the terminal.

## Shells

Next up is the *shell*, a program that runs inside the terminal and acts as a command interpreter. The shell offers input and output handling via streams, supports variables, has some built-in commands you can use, deals with command execution and status, and usually supports both interactive usage as well as scripted usage (“**Scripting**” on page 62).

The shell is formally defined in **sh**, and we often come across the term **POSIX shell**, which will become more important in the context of scripts and portability.

Originally, we had the Bourne shell `sh`, named after the author, but nowadays it's usually replaced with the **bash** shell—a wordplay on the original version, short for “Bourne Again Shell”—which is widely used as the default.

If you are curious about what you're using, use the `file -h /bin/sh` command to find out, or if that fails, try `echo $0` or `echo $SHELL`.



In this section, we assume the bash shell (`bash`), unless we call it out explicitly.

There are many more implementations of `sh` as well as other variants, such as the Korn shell, `ksh`, and C shell, `csh`, which are not widely used today. We will, however, review modern bash replacements in “**Human-Friendly Shells**” on page 48.

Let's start our shell basics with two fundamental features: streams and variables.

## Streams

Let's start with the topic of input (streams) and output (streams), or I/O for short. How can you feed a program some input? How do you control where the output of a program lands, say, on the terminal or in a file?

First off, the shell equips every process with three default file descriptors (FDs) for input and output:

- `stdin` (FD 0)
- `stdout` (FD 1)
- `stderr` (FD 2)

These FDs are, as depicted in **Figure 3-2**, by default connected to your screen and keyboard, respectively. In other words, unless you specify something else, a command you enter in the shell will take its input (`stdin`) from your keyboard, and it will deliver its output (`stdout`) to your screen.

The following shell interaction demonstrates this default behavior:

```
$ cat
This is some input I type on the keyboard and read on the screen^C
```

In the preceding example using `cat`, you see the defaults in action. Note that I used `Ctrl+C` (shown as `^C`) to terminate the command.

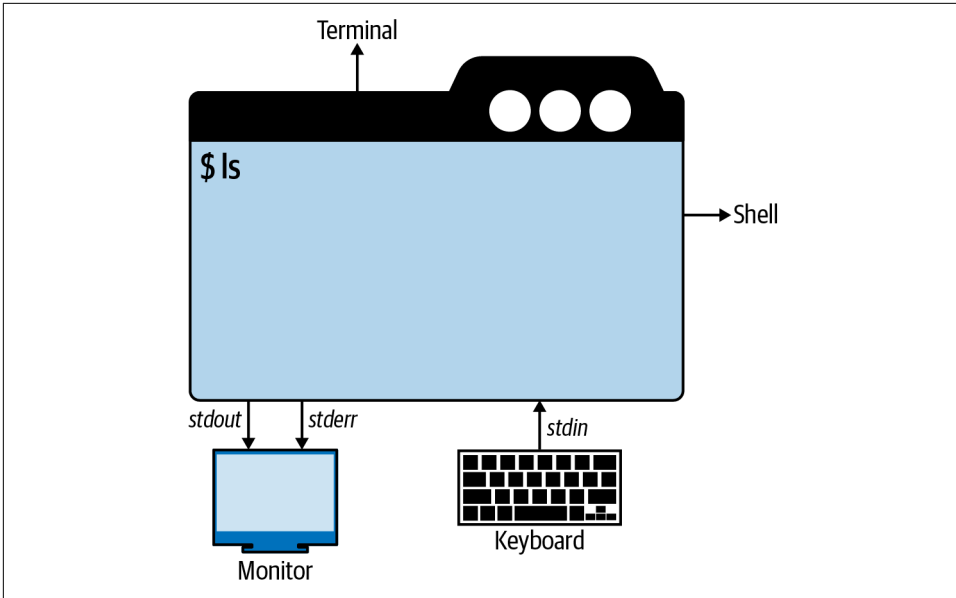


Figure 3-2. Shell I/O default streams

If you don't want to use the defaults the shell gives you—for example, you don't want `stderr` to be outputted on the screen but want to save it in a file—you can **redirect** the streams.

You redirect the output stream of a process using `$FD>` and `<$FD`, with `$FD` being the file descriptor—for example, `2>` means redirect the `stderr` stream. Note that `1>` and `>` are the same since `stdout` is the default. If you want to redirect both `stdout` and `stderr`, use `&>`, and when you want to get rid of a stream, you can use `/dev/null`.

Let's see how that works in the context of a concrete example, downloading some HTML content via `curl`:

```
$ curl https://example.com &> /dev/null ❶

$ curl https://example.com > /tmp/content.txt 2> /tmp/curl-status ❷
$ head -3 /tmp/content.txt
<!doctype html>
<html>
<head>
$ cat /tmp/curl-status
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left   Speed
100 1256 100 1256    0    0  3187      0 --:--:-- --:--:-- --:--:-- 3195

$ cat > /tmp/interactive-input.txt ❸
```

```
$ tr < /tmp/curl-status [A-Z] [a-z] ④
% total    % received % xferd  average speed  time    time    time  current
          dload  upload  total  spent   left   speed
100  1256  100  1256    0    0  3187    0  --:--:-- --:--:-- --:--:-- 3195
```

- ❶ Discard all output by redirecting both `stdout` and `stderr` to `/dev/null`.
- ❷ Redirect the output and status to different files.
- ❸ Interactively enter input and save to file; use `Ctrl+D` to stop capturing and store the content.
- ❹ Lowercase all words, using the `tr` command that reads from `stdin`.

Shells usually understand a number of special characters, such as:

#### *Ampersand (&)*

Placed at the end of a command, executes the command in the background (see also “[Job control](#)” on page 40)

#### *Backslash (\)*

Used to continue a command on the next line, for better readability of long commands

#### *Pipe (|)*

Connects `stdout` of one process with the `stdin` of the next process, allowing you to pass data without having to store it in files as a temporary place

## Pipes and the UNIX Philosophy

While **pipes** might seem not too exciting at first glance, there’s much more to them. I once had a nice interaction with Doug McIlroy, the inventor of pipes. I wrote an article, “[Revisiting the Unix Philosophy in 2018](#)”, in which I drew parallels between UNIX and microservices. Someone commented on the article, and that comment led to Doug sending me an email (very unexpectedly, and I had to verify to believe it) to clarify things.

Again, let’s see some of the theoretical content in action. Let’s try to figure out how many lines an HTML file contains by downloading it using `curl` and then piping the content to the `wc` tool:

```
$ curl https://example.com 2> /dev/null | \ ❶
wc -l ❷
46
```

- ❶ Use `curl` to download the content from the URL, and discard the status that it outputs on `stderr`. (Note: in practice, you'd use the `-s` option of `curl`, but we want to learn how to apply our hard-gained knowledge, right?)
- ❷ The `stdout` of `curl` is fed to `stdin` of `wc`, which counts the number of lines with the `-l` option.

Now that you have a basic understanding of commands, streams, and redirection, let's move on to another core shell feature, the handling of variables.

## Variables

A term you will come across often in the context of shells is *variables*. Whenever you don't want to or cannot hardcode a value, you can use a variable to store and change a value. Use cases include the following:

- When you want to handle configuration items that Linux exposes—for example, the place where the shell looks for executables captured in the `$PATH` variable. This is kind of an interface where a variable might be read/write.
- When you want to interactively query the user for a value, say, in the context of a script.
- When you want to shorten input by defining a long value once—for example, the URL of an HTTP API. This use case roughly corresponds to a `const` value in a program language since you don't change the value after you have declared the variable.

We distinguish between two kinds of variables:

### *Environment variables*

Shell-wide settings; list them with `env`.

### *Shell variables*

Valid in the context of the current execution; list with `set` in `bash`. Shell variables are not inherited by subprocesses.

You can, in `bash`, use `export` to create an environment variable. When you want to access the value of a variable, put a `$` in front of it, and when you want to get rid of it, use `unset`.

OK, that was a lot of information. Let's see how that looks in practice (in `bash`):

```
$ set MY_VAR=42 ❶
$ set | grep MY_VAR ❷
_=MY_VAR=42
$ export MY_GLOBAL_VAR="fun with vars" ❸
```

```

$ set | grep 'MY_*' ❹
MY_GLOBAL_VAR='fun with vars'
_=MY_VAR=42

$ env | grep 'MY_*' ❺
MY_GLOBAL_VAR=fun with vars

$ bash ❻
$ echo $MY_GLOBAL_VAR ❼
fun with vars

$ set | grep 'MY_*' ❽
MY_GLOBAL_VAR='fun with vars'

$ exit ❾
$ unset $MY_VAR
$ set | grep 'MY_*'
MY_GLOBAL_VAR='fun with vars'

```

- ❶ Create a shell variable called `MY_VAR`, and assign a value of 42.
- ❷ List shell variables and filter out `MY_VAR`. Note the `_`, indicating it's not exported.
- ❸ Create a new environment variable called `MY_GLOBAL_VAR`.
- ❹ List shell variables and filter out all that start with `MY_`. We see, as expected, both of the variables we created in the previous steps.
- ❺ List environment variables. We see `MY_GLOBAL_VAR`, as we would hope.
- ❻ Create a new shell session—that is, a child process of the current shell session that doesn't inherit `MY_VAR`.
- ❼ Access the environment variable `MY_GLOBAL_VAR`.
- ❽ List the shell variables, which gives us only `MY_GLOBAL_VAR` since we're in a child process.
- ❾ Exit the child process, remove the `MY_VAR` shell variable, and list our shell variables. As expected, `MY_VAR` is gone.



In [Table 3-1](#) I put together common shell and environment variables. You will find those variables almost everywhere, and they are important to understand and to use. For any of the variables, you can have a look at the respective value using `echo $XXX`, with `XXX` being the variable name.

*Table 3-1. Common shell and environment variables*

Variable	Type	Semantics
EDITOR	Environment	The path to program used by default to edit files
HOME	POSIX	The path of the home directory of the current user
HOSTNAME	bash shell	The name of the current host
IFS	POSIX	List of characters to separate fields; used when the shell splits words on expansion
PATH	POSIX	Contains a list of directories in which the shell looks for executable programs (binaries or scripts)
PS1	Environment	The primary prompt string in use
PWD	Environment	The full path of the working directory
OLDPWD	bash shell	The full path of the directory before the last <code>cd</code> command
RANDOM	bash shell	A random integer between 0 and 32767
SHELL	Environment	Contains the currently used shell
TERM	Environment	The terminal emulator used
UID	Environment	Current user unique ID (integer value)
USER	Environment	Current user name
_	bash shell	Last argument to the previous command executed in the foreground
?	bash shell	Exit status; see <a href="#">“Exit status” on page 39</a>
\$	bash shell	The ID of the current process (integer value)
0	bash shell	The name of the current process

Further, check out the full list of [bash-specific variables](#), and also note that the variables from [Table 3-1](#) will come in handy again in the context of [“Scripting” on page 62](#).

### Exit status

The shell communicates the completion of a command execution to the caller using what is called the *exit status*. In general, it is expected that a Linux command returns a status when it terminates. This can either be a normal termination (happy path) or an abnormal termination (something went wrong). A `0` exit status means that the command was successfully run, without any errors, whereas a nonzero value between 1 and 255 signals a failure. To query the exit status, use `echo $?`.

Be careful with exit status handling in a pipeline, since some shells make only the last status available. You can work around that limitation **by using \$PIPESTATUS**.

## Built-in commands

Shells come with a number of built-in commands. Some useful examples are `yes`, `echo`, `cat`, or `read` (depending on the Linux distro, some of those commands might not be built-ins but located in `/usr/bin`). You can use the `help` command to list built-ins. Do remember, however, that everything else is a shell-external program that you usually can find in `/usr/bin` (for user commands) or in `/usr/sbin` (for administrative commands).

How do you know where to find an executable? Here are some ways:

```
$ which ls
/usr/bin/ls

$ type ls
ls is aliased to `ls --color=auto`
```



One of the technical reviewers of this book rightfully pointed out that `which` is a non-POSIX, external program that may not always be available. Also, they suggested using `command -v` rather than `which` to get the program path and or shell alias/function. See also the [shellcheck docs](#) for further details on the matter.

## Job control

A feature most shells support is called *job control*. By default, when you enter a command, it takes control of the screen and the keyboard, which we usually call *running in the foreground*. But what if you don't want to run something interactively, or, in case of a server, what if there is no input from `stdin` at all? Enter job control and background jobs: to launch a process in the background, put an `&` at the end, or to send a foreground process to the background, press `Ctrl+Z`.

The following example shows this in action, giving you a rough idea:

```
$ watch -n 5 "ls" & ❶

$ jobs ❷
Job      Group    CPU      State    Command
❶        3021     0%       stopped watch -n 5 "ls" &

$ fg ❸
Every 5.0s: ls                                     Sat Aug 28 11:34:32 2021

Dockerfile
app.yaml
example.json
```

```
main.go
script.sh
test
```

- ❶ By putting the `&` at the end, we launch the command in the background.
- ❷ List all jobs.
- ❸ With the `fg` command, we can bring a process to the foreground. If you want to quit the `watch` command, use `Ctrl+C`.

If you want to keep a background process running, even after you close the shell you can prepend the `nohup` command. Further, for a process that is already running and wasn't prepended with `nohup`, you can use `disown` after the fact to achieve the same effect. Finally, if you want to get rid of a running process, you can use the `kill` command with various levels of forcefulness (see [“Signals” on page 214](#) for more details).

Rather than job control, I recommend using terminal multiplexer, as discussed in [“Terminal Multiplexer” on page 55](#). These programs take care of the most common use cases (shell closes, multiple processes running and need coordination, etc.) and also support working with remote systems.

Let's move on to discuss modern replacements for frequently used core commands that have been around forever.

## Modern Commands

There are a handful of commands you will find yourself using over and over again on a daily basis. These include commands for navigating directories (`cd`), listing the content of a directory (`ls`), finding files (`find`), and displaying the content of files (`cat`, `less`). Given that you are using these commands so often, you want to be as efficient as possible—every keystroke counts.

Modern variations exist for some of these often-used commands. Some of them are drop-in replacements, and others extend the functionality. All of them offer somewhat sane default values for common operations and rich output that is generally easier to comprehend, and they usually lead to you typing less to accomplish the same task. This reduces the friction when you work with the shell, making it more enjoyable and improving the flow. If you want to learn more about modern tooling, check out [Appendix B](#). In this context, a word of caution, especially if you're applying this knowledge in an enterprise environment: I have no stake in any of these tools and purely recommend them because I have found them useful myself. A good way to go about installing and using any of these tools is to use a version of the tool that has been vetted by your Linux distro of choice.

## Listing directory contents with **exa**

Whenever you want to know what a directory contains, you use **ls** or one of its variants with parameters. For example, in bash I used to have **l** aliased to **ls -GAhltr**. But there's a better way: **exa**, a modern replacement for **ls**, written in Rust, with built-in support for Git and tree rendering. In this context, what would you guess is the most often used command after you've listed the directory content? In my experience it's to clear the screen, and very often people use **clear**. That's typing five characters and then hitting ENTER. You can have the same effect much faster—simply use Ctrl+L.

## Viewing file contents with **bat**

Let's assume that you listed a directory's contents and found a file you want to inspect. You'd use **cat**, maybe? There's something better I recommend you have a look at: **bat**. The **bat** command, shown in [Figure 3-3](#), comes with syntax highlighting, shows non-printable characters, supports Git, and has an integrated pager (the page-wise viewing of files longer than what can be displayed on the screen).

## Finding content in files with **rg**

Traditionally, you would use **grep** to find something in a file. However, there's a modern command, **rg**, that is fast and powerful.

We're going to compare **rg** to a **find** and **grep** combination in this example, where we want to find YAML files that contain the string "sample":

```
$ find . -type f -name "*.yaml" -exec grep "sample" '{}' \; -print ❶
    app: sample
    app: sample
./app.yaml

$ rg -t "yaml" sample ❷
app.yaml
9:      app: sample
14:      app: sample
```

❶ Use **find** and **grep** together to find a string in YAML files.

❷ Use **rg** for the same task.

If you compare the commands and the results in the previous example, you see that not only is **rg** easier to use but the results are more informative (providing context, in this case the line number).

	File: main.go
1	package main
2	
3	import (
4	"fmt"
5	"net/http"
6	)
7	
8	func main() {
9	http.HandleFunc("/", HelloServer)
10	http.ListenAndServe(":8080", nil)
11	}
12	
13	func HelloServer(w http.ResponseWriter, r *http.Request) {
14	fmt.Fprintf(w, "Hello, %s!", r.URL.Path[1:])
15	}
	File: app.yaml
1	apiVersion: apps/v1
2	kind: Deployment
3	metadata:
4	name: something
5 +	namespace: xample
6	spec:
7	selector:
8	matchLabels:
9	app: sample
10 ~	replicas: 2
11	template:
12	metadata:
13	labels:
14	app: sample
15	spec:
16	containers:
17	- name: example
18 -	image: public.ecr.aws/mhausenblas/example:stable

Figure 3-3. Rendering of a Go file (top) and a YAML file (bottom) by bat

## JSON data processing with jq

And now for a bonus command. This one, `jq`, is not an actual replacement but more like a specialized tool for JSON, a popular textual data format. You find JSON in HTTP APIs and configuration files alike.

So, use `jq` rather than `awk` or `sed` to pick out certain values. For example, by using a [JSON generator](#) to generate some random data, I have a 2.4 kB JSON file *example.json* that looks something like this (only showing the first record here):

```
[
  {
    "_id": "612297a64a057a3fa3a56fcf",
    "latitude": -25.750679,
    "longitude": 130.044327,
    "friends": [
      {
        "id": 0,
        "name": "Tara Holland"
      },
      {
        "id": 1,
        "name": "Giles Glover"
      },
      {
        "id": 2,
        "name": "Pennington Shannon"
      }
    ],
    "favoriteFruit": "strawberry"
  },
  ...
]
```

Let's say we're interested in all "first" friends—that is, entry 0 in the `friends` array—of people whose favorite fruit is "strawberry." With `jq` you would do the following:

```
$ jq 'select(.[].favoriteFruit=="strawberry") | .[].friends[0].name' example.json
"Tara Holland"
"Christy Mullins"
"Snider Thornton"
"Jana Clay"
"Wilma King"
```

That was some CLI fun, right? If you're interested in finding out more about the topic of modern commands and what other candidates there might be for you to replace, check out the [modern-unix repo](#), which lists suggestions. Let's now move our focus to some common tasks beyond directory navigation and file content viewing and how to go about them.

# Common Tasks

There are a number of things you likely find yourself doing often, and there are certain tricks you can use to speed up your tasks in the shell. Let’s review these common tasks and see how we can be more efficient.

## Shorten often-used commands

One fundamental insight with interfaces is that commands that you are using very often should take the least effort—they should be quick to enter. Now apply this idea to the shell: rather than `git diff --color-moved`, I type `d` (a single character), since I’m viewing changes in my repositories many hundreds of times per day. Depending on the shell, there are different ways to achieve this: in `bash` this is called an *alias*, and in `Fish` (“[Fish Shell](#)” on page 49) there are *abbreviations* you can use.

## Navigating

When you enter commands on the shell prompt, there are a number of things you might want to do, such as navigating the line (for example, moving the cursor to the start) or manipulating the line (say, deleting everything left of the cursor). [Table 3-2](#) lists common shell shortcuts.

Table 3-2. Shell navigation and editing shortcuts

Action	Command	Note
Move cursor to start of line	Ctrl+a	-
Move cursor to end of line	Ctrl+e	-
Move cursor forward one character	Ctrl+f	-
Move cursor back one character	Ctrl+b	-
Move cursor forward one word	Alt+f	Works only with left Alt
Move cursor back one word	Alt+b	-
Delete current character	Ctrl+d	-
Delete character left of cursor	Ctrl+h	-
Delete word left of cursor	Ctrl+w	-
Delete everything right of cursor	Ctrl+k	-
Delete everything left of cursor	Ctrl+u	-
Clear screen	Ctrl+l	-
Cancel command	Ctrl+c	-
Undo	Ctrl+_	bash only
Search history	Ctrl+r	Some shells
Cancel search	Ctrl+g	Some shells

Note that not all shortcuts may be supported in all shells, and certain actions such as history management may be implemented differently in certain shells. In addition, you might want to know that these shortcuts are based on Emacs editing keystrokes. Should you prefer `vi`, you can use `set -o vi` in your `.bashrc` file, for example, to perform command-line editing based on `vi` keystrokes. Finally, taking [Table 3-2](#) as a starting point, try out what your shell supports and see how you can configure it to suit your needs.

## File content management

You don't always want to fire up an editor such as `vi` to add a single line of text. And sometimes you can't do it—for example, in the context of writing a shell script (“Scripting” on page 62).

So, how can you manipulate textual content? Let's have a look at a few examples:

```
$ echo "First line" > /tmp/something ❶

$ cat /tmp/something ❷
First line

$ echo "Second line" >> /tmp/something && \ ❸
  cat /tmp/something
First line
Second line

$ sed 's/line/LINE/' /tmp/something ❹
First LINE
Second LINE

$ cat << 'EOF' > /tmp/another ❺
First line
Second line
Third line
EOF

$ diff -y /tmp/something /tmp/another ❻
First line
Second line
                                     First line
                                     Second line
> Third line
```

- ❶ Create a file by redirecting the echo output.
- ❷ View content of file.
- ❸ Append a line to file using the `>>` operator and then view content.
- ❹ Replace content from file using `sed` and output to `stdout`.



- ❺ Create a file using the [here document](#).
- ❻ Show differences between the files we created.

Now that you know the basic file content manipulation techniques, let's have a look at the advanced viewing of file contents.

## Viewing long files

For long files—that is, files that have more lines than the shell can display on your screen—you can use pagers like `less` or `bat` (`bat` comes with a built-in pager). With paging, a program splits the output into pages where each page fits into what the screen can display and some commands to navigate the pages (view next page, previous page, etc.).

Another way to deal with long files is to display only a select region of the file, like the first few lines. There are two handy commands for this: `head` and `tail`.

For example, to display the beginning of a file:

```
$ for i in {1..100} ; do echo $i >> /tmp/longfile ; done ❶  
  
$ head -5 /tmp/longfile ❷  
1  
2  
3  
4  
5
```

- ❶ Create a long file (100 lines here).
- ❷ Display the first five lines of the long file.

Or, to get live updates of a file that is constantly growing, we could use:

```
$ sudo tail -f /var/log/Xorg.0.log ❶  
[ 36065.898] (II) event14 - ALPS01:00 0911:5288 Mouse: device is a pointer  
[ 36065.900] (II) event15 - ALPS01:00 0911:5288 Touchpad: device is a touchpad  
[ 36065.901] (II) event4 - Intel HID events: is tagged by udev as: Keyboard  
[ 36065.901] (II) event4 - Intel HID events: device is a keyboard  
...
```

- ❶ Display the end of a log file using `tail`, with the `-f` option meaning to follow, or to update automatically.

Lastly, in this section we look at dealing with date and time.

## Date and time handling

The `date` command can be a useful way to generate unique file names. It allows you to generate dates in various formats, including the **Unix time stamp**, as well as to convert between different date and time formats.

```
$ date +%s ❶  
1629582883
```

```
$ date -d @1629742883 '+%m/%d/%Y:%H:%M:%S' ❷  
08/21/2021:21:54:43
```

- ❶ Create a UNIX time stamp.
- ❷ Convert a UNIX time stamp to a human-readable date.

### On the UNIX Epoch Time

The UNIX epoch time (or simply UNIX time) is the number of seconds elapsed since 1970-01-01T00:00:00Z. UNIX time treats every day as exactly 86,400 seconds long.

If you're dealing with software that stores UNIX time as a signed 32-bit integer, you might want to pay attention since this will cause issues on 2038-01-19, as then the counter will overflow, which is also known as the **Year 2038 problem**.

You can use **online converters** for more advanced operations, supporting microseconds and milliseconds resolutions.

With that we wrap up the shell basics section. By now you should have a good understanding of what terminals and shells are and how to use them to do basic tasks such as navigating the filesystem, finding files, and more. We now move on to the topic of human-friendly shells.

## Human-Friendly Shells

While the **bash shell** is likely still the most widely used shell, it is not necessarily the most human-friendly one. It has been around since the late 1980s, and its age sometimes shows. There are a number of modern, human-friendly shells I strongly recommend you evaluate and use instead of bash.

We'll first examine in detail one concrete example of a modern, human-friendly shell called the Fish shell and then briefly discuss others, just to make sure you have an idea about the range of choices. We wrap up this section with a quick recommendation and conclusion in **"Which Shell Should I Use?" on page 55**.

# Fish Shell

The **Fish shell** describes itself as a smart and user-friendly command-line shell. Let's have a look at some basic usage first and then move on to configuration topics.

## Basic usage


For many daily tasks, you won't notice a big difference from `bash` in terms of input; most of the commands provided in [Table 3-2](#) are valid. However, there are two areas where `fish` is different from and much more convenient than `bash`:

*There is no explicit history management.*

You simply type and you get previous executions of a command shown. You can use the up and down key to select one (see [Figure 3-4](#)).

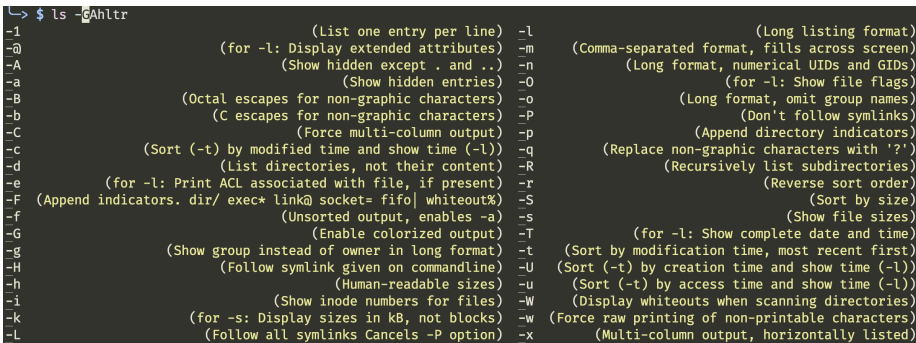
*Autosuggestions are available for many commands.*

This is shown in [Figure 3-5](#). In addition, when you press `Tab`, the Fish shell will try to complete the command, argument, or path, giving you visual hints such as coloring your input red if it doesn't recognize the command.



```
↳ $ exa -long --all --git
app.yaml Dockerfile example.json main.go script.sh test
```

Figure 3-4. Fish history handling in action



```
↳ $ ls -lAhltr
-l          (List one entry per line) -l          (Long listing format)
-@          (for -l: Display extended attributes) -m          (Comma-separated format, fills across screen)
-A          (Show hidden except . and ..) -n          (Long format, numerical UIDs and GIDs)
-a          (Show hidden entries) -O          (for -l: Show file flags)
-B          (Octal escapes for non-graphic characters) -o          (Long format, omit group names)
-b          (C escapes for non-graphic characters) -P          (Don't follow symlinks)
-C          (Force multi-column output) -p          (Append directory indicators)
-c          (Sort (-t) by modified time and show time (-l)) -q          (Replace non-graphic characters with '?')
-d          (List directories, not their content) -R          (Recursively list subdirectories)
-e          (for -l: Print ACL associated with file, if present) -r          (Reverse sort order)
-F          (Append indicators. dir/ exec* link@ socket= fifo| whiteout%) -S          (Sort by size)
-f          (Unsorted output, enables -a) -s          (Show file sizes)
-g          (Enable colorized output) -T          (for -l: Show complete date and time)
-g          (Show group instead of owner in long format) -t          (Sort by modification time, most recent first)
-H          (Follow symlink given on commandline) -U          (Sort (-t) by creation time and show time (-l))
-h          (Human-readable sizes) -u          (Sort (-t) by access time and show time (-l))
-i          (Show inode numbers for files) -W          (Display whiteouts when scanning directories)
-k          (for -s: Display sizes in kB, not blocks) -w          (Force raw printing of non-printable characters)
-L          (Follow all symlinks Cancels -P option) -x          (Multi-column output, horizontally listed)
```

Figure 3-5. Fish autosuggestion in action

Table 3-3 lists some common fish commands. In this context, note specifically the handling of environment variables.

Table 3-3. Fish shell reference

Task	Command
Export environment variable KEY with value VAL	set -x KEY VAL
Delete environment variable KEY	set -e KEY
Inline env var KEY for command cmd	env KEY=VAL cmd
Change path length to 1	set -g fish_prompt_pwd_dir_length 1
Manage abbreviations	abbr
Manage functions	functions and funcnd

Unlike other shells, fish stores the exit status of the last command in a variable called `$status` instead of in  `$?` .

If you’re coming from bash, you may also want to consult the [Fish FAQ](#), which addresses most of the gotchas.

### Configuration

To **configure the Fish shell**, you simply enter the `fish_config` command (you might need to add the `browse` subcommand, depending on your distro), and fish will launch a server via <http://localhost:8000> and automatically open your default browser with a fancy UI, shown in [Figure 3-6](#), which allows you to view and change settings.

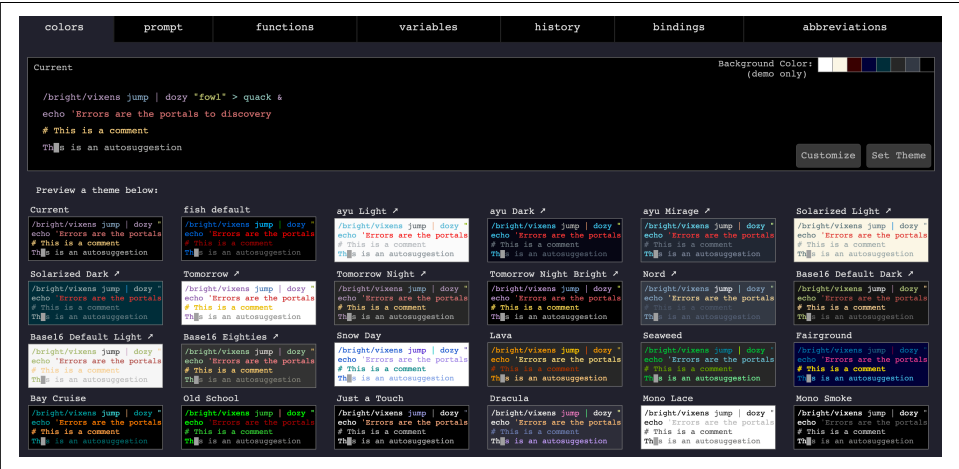


Figure 3-6. Fish shell configuration via browser



To switch between vi and Emacs (default) key bindings for command-line navigation, use the `fish_vi_key_bindings` to start vi mode, and use `fish_default_key_bindings` to reset it to Emacs. Note that the changes will take place in all active shell sessions immediately.

Let's now see how I have configured my environment. In fact, my config is rather short; in `config.fish` I have the following:

```
set -x FZF_DEFAULT_OPTS "-m --bind='ctrl-o:execute(nvim {})+abort'"
set -x FZF_DEFAULT_COMMAND 'rg --files'
set -x EDITOR nvim
set -x KUBE_EDITOR nvim
set -ga fish_user_paths /usr/local/bin
```

My prompt, defined in `fish_prompt.fish`, looks as follows:

```
function fish_prompt
    set -l retc red
    test $status = 0; and set retc blue

    set -q __fish_git_prompt_showupstream
    or set -g __fish_git_prompt_showupstream auto

    function _nim_prompt_wrapper
        set retc $argv[1]
        set field_name $argv[2]
        set field_value $argv[3]

        set_color normal
        set_color $retc
        echo -n '-'
        set_color -o blue
        echo -n '['
        set_color normal
        test -n $field_name
        and echo -n $field_name:
        set_color $retc
        echo -n $field_value
        set_color -o blue
        echo -n ']'
    end

    set_color $retc
    echo -n 'T'
    set_color -o blue
    echo -n [
    set_color normal
    set_color c07933
    echo -n (prompt_pwd)
    set_color -o blue
    echo -n ']'
end
```

```

# Virtual Environment
set -q VIRTUAL_ENV_DISABLE_PROMPT
or set -g VIRTUAL_ENV_DISABLE_PROMPT true
set -q VIRTUAL_ENV
and _nin_prompt_wrapper $retc V (basename "$VIRTUAL_ENV")

# git
set prompt_git (fish_git_prompt | string trim -c '()')
test -n "$prompt_git"
and _nin_prompt_wrapper $retc G $prompt_git

# New line
echo

# Background jobs
set_color normal
for job in (jobs)
    set_color $retc
    echo -n '| '
    set_color brown
    echo $job
end
set_color blue
echo -n '↪ '
    set_color -o blue
echo -n '$ '
set_color normal
end

```

The preceding prompt definition yields the prompt shown in [Figure 3-7](#); note the difference between a directory that contains a Git repo and one that does not, a built-in visual cue to speed up your flow. Also, notice the current time on the righthand side.



Figure 3-7. Fish shell prompt

My abbreviations—think of these as alias replacements, as found in other shells—look as follows:

```

$ abbr
abbr -a -U -- :q exit
abbr -a -U -- cat bat
abbr -a -U -- d 'git diff --color-moved'
abbr -a -U -- g git
abbr -a -U -- grep 'grep --color=auto'
abbr -a -U -- k kubectl

```

```
abbr -a -U -- l 'exa --long --all --git'
abbr -a -U -- ll 'ls -GAhltr'
abbr -a -U -- m make
abbr -a -U -- p 'git push'
abbr -a -U -- pu 'git pull'
abbr -a -U -- s 'git status'
abbr -a -U -- stat 'stat -x'
abbr -a -U -- vi nvim
abbr -a -U -- wget 'wget -c'
```

To add a new abbreviation, use `abbr --add`. Abbreviations are handy for simple commands that take no arguments. What if you have a more complicated construct you want to shorten? Say you want to shorten a sequence involving `git` that also takes an argument. Meet functions in Fish.

Let's now take a look at an example function, which is defined in the file named *c.fish*. We can use the `functions` command to list all defined functions, the `function` command to create a new function, and in this case the command `function c` to edit it as follows:

```
function c
    git add --all
    git commit -m "$argv"
end
```

With that we have reached the end of the Fish section, in which we walked through a usage tutorial and configuration tips. Now let's have a quick look at other modern shells.

## Z-shell

**Z-shell**, or `zsh`, is a Bourne-like shell with a powerful **completion** system and rich theming support. With **Oh My Zsh**, you can pretty much configure and use `zsh` in the way you've seen earlier on with `fish` while retaining wide backward compatibility with `bash`.

`zsh` uses five startup files, as shown in the following example (note that if `$ZDOTDIR` is not set, `zsh` uses `$HOME` instead):

```
$ZDOTDIR/.zshenv ❶
$ZDOTDIR/.zprofile ❷
$ZDOTDIR/.zshrc ❸
$ZDOTDIR/.zlogin ❹
$ZDOTDIR/.zlogout ❺
```

- ❶ Sourced on all invocations of the shell. It should contain commands to set the search path, plus other important environment variables. But it should not contain commands that produce output or assume the shell is attached to a `tty`.

- ② Meant as an alternative to *.zlogin* for ksh fans (these two are not intended to be used together); similar to *.zlogin*, except that it is sourced before *.zshrc*.
- ③ Sourced in interactive shells. It should contain commands to set up aliases, functions, options, key bindings, and so on.
- ④ Sourced in login shells. It should contain commands that should be executed only in login shells. Note that *.zlogin* is not the place for alias definitions, options, environment variable settings, and the like.
- ⑤ Sourced when login shells exit.

For more zsh plug-ins, see also the [awesome-zsh-plugins repo on GitHub](#). If you want to learn zsh, consider reading “[An Introduction to the Z Shell](#)” by Paul Falstad and Bas de Bakker.

## Other Modern Shells

In addition to *fish* and *zsh*, there are a number of other interesting—but not necessarily always bash-compatible—shells available out there. When you have a look at those, ask yourself what the focus of the respective shell is (interactive usage vs. scripting) and how active the community around it is.

Some examples of modern shells for Linux I came across and can recommend you have a look at include the following:

### *Oil shell*

Targets Python and JavaScript users. Put in other words, the focus is less on interactive use but more on scripting.

### *murex*

A POSIX shell that sports interesting features such as an integrated testing framework, typed pipelines, and event-driven programming.

### *Nushell*

An experimental new shell paradigm, featuring tabular output with a powerful query language. Learn more via the detailed [Nu Book](#).

### *PowerShell*

A cross-platform shell that started off as a fork of the Windows PowerShell and offers a different set of semantics and interactions than POSIX shells.

There are many more options out there. Keep looking and see what works best for you. Try thinking beyond bash and optimize for your use case.



## Which Shell Should I Use?

At this point in time, every modern shell—other than bash—seems like a good choice, from a human-centric perspective. Smooth auto-complete, easy config, and smart environments are no luxury in 2022, and given the time you usually spend on the command line, you should try out different shells and pick the one you like most. I personally use the Fish shell, but many of my peers are super happy with the Z-shell.

You may have issues that make you hesitant to move away from bash, such as the following:

- You work in remote systems and/or cannot install your own shell.
- You’ve stayed with bash due to compatibility and/or muscle memory. It can be hard to get rid of certain habits.
- Almost all instructions (implicitly) assume bash. For example, you’ll see instructions like `export FOO=BAR`, which is bash specific.

It turns out that these issues are by and large not relevant to most users. While you may have to temporarily use bash in a remote system, most of the time you will be working in an environment that you control. There is a learning curve, but the investment pays off in the long run.

With that, let’s focus on another way to boost your productivity in the terminal: multiplexer.

## Terminal Multiplexer

We came across terminals at the beginning of this chapter, in [“Terminals” on page 33](#). Now let’s dive deeper into the topic of how to improve your terminal usage, building on a concept that is both simple and powerful: multiplexing.

Think of it in this way: you usually work on different things that can be grouped together. For example, you may work on an open source project, author a blog post or docs, access a server remotely, interact with an HTTP API to test things, and so forth. These tasks may each require one or more terminal windows, and often you want or need to do potentially interdependent tasks in two windows at the same time. For example:

- You are using the `watch` command to periodically execute a directory listing and at the same time edit a file.
- You start a server process (a web server or application server) and want to have it running in the foreground (see also [“Job control” on page 40](#)) to keep an eye on the logs.

- You want to edit a file using `vi` and at the same time use `git` to query the status and commit changes.
- You have a VM running in the public cloud and want to `ssh` into it while having the possibility to manage files locally.

Think of all these examples as things that logically belong together and that in terms of time duration can range from short term (a few minutes) to long term (days and weeks). The grouping of those tasks is usually called a *session*.

Now, there are a number of challenges if you want to achieve this grouping:

- You need multiple windows, so one solution is to launch multiple terminals or, if the UI supports it, multiple instances (tabs).
- You would like to have all the windows and paths around, even if you close the terminal or the remote side closes down.
- You want to expand or zoom in and out to focus on certain tasks while keeping an overview of all your sessions and being able to navigate between them.

To enable these tasks, people came up with the idea of overlaying a terminal with multiple windows (and sessions, to group windows)—in other words, multiplexing the terminal I/O.

Let's have a brief look at the original implementation of terminal multiplexing, called `screen`. Then we'll focus in-depth on a widely used implement called `tmux` and wrap up with other options in this space.

## screen

`screen` is the original terminal multiplexer and is still used. Unless you're in a remote environment where nothing else is available and/or you can't install another multiplexer, you should probably not be using `screen`. One reason is that it's not actively maintained anymore, and another is that it's not very flexible and lacks a number of features modern terminal multiplexers have.

## tmux

`tmux` is a flexible and rich terminal multiplexer that you can bend to your needs. As you can see in [Figure 3-8](#), there are three core elements you're interacting with in `tmux`, from coarse-grained to fine-grained units:

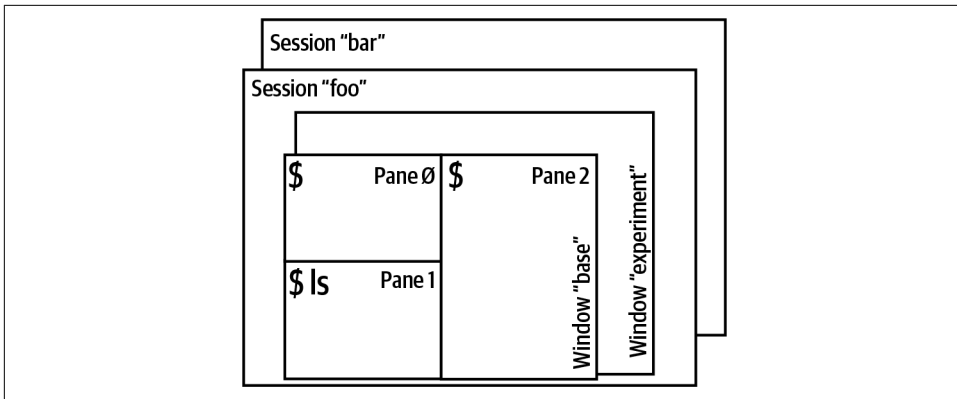


Figure 3-8. The *tmux* elements: sessions, windows, and panes

### Sessions

A logical unit that you can think of as a working environment dedicated to a specific task such as “working on project X” or “writing blog post Y.” It’s the container for all other units.

### Windows

You can think of a window as a tab in a browser, belonging to a session. It’s optional to use, and often you only have one window per session.

### Panes

These are your workhorses, effectively a single shell instance running. A pane is part of a window, and you can easily split it vertically or horizontally, as well as expand/collapse it (think: zoom) and close panes as you need them.

Just like *screen*, in *tmux* you have the ability to attach and detach a session. Let’s assume we start from scratch, let’s launch it with a session called *test*:

```
$ tmux new -s test
```

With the preceding command, *tmux* is running as a server, and you find yourself in a shell you’ve configured in *tmux*, running as the client. This client/server model allows you to create, enter, leave, and destroy sessions and use the shells running in it without having to think of the processes running (or failing) in it.

*tmux* uses *Ctrl+b* as the default keyboard shortcut, also called *prefix* or *trigger*. So for example, to list all windows, you would press *Ctrl+b* and then *w*, or to expand the current (active) pane, you would use *Ctrl+b* and then *z*.



In `tmux` the default trigger is `Ctrl+b`. To improve the flow, I mapped the trigger to an unused key, so a single keystroke is sufficient. I did this by first mapping the trigger to the Home key in `tmux` and then mapping that Home key to the Caps Lock key by changing its mapping in `/usr/share/X11/xkb/symbols/pc` to `key <CAPS> { [ Home ] };`.

This double-mapping was a workaround I needed to do. Depending on your target key or terminal, you might not have to do this, but I encourage you to map `Ctrl+b` to an unused key you can easily reach since you will press it many times a day.

You can now use any of the commands listed in [Table 3-4](#) to manage further sessions, windows, and panes. Also, when pressing `Ctrl+b+d`, you can detach sessions. This means effectively that you put `tmux` into the background.

When you then start a new terminal instance or, say, you `ssh` to your machine from a remote place, you can then attach to an existing session, so let's do that with the `test` session we created earlier:

```
$ tmux attach -t test ❶
```

- ❶ Attach to existing session called `test`. Note that if you want to detach the session from its previous terminal, you would also supply the `-d` parameter.

[Table 3-4](#) lists common `tmux` commands grouped by the units discussed, from widest scope (session) to narrowest (pane).

*Table 3-4. tmux reference*

Target	Task	Command
Session	Create new	<code>:new -s NAME</code>
Session	Rename	<code>trigger + \$</code>
Session	List all	<code>trigger + s</code>
Session	Close	<code>trigger</code>
Window	Create new	<code>trigger + c</code>
Window	Rename	<code>trigger + ,</code>
Window	Switch to	<code>trigger + 1 ... 9</code>
Window	List all	<code>trigger + w</code>
Window	Close	<code>trigger + &amp;</code>
Pane	Split horizontal	<code>trigger + "</code>
Pane	Split vertical	<code>trigger + %</code>
Pane	Toggle	<code>trigger + z</code>
Pane	Close	<code>trigger + x</code>

Now that you have a basic idea of how to use `tmux`, let's turn our attention to configuring and customizing it. My `.tmux.conf` looks as follows:

```
unbind C-b ❶
set -g prefix Home
bind Home send-prefix
bind r source-file ~/.tmux.conf \; display "tmux config reloaded :)" ❷
bind \ split-window -h -c "#{pane_current_path}" ❸
bind - split-window -v -c "#{pane_current_path}"
bind X confirm-before kill-session ❹
set -s escape-time 1 ❺
set-option -g mouse on ❻
set -g default-terminal "screen-256color" ❼
set-option -g status-position top ❽
set -g status-bg colour103
set -g status-fg colour215
set -g status-right-length 120
set -g status-left-length 50
set -g window-status-style fg=colour215
set -g pane-active-border-style fg=colour215
set -g @plugin 'tmux-plugins/tmux-resurrect' ❾
set -g @plugin 'tmux-plugins/tmux-continuum'
set -g @continuum-restore 'on'
run '~/.tmux/plugins/tpm/tpm'
```

- ❶ This line and the next two lines change the trigger to Home.
- ❷ Reload config via trigger + r.
- ❸ This line and the next redefine pane splitting; retain current directory of existing pane.
- ❹ Adds shortcuts for new and kill sessions.
- ❺ No delays.
- ❻ Enable mouse selections.
- ❼ Set the default terminal mode to 256-color mode.
- ❽ Theme settings (next six lines).
- ❾ From here to the end: plug-in management.

First install `tpm`, the `tmux` plug-in manager, and then trigger + I for the plug-ins. The plug-ins used here are the following:

### *tmux-resurrect*

Allows you to restore sessions with Ctrl+s (safe) and Ctrl+r (restore)

### *tmux-continuum*

Automatically saves/restores a session (15-minute interval)

Figure 3-9 shows my Alacritty terminal running `tmux`. You can see the sessions with the shortcuts 0 to 9, located in the left upper corner.

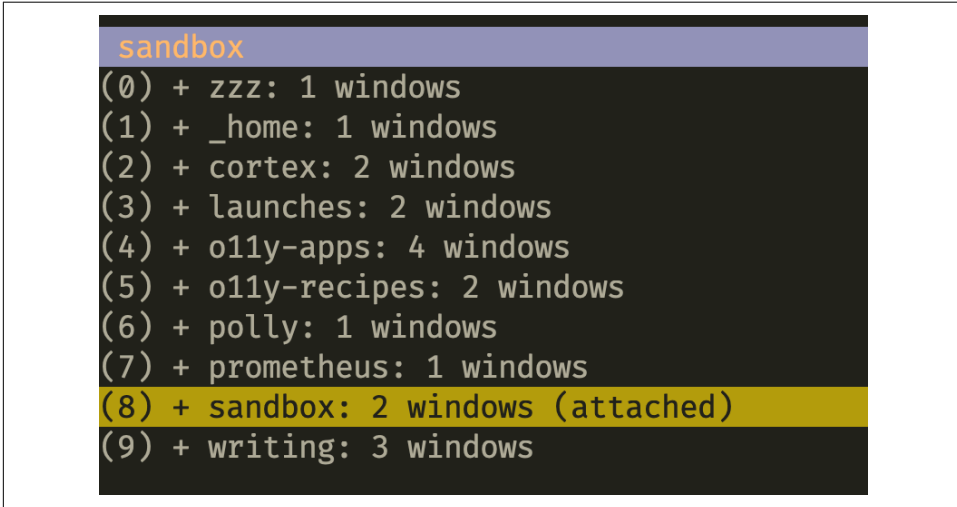


Figure 3-9. An example `tmux` instance in action, showing available sessions

While `tmux` certainly is an excellent choice, there are indeed other options than `tmux`, so let's have a peek.

## Other Multiplexers

Other terminal multiplexers you can have a look at and try out include the following:

### *tmuxinator*

A meta-tool allowing you to manage `tmux` sessions

### *Byobu*

A wrapper around either `screen` or `tmux`; it's especially interesting if you're using the Ubuntu- or Debian-based Linux distros

### *Zellij*

Calls itself a terminal workspace, is written in Rust, and goes beyond what `tmux` offers, including a layout engine and a powerful plug-in system

### *dtm*

Brings the concept of tiling window management to the terminal; it's powerful but has a learning curve like `tmux`

### *3mux*

A simple terminal multiplexer written in Go; it's easy to use but not as powerful as `tmux`

With this quick review of multiplexer options out of the way, let's talk about selecting one.

## Bringing It All Together: Terminal, mux, and shell

I'm using Alacritty as my terminal. It's fast, and best of all, to configure it I'm using a YAML configuration file that I can version in Git, allowing me to use it on any target system in seconds. This config file called *alacritty.yml* defines all my settings for the terminal, from colors to key bindings to font sizes.

Most of the settings apply right away (hot-reload), others when I save the file. One setting, called `shell`, defines the integration between the terminal multiplexer I use (`tmux`) and the shell I use (`fish`) and looks as follows:

```
...
shell:
  program: /usr/local/bin/fish
  args:
    - -l
    - -i
    - -c
    - "tmux new-session -A -s zzz"
...
```

In the preceding snippet, I configure Alacritty to use `fish` as the default shell, but also, when I launch the terminal, it automatically attaches to a specific session. Together with the `tmux-continuum` plug-in, this gives me peace of mind. Even if I switch off the computer, once I restart I find my terminal with all its sessions, windows, and panes (almost) exactly in the state it was in before a crash, besides the shell variables.

## Which Multiplexer Should I Use?

Unlike with shells for human users, I do have a concrete preference here in the context of terminal multiplexer: use `tmux`. The reasons are manifold: it is mature, stable, rich (has many available plug-ins), and flexible. Many folks are using it, so there's plenty of material out there to read up on as well as help available. The other

multiplexers are exciting but relatively new or are, as is the case with `screen`, no longer in their prime.

With that, I hope I was able to convince you to consider using a terminal multiplexer to improve your terminal and shell experience, speed up your tasks, and make the overall flow smoother.

Now, we turn our attention to the last topic in this chapter, automating tasks with shell scripts.

## Scripting

In the previous sections of this chapter, we focused on the manual, interactive usage of the shell. Once you’ve done a certain task over and over again manually on the prompt, it’s likely time to automate the task. This is where scripts come in.

Here we focus on writing scripts in `bash`. This is due to two reasons:

- Most of the scripts out there are written in `bash`, and hence you will find a lot of examples and help available for `bash` scripts.
- The likelihood of finding `bash` available on a target system is high, making your potential user base bigger than if you used a (potentially more powerful but esoteric or not widely used) alternative to `bash`.

Just to provide you with some context before we start, there are shell scripts out there that clock in at **several thousands** of lines of code. Not that I encourage you to aim for this—quite the opposite: if you find yourself writing long scripts, ask yourself if a proper scripting language such as Python or Ruby is the better choice.

Let’s step back now and develop a short but useful example, applying good practices along the way. Let’s assume we want to automate the task of displaying a single state-ment on the screen that, given a user’s GitHub handle, shows when the user joined, using their full name, something along the lines of the following:

```
XXXX XXXXX joined GitHub in YYYY
```

How do we go about automating this task with a script? Let’s start with the basics, then review portability, and work our way up to the “business logic” of the script.

## Scripting Basics

The good news is that by interactively using a shell, you already know most of the relevant terms and techniques. In addition to variables, streams and redirection, and common commands, there are a few specific things you want to be familiar with in the context of scripts, so let’s review them.



## Advanced data types

While shells usually treat everything as strings (if you want to perform some more complicated numerical tasks, you should probably not use a shell script), they do support some advanced data types such as arrays.

Let's have a look at arrays in action:

```
os=( 'Linux' 'macOS' 'Windows' ) ❶  
echo "${os[0]}" ❷  
numberofos="${#os[@]}" ❸
```

- ❶ Define an array with three elements.
- ❷ Access the first element; this would print Linux.
- ❸ Get the length of the array, resulting in `numberofos` being 3.

## Flow control

Flow control allows you to branch (`if`) or repeat (`for` and `while`) in your script, making the execution dependent on a certain condition.

Some usage examples of flow control:

```
for afile in /tmp/* ; do ❶  
    echo "$afile"  
done  
  
for i in {1..10}; do ❷  
    echo "$i"  
done  
  
while true; do  
    ...  
done ❸
```

- ❶ Basic loop iterating over a directory, printing each file name
- ❷ Range loop
- ❸ Forever loop; break out with `Ctrl+C`

## Functions

Functions allow you to write more modular and reusable scripts. You have to define the function before you use it since the shell interprets the script from top to bottom.

A simple function example:

```
sayhi() { ❶
    echo "Hi $1 hope you are well!"
}

sayhi "Michael" ❷
```

- ❶ Function definition; parameters implicitly passed via \$n
- ❷ Function invocation; the output is “Hi Michael hope you are well!”

## Advanced I/O

With `read` you can read user input from `stdin` that you can use to elicit runtime input—for example, with a menu of options. Further, rather than using `echo`, consider `printf`, which allows you fine-grained control over the output, including colors. `printf` is also more portable than `echo`.

Following is an example usage of the advanced I/O in action:

```
read name ❶
printf "Hello %s" "$name" ❷
```

- ❶ Read value from user input.
- ❷ Output value read in the previous step.

There are other, more advanced concepts available for you, such as [signals and traps](#). Given that we want to provide only an overview and introduction to the scripting topic here, I will refer you to the excellent [bash Scripting Cheatsheet](#) for a comprehensive reference of all the relevant constructs. If you are serious about writing shell scripts, I recommend you read [bash Cookbook](#) by Carl Albing, JP Vossen, and Cameron Newham, which contains lots and lots of great snippets you can use as a starting point.

## Writing Portable bash Scripts

We’ll now look at what it means to write portable scripts in bash. But wait. What does *portable* mean, and why should you care?

At the beginning of [“Shells” on page 33](#), we defined what *POSIX* means, so let’s build on that. When I say “portable,” I mean that we are not making too many assumptions—implicitly or explicitly—about the environment a script will be executed in. If a script is portable, it runs on many different systems (shells, Linux distros, etc.).

But remember that, even if you pin down the type of shell, in our case to bash, not all features work the same way across different versions of a shell. At the end of the day, it boils down to the number of different environments you can test your script in.

## Executing portable scripts

How are scripts executed? First, let's state that scripts really are simply text files; the extension doesn't matter, although often you find `.sh` used as a convention. But there are two things that turn a text file into a script that is executable and able to be run by the shell:

- The text file needs to declare the interpreter in the first line, using what is called *shebang* (or *hashbang*), which is written as `#!` (see also the first line of the template that follows).
- Then, you need to make the script executable using, for example, `chmod +x`, which allows everyone to run it, or, even better, `chmod 750`, which is more along the lines of the least privileges principle, as it allows only the user and group associated with the script to run it. We'll dive deep into this topic in [Chapter 4](#).

Now that you know the basics, let's have a look at a concrete template we can use as a starting point.

### A skeleton template

A skeleton template for a portable bash shell script that you can use as a seed looks as follows:

```
#!/usr/bin/env bash ❶
set -o errexit ❷
set -o nounset ❸
set -o pipefail ❹

firstargument="${1:-somedefaultvalue}" ❺

echo "$firstargument"
```

- ❶ The *hashbang* instructs the program loader that we want it to use bash to interpret this script.
- ❷ Define that we want to stop the script execution if an error happens.
- ❸ Define that we treat unset variables as an error (so the script is less likely to fail silently).
- ❹ Define that when one part of a pipe fails, the whole pipe should be considered failed. This helps to avoid silent failures.
- ❺ An example command-line parameter with a default value.

We will use this template later in this section to implement our GitHub info script.

## Good practices

I'm using *good* practices instead of *best* practices because what you should do depends on the situation and how far you want to go. There is a difference between a script you write for yourself and one that you ship to thousands of users, but in general, high-level good practices writing scripts are as follows:

### *Fail fast and loud*

Avoid silent fails, and fail fast; things like `errexit` and `pipefail` do that for you. Since bash tends to fail silently by default, failing fast is almost always a good idea.

### *Sensitive information*

Don't hardcode any sensitive information such as passwords into the script. Such information should be provided at runtime, via user input or calling out to an API. Also, consider that a `ps` reveals program parameters and more, which is another way that sensitive information can be leaked.

### *Input sanitization*

Set and provide sane defaults for variables where possible, and sanitize the input you receive from users or other sources. For example, launch parameters provided or interactively ingested via the `read` command to avoid situations where an innocent-looking `rm -rf "$PROJECTHOME/"` wipes your drive because the variable wasn't set.

### *Check dependencies*

Don't assume that a certain tool or command is available, unless it's a build-in or you know your target environment. Just because your machine has `curl` installed doesn't mean the target machine has. If possible, provide fallbacks—for example, if no `curl` is available, use `wget`.

### *Error handling*

When your script fails (and it's not a matter of if but when and where), provide actionable instructions for your users. For example, rather than `Error 123`, say what has failed and how your user can fix the situation, such as `Tried to write to /project/xyz/ but seems this is read-only for me`.

### *Documentation*

Document your scripts inline (using `# Some doc here`) for main blocks, and try to stick to 80-column width for readability and diffing.

### *Versioning*

Consider versioning your scripts using Git.

## Testing

Lint and *test* the scripts. Since this is such an important practice, we will discuss it in greater detail in the next section.

Let's now move on to making scripts safe(r) by linting them while developing and testing them before you distribute them.

## Linting and Testing Scripts

While you're developing, you want to check and lint your scripts, making sure that you're using commands and instructions correctly. There's a nice way to do that, depicted in [Figure 3-10](#), with the program **ShellCheck**; you can download and install it locally, or you can also use the online version via [shellcheck.net](https://shellcheck.net). Also, consider formatting your script with **shfmt**. It automatically fixes issues that can be reported later by **shellcheck**.

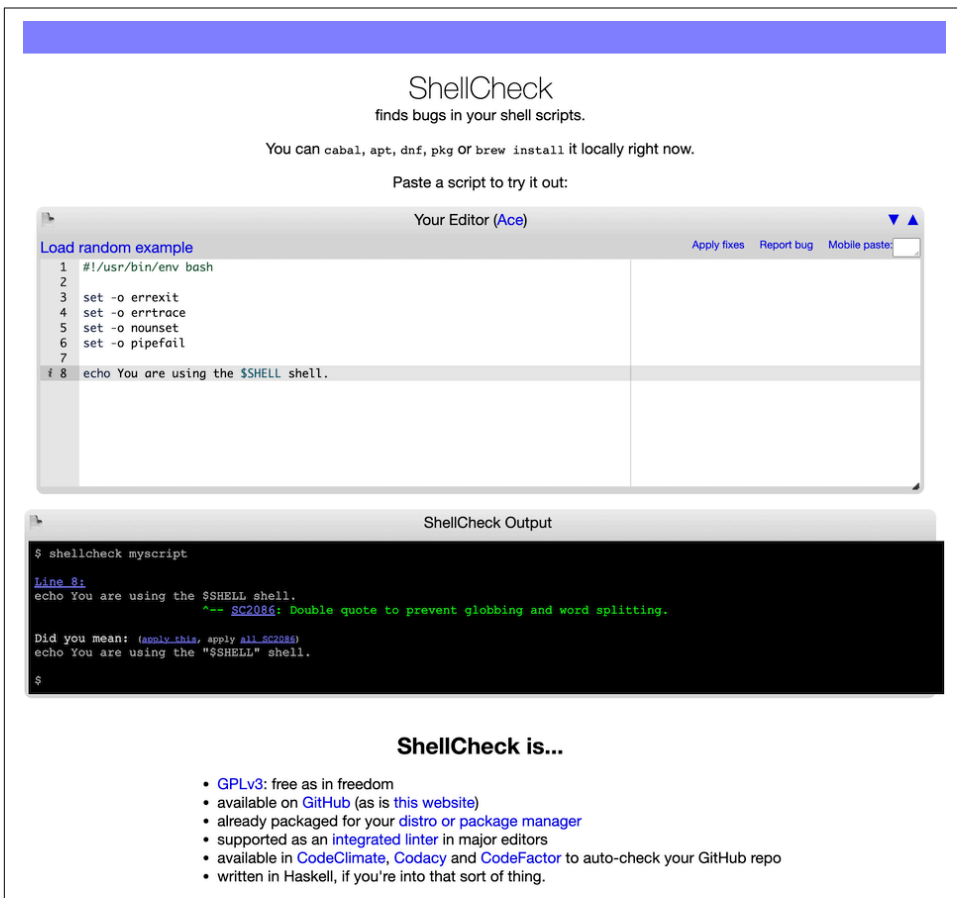


Figure 3-10. A screenshot of the online ShellCheck tool

And further, before you check your script into a repo, consider using **bats** to test it. **bats**, short for Bash Automated Testing System, allows you to define test files as a bash script with special syntax for test cases. Each test case is simply a bash function with a description, and you would typically invoke these scripts as part of a CI pipeline—for example, as a GitHub action.

Now we'll put our good practices for script writing, linting, and testing into use. Let's implement the example script we specified in the beginning of this section.

## End-to-End Example: GitHub User Info Script

In this end-to-end example, we bring all of the preceding tips and tooling together to implement our example script that is supposed to take a GitHub user handle and print out a message that contains what year the user joined, along with their full name.

This is how one implementation looks, taking the good practices into account. Store the following in a file called *gh-user-info.sh*, and make it executable:

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

set -o errexit
set -o errtrace
set -o nounset
set -o pipefail

### Command line parameter:
targetuser="${1:-mhausenblas}" ❶

### Check if our dependencies are met:
if ! [ -x "$(command -v jq)" ]
then
    echo "jq is not installed" >&2
    exit 1
fi

### Main:
githubapi="https://api.github.com/users/"
tmpuserdump="/tmp/ghuserdump_${targetuser}.json"

result=$(curl -s $githubapi${targetuser}) ❷
echo $result > $tmpuserdump

name=$(jq .name $tmpuserdump -r) ❸
created_at=$(jq .created_at $tmpuserdump -r)

joinyear=$(echo $created_at | cut -f1 -d"-") ❹
echo $name joined GitHub in $joinyear ❺
```

- ❶ Provide a default value to use if user doesn't supply one.
- ❷ Using `curl`, access the **GitHub API** to download the user information as a JSON file, and store it in a temporary file (next line).
- ❸ Using `jq`, pull out the fields we need. Note that the `created_at` field has a value that looks something like "2009-02-07T16:07:32Z".
- ❹ Using `cut`, extract the year from the `created_at` field in the JSON file.
- ❺ Assemble the output message and print to screen.

Now let's run it with the defaults:

```
$ ./gh-user-info.sh  
Michael Hausenblas joined GitHub in 2009
```

Congratulations, you now have everything at your disposal to use the shell, both interactively on the prompt and for scripting. Before we wrap up, take a moment to think about the following concerning our *gh-user-info.sh* script:

- What if the JSON blob the GitHub API returns is not valid? What if we encounter a 500 HTTP error? Maybe adding a message along the lines of “try later” is more useful if there's nothing the user can do themselves.
- For the script to work, you need network access, otherwise the `curl` call will fail. What could you do about a lack of network access? Informing the user about it and suggesting what they can do to check networking may be an option.
- Think about improvements around dependency checks—for example, we implicitly assume here that `curl` is installed. Can you maybe add a check that makes the binary variable and falls back to `wget`?
- How about adding some usage help? If the script is called with an `-h` or `--help` parameter, perhaps show a concrete usage example and the options that users can use to influence the execution (ideally, including defining default values used).

You see now that, although this script looks good and works in most cases, there's always something you can improve, such as making the script more robust and providing actionable error messages. In this context, consider using frameworks such as **bashing**, **rerun**, or **rr** to improve modularity.

# Conclusion

In this chapter, we focused on working with Linux in the terminal, a textual user interface. We discussed shell terminology, provided a hands-on introduction to using the shell basics, and reviewed common tasks and how you can improve your shell productivity using modern variants of certain commands (such as `exa` rather than `ls`).

Then, we looked at modern, human-friendly shells, specifically at `fish`, and how to configure and use them. Further, we covered the terminal multiplexer by using `tmux` as the hands-on example, enabling you to work with multiple local or remote sessions. Using modern shells and multiplexers can dramatically improve your efficiency on the command line, and I strongly recommend you consider adopting them.

Lastly, we discussed automating tasks by writing safe and portable shell scripts, including linting and testing said scripts. Remember that shells are effectively command interpreters, and as with any kind of language, you have to practice to get fluent. Having said this, now that you're equipped with the basics of using Linux from the command line, you can already work with the majority of Linux-based systems out there, be it an embedded system or a cloud VM. In any case, you'll find a way to get hold of a terminal and issue commands interactively or via executing scripts.

If you want to dive deeper into the topics discussed in this chapter, here are some additional resources:

## *Terminals*

- [“Anatomy of a Terminal Emulator”](#)
- [“The TTY Demystified”](#)
- [“The Terminal, the Console and the Shell—What Are They?”](#)
- [“What Is a TTY on Linux? \(and How to Use the `tty` Command\)”](#)
- [“Your Terminal Is Not a Terminal: An Introduction to Streams”](#)

## *Shells*

- [“Unix Shells: `bash`, `fish`, `ksh`, `tcsh`, `zsh`”](#)
- [“Comparison of Command Shells”](#)
- [“`bash` vs `zsh`” thread on reddit](#)
- [“Ghost in the Shell—Part 7—ZSH Setup”](#)