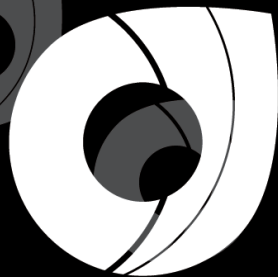




Corndel DevOps Engineering Programme

in association with Softwire

Module 4: Command Line Tasks



Corndel
Digital.

Introduction

In the modern world we usually interact with computers via a **graphical user interface (GUI)**. Under the hood, all those visible interactions are functions being run to update and fetch data. The GUI is not the only way to interact with a computer and trigger those functions, and in this module we'll introduce interacting with computers via a text-only mode known as the **command line**.

The command line typically is more powerful and flexible than a corresponding GUI, and for many DevOps-focused tasks it is the only option. Being proficient in command line work is an essential skill for any developer or DevOps engineer.

There is a rich ecosystem of command line-based software. Such tools may appear out-dated, but their development is driven by the fact that command line applications are more performant, more able to inter-operate, and much easier to automate than GUI-based equivalents. It also helps that developing CLI-only software is faster and cheaper.

In this module we'll learn how to work effectively at the command line. We'll learn what **terminals** and **shells** are, and discuss how they help us talk to the operating system, and use this knowledge to build complex commands and scripts to help automate command line tasks. Finally, we'll move to discuss **virtual machines**, and how command line tools are helpful, and indeed required, to work effectively with them.

Table of Contents

Core Material

Introduction

Terminology

Chapter 1:

The Shell

- Common Shells
- Common Tasks

Chapter 2:

Bash Scripting

- Why are shell scripts useful?
- Syntax
- Shell Options
- Scheduled Jobs using Cron
- Finally, An Example

Chapter 3:

Virtual Machines

- What is a Virtual Machine?
- Using a local VM
- The Ubiquity of Virtualisation

Chapter 4:

Automating VM Management

- Shell Setup Scripts
- Images
- Configuration Management Tools

Additional Material

Validating Shell Scripts

PowerShell Aliasing

Package Managers

Vagrant

Windows Subsystem for Linux

Secret Management

Are VMs obsolete?

Terminology

Let's start by defining some of the common terms we'll use throughout this module. There is a lot of technical jargon surrounding command line interfaces, much of it historical in origin. Getting the terminology right is the first step in mastering what can seem an impenetrable topic.

Kernel

The **kernel** is a central component of the operating system. It acts as a bridge between software and hardware, and is responsible for controlling the programs that run on the computer. In particular, the kernel is responsible for managing processes, memory and disk usage.

There are just a handful of kernels used on modern operating systems. They're complicated pieces of software, and it rarely makes sense to build one from scratch.

- **Linux Kernel.** A famous piece of open-source software, the Linux kernel, first released in 1991, powers all Linux distributions, and many other platforms (including the Android operating system)
- **Darwin Kernel.** The Darwin kernel powers all modern Apple operating systems (e.g. macOS and iOS). It was developed and released by Apple in 2000, based on earlier work at NEXT.

- **Windows NT Kernel.** Released as part of Windows 3.1 in 1993 by Microsoft, the Windows NT kernel has been part of all Windows OS releases since (earlier Windows versions were instead based on MS-DOS).

Operating System

A computer **operating system (OS)** is the collection of software that manages a machine's hardware resources, user interactions, and provides common services for other software installed on the system. It consists of a kernel, plus a lot of other applications needed to make the computer useful.

Operating systems have a long history, and have changed greatly as computing hardware has become more complex and user demands of computers have evolved. In the modern world, there are a few main groups of operating systems in widespread use today, closely tied to the kernels mentioned above.

- Windows
- Android
- macOS and iOS
- Linux Distributions (e.g. Ubuntu)

You may also encounter less mainstream operating systems from the wider family of UNIX-like OSes. We won't discuss them explicitly as users on more specialised

systems are likely already well aware of the shell tools at their disposal.

Shell

A shell is a piece of software that provides an interface for a user to communicate with the kernel. Shells are one of the many pieces of software (besides the kernel) that make up an operating system. Every modern OS includes at least one, and you can often install more. Some examples of shells are:

1. bash
2. ksh
3. zsh
4. Windows Command Prompt
5. PowerShell

Terminal

Shells and terminals are often confused. The shell is a piece of software. The terminal is the tool that displays the shell's output to a user. The name derives from the word 'terminate' - it is where the computer ends!

The idea of a terminal can be a little confusing to users of modern computer hardware. Nowadays, a terminal is just the software window that displays the output from a shell program (i.e. a featureless black box). Hardware or software, terminals do not do any complex processing. They exist to

pass text back-and-forth between user and computer.

A Bit of History

It's worth taking a moment to cover some of the history of terminal hardware. It's not directly relevant to modern computing, however the nomenclature of now-obsolete hardware remains in widespread use, and the context can help make sense of it.

The original terminals were called **teletypewriters (TTYs)**. These were physical devices generally separated from the computer. The user types into it, and the TTY types the text onto paper *and* inputs the text into the system. The computer could then reply - typing onto the same paper.

As time progressed, there was a move from TTYs to '**Console Cabinets**'. The Console was a cabinet with a screen and a keyboard which could act as a terminal. Technically, the console here is the device and the terminal is the software running on the console, but the terms are widely used interchangeably in modern computing.

We don't use the original TTYs any more, nor do we have dedicated console cabinets, but the concept of a terminal is still extremely useful. Over time, TTY has come to mean a software based version of a TTY. They generally provide a text-based app that

allows the user to interact with the shell of their choice.

Modern Terminals

Modern terminals are a software window, just one of many in a GUI-based operating system. Some are standalone applications (e.g. iTerm, Windows Terminal, Linux terminals, Cmder, PuTTY) while others are embedded in larger applications (e.g. the VSCode terminal).

Generally speaking, it is the shell running in a terminal that determines its utility, not the terminal software itself. In most environments you can freely mix-and-match terminals and shells.

POSIX

You'll find a lot of references to POSIX when researching tools for Linux and Mac. But what is it? POSIX is a set of standards published by the IEEE (IEEE 1003, to be precise). It stands for "Portable Operating System Interface". POSIX defines an industry standard for how a Unix-like operating system should behave.

As well as core operating system APIs (e.g. file system interaction, networking). POSIX also defines standard behaviour for many common command-line utilities such as grep, sed and awk. POSIX is not an operating system, but is a standard that many operating systems comply with (or are close to).

POSIX began in 1988 in an effort to standardise the disparate behaviour of UNIX variants at the time, and we can largely thank the standards for ensuring that many modern operating systems look and feel very similar, allowing users and programs to move between them relatively easily.

Chapter 1

The Shell

A shell is an application that translates user input into commands the operating system can understand. "Wait", I hear you say, "Isn't that what every computer program does?" Sort of. The aim of a shell is quite specific though. Its purpose is to help you communicate with the operating system, nothing more. You can think of it as a translator.

Shells are hugely important to software development, systems administration and DevOps. With a shell you can manage core operating system features (e.g. monitor system resources, manage system configuration, or create new files) but also ask the system to launch other applications.

In this Section we'll introduce shells and explore their basic functionality. We'll take the opportunity to dip our toes into the world of CLI tools, in particular exploring some common utilities in the UNIX ecosystem. You'll learn how to navigate and maintain a directory structure, manage file permissions, handle data and issue network requests. This will lay the foundations for the next section where we'll start writing shell scripts to automate our workflows.

This is an extremely wide-ranging section, introducing many pieces of software about which whole books have been written. Don't worry if you're not quite sure of some details at this stage. A lot of details are best picked

up through practice, and we'll link to some further resources at the end of this module.

Common Shells

Let's run through some of the shells you'll hear about. This is not an exhaustive list! After a brief overview we'll dive into the practical details of how to use a shell, and the syntax you'll need to accomplish routine tasks.

Before we introduce individual shells, it's worth reviewing what the shell ecosystem looks like. No two shells are quite identical. You can think of each as its own scripting language and feature bundle, that may-or-may-not be tied to a particular operating system. But they do fall into groups, and there's three that are worth talking about:

- **Bourne-family shells.** These are the de-facto shells on most UNIX-like operating systems (a somewhat vague category that includes Apple Macs and all Linux distributions). It includes the original Bourne shell, as well as more modern options such as bash, the korn shell and the Z-shell. We'll focus on these for much of this section.
- **C-family shells.** An alternative shell family on UNIX-like operating systems, including the original C shell

(csh) and tsh. These shells have largely fallen out of popularity, in particular due to weak scripting support when compared to the Bourne shell and its successors. For this reason we won't discuss or use the C-shell further, but you should be aware that it exists!

- **Windows shells.** Being an essentially unrelated operating system (with a completely different kernel) to UNIX-like systems, Windows has its own set of shells developed by Microsoft.

Keep in mind that shells are complex computer programs and can take some time to learn! We won't dive into all the command syntax and features here. Rather, we'll introduce tools as and when they are needed. You'll find the internet an extremely powerful resource when researching how to accomplish tasks via a shell program. Their ubiquity, and the relatively small number of commonly used shells, mean that you're unlikely to hit a problem that hasn't been solved before.

Bourne Shell (sh)

Shells have a long history and pre-date graphical user interfaces on computers. One of the first to be widely distributed was the **Bourne shell**. Included with UNIX Version 7 in 1979, the Bourne shell was developed by

Stephen Bourne at Bell Labs. It built on common features found in earlier shell programs, while adding extensions to define functionality now common to the derivatives of the Bourne shell that dominate the modern shell scripting landscape.

You're unlikely to use the Bourne shell for professional work today. While sh is an important part of computing history, several more modern, more powerful shells are backward compatible with it, while also providing more advanced features. Shells derived from the Bourne shell (sh) include the Korn shell (ksh), bash and zsh.

The bourne shell is still commonly available on UNIX-derived systems (including MacOS and Linux) at `/bin/sh`.

Bash

Bash stands for Bourne Again SHell, and was released for free in 1989 by Brian Fox as part of the GNU Project. *Bash is the de-facto standard*. You'll see it on most UNIX-like operating systems. With a little configuration you can run bash on Windows (via WSL, or emulated using tools such as Git BASH or Cygwin).

Every shell has its own scripting language and features, but when people simply talk about "shell scripting" they almost always mean using bash. This ubiquity is important.

You can rely on bash being present on almost all machines you might need to work with. Similarly, if you're writing a shell script that can run natively on most non-Windows operating systems, you should make sure it's bash compliant.

As the name implies, bash is part of the Bourne-family of shells. It is entirely backward-compatible with the Bourne shell. Bash is the default shell on most Linux distributions, and on Mac OS prior to version 10.15 (Mac OS Catalina).

Z-Shell (zsh)

The Z-Shell (also known as zsh) is another popular, modern Bourne-family shell on Linux and Mac operating systems. It was released in 1990, and can be considered an alternative bash. The mainstream features of the bash and zsh are very similar, and there's no steep learning curve between the two. Each has its own unique features, with zsh having some bells-and-whistles that certain users prefer. As of Mac OS 10.15 (Catalina), zsh is the default shell on Apple Macs.

One of the biggest selling points of zsh is it's support for customisation and plugins. There are thousands available, and an online community maintain a free bundle of plugins and themes known as Oh-My-Zsh that can provide powerful functionality with minimal config. You can read more about this project

on [github](https://github.com/ohmyzsh/ohmyzsh) (<https://github.com/ohmyzsh/ohmyzsh>).

Windows Shells

All the shells we've talked about so far are available across a range of UNIX-like operating systems (including Linux and Mac OS). Unfortunately the situation on Windows is a little more complicated. Windows has an entirely different kernel and uses several of its own shells with distinct command syntax, each of which is integrated with, and inseparable from, a custom terminal.

There are options for running Bourne-family shells on Windows, typically via emulation. Many users choose to install tools such as **git BASH**, or **cygwin**. These tools let you use an alternative shell, and under-the-hood they map the OS-level interactions to Windows API calls. However, there's a limited ecosystem for installing command line tools and these approaches are no substitute for a real UNIX-like system.

Windows Shell

The **Windows shell** is the graphical user interface for Windows, including all the hallmark elements such as the Desktop, Start Bar and Task Bar. This may seem a bit odd, but there's no reason a shell needs to be a text-based CLI tool (although most are)! While it's worth mentioning, the value of a

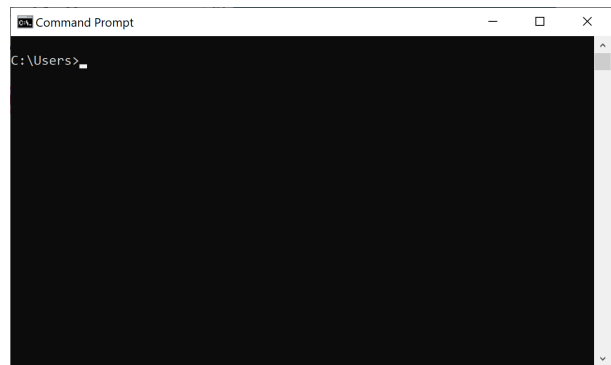
GUI-based shell is limited for DevOps purposes where you'll usually prefer the lightweight and powerful options of a CLI shell.

Command Prompt

Command prompt (`cmd.exe`) is the original Windows shell, common to all versions of the current Windows NT architecture (back to Windows NT 3.1 in 1993). Its history stretches back further, and is based on the `COMMAND.COM` shell found on MS-DOS systems. It fills a similar role to the UNIX shells discussed previously, but is more limited, uses its own syntax and is not nearly so popular with its users!

Like other shells on the Windows platform, `cmd.exe` provides its own terminal - the Command Prompt window. The GUI is limited, and does not support text styling, nor much in the way of keyboard shortcuts or sophisticated navigation.

Similar to UNIX shells, `cmd.exe` can run as an interactive command line session or used to execute scripts known as "batch files" (with the file extension `.bat`)



Command Prompt

Command prompt is generally thought of a weak tool. There are some specific Windows features where `cmd.exe` is required, but wherever possible you should look towards the next option as your native Windows shell: **Powershell.**

PowerShell

PowerShell was developed as a wholesale replacement to overcome some of the limitations of `cmd.exe`. It provides Windows users with a more powerful shell experience, starting from Windows 7. Like command prompt, PowerShell supports both interactive sessions and can run script files. Once again it uses its own scripting language that you'll need to learn if interacting with the Windows operating system. Powershell script files usually have the extension `.ps1`.

PowerShell is available in two flavours. The first, **Windows PowerShell**, is distributed as part of Windows and is based on the .NET

Framework. Microsoft have recently released a cross-platform open-source PowerShell implementation, **PowerShell Core** based on .NET Core. You can download PowerShell Core from [github](https://github.com/PowerShell/PowerShell) (<https://github.com/PowerShell/PowerShell>) if interested in this option on Mac or Linux devices.

PowerShell takes an object-oriented approach to scripting, and works using

'cmdlets' - verb-noun pair that describe an action being performed. Cmdlets can take parameters and, similar to other scripting languages, cmdlets can be composed to build more complex functionality using piping.

For example, this powershell snippet shows how to find all Python files nested under the current directory:

```
# Normal, verbose version (using explicit cmdlet naming)
Get-ChildItem -Recurse *.py | ForEach-Object { Write-Host $_.FullName }

# Shortened, equivalent version
gci -s *.py | % { $_.FullName }
```

The equivalent operation using bash is typically performed with the `find` command:

```
find . -type f -name '*.py'
```

Powershell's cmdlet-based approach to script is extremely verbose and readable by default. This can be helpful, but PowerShell also includes many aliases to support more succinct commands where brevity is more important than readability.

WSL

What are your options if you're working on Windows, but want to use industry-standard UNIX shells for scripting and automation?

Well, you're in luck. One of the latest developer-focused Windows features is the **Windows Subsystem for Linux (WSL)**. In short, WSL (version 2) adds a Linux kernel to

your Windows operating system. You can install your chosen Linux distribution and run Linux command line tools (including shells) alongside your normal Windows command line and GUI applications. WSL doesn't, to date, support GUI applications.

In WSL you can run standard UNIX shells such as `bash` or `zsh`. This greatly reduces the need for imperfect emulation tools like `git bash` and `cygwin` on the Windows platform.

Common Tasks

We've introduced the most common shells across the major modern operating systems. Now let's start using them. Here we'll run through some operations you'll use a shell for. Individually most of these may seem trivial, but in the next section we'll use these pieces to build sophisticated scripts.

For all these tasks we'll discuss the operation in bash terminology, but will give code snippets for PowerShell. Other UNIX shells will operate identically to bash.

Managing Directories

Directory navigation and manipulation commands are an essential part of general shell usage. You'll use commands like `cd`, `pwd`, `ls`, `mkdir` and `rm` every day to work with directories. There are all standard UNIX shell commands, and PowerShell includes them also as aliases.

You use `cd` to navigate to a new directory, providing either a relative or absolute path from your current location.

```
$ cd ./subdirectory
```

In PowerShell `cd` is an Alias for the `Set-Location` cmdlet. Some PowerShell cmdlets allow you to leave out the parameter name,

as shown in this example. This is fine for simple commands, but bear in mind that idiomatic PowerShell scripting should be explicit and use the full form.

```
# These do the same thing
$ Set-Location -Path ./subdirectory
$ Set-Location ./subdirectory
$ cd ./subdirectory
```

What if you don't know your location? This is usually shown as part of your shell prompt, but sometimes it's useful to access programmatically. The `pwd` command will return your current location.

```
$ pwd
/Users
```

PowerShell uses the `Get-Location` cmdlet for this operation. This is aliased to `pwd` to match the UNIX convention.

```
$ Get-Location
Path
----
/Users
```

Note that, while `pwd` (bash) and `Get-Location` (PowerShell) are superficially similar, they are not doing the same thing. `pwd` returns a string, while `Get-Location` returns an object of type `System.Management.Automation.PathInfo`! The UNIX ecosystem is built of many

small, independent applications that communicate via text streams. PowerShell is different: it uses a common object-oriented framework to pass data between cmdlets.

As well as navigating, you'll want to create new directories (`mkdir`) and remove them (`rm`). By default `rm` will only remove a file. To remove a directory you'll need to set the recursive flag (`-r`). Running `rm` recursively is risky. The command does not ask for confirmation, and there's no way to recover

deleted files. For added safety, you can run `rm` with the `-i` flag. This will prompt the user for confirmation of each deletion.

```
$ mkdir new_directory
# Warning! This is risky!
$ rm -r new_directory
```

In PowerShell you can run exactly the same commands. If you prefer the full cmdlet forms, they are:

```
$ New-Item -Path new_directory -Type Directory
    Directory: /Users/demo

Mode                LastWriteTime         Length Name
----                -
d-----          15/01/2020   13:11             new_directory

$ Remove-Item -Path new_directory
```

Handling Files

You'll need to know how to interact with files via the command line. When writing shell scripts you'll often use files as data sources, destinations, or simply to persist logs of what your script was doing. In the UNIX ecosystem there are, once again, some standard tools that can help you. We'll also talk about how to edit files via the command line - another key skill for occasions when you have no GUI text editor available.

Creating And Deleting Files

There are many ways to make a file using bash. The simplest (and most common) command is `touch`. `Touch` can also be used to modify file access and modification timestamps on an existing file, but that's much less commonly used.

```
# creates my_file.txt in the current directory
$ touch my_file.txt
```

We've already seen how to delete a file! Earlier we used `rm -r` to remove a directory. Without the `-r` flag, `rm` deletes files only.

```
# deletes my_file.txt if it exists
$ rm my_file.txt
```

In PowerShell these same commands will work, but again the more idiomatic forms use the full cmdlet names:

```
$ New-Item -Path my_file.txt -Type File
$ Remove-Item -Path my_file.txt
```

Reading File Content

UNIX shells have a plethora of options for reading the contents of a file. Either of the commands below will read out the entire contents of a file and print it to your terminal:

```
# Both of these commands print the file
# contents to the terminal
cat my_file.txt
< my_file.txt
```

But that's not all! Reading the whole file might not be a good idea for a multi-gigabyte log file. You can use pager programs such as `less` to efficiently search through larger files.

```
less my_file.txt
```

Glob Patterns

Glob patterns (or 'filename expansions') are a way of finding a set of files that match a naming pattern. You've probably used them already. The name 'glob' comes from a very old Unix utility; you're unlikely to find glob on any modern system, and the functionality is now built-in into shells. Nevertheless, the name lives on.

Glob patterns use wildcards to find files that match a pattern. Here are some of the basic ones: These are supported on all modern shells.

`*.py` matches any Python file in the current directory, and `log_[123].txt` would match `log_1.txt` and `log_2.txt` but not

Wildcard	Matches
<code>*</code>	any number of characters
<code>?</code>	single character
<code>[0-9]</code>	any digit
<code>[aA]</code>	'a' or 'A'

`log_4.txt` etc.

Core glob syntax has been expanded over the years, with globstar (or the 'recursive wildcard') being perhaps the most important addition. globstar redefines `**` to recursively scan subdirectories for matching files. For example `**/*.py` will find all Python files under the current location, regardless of directory nesting. This behaviour is enabled by default in zsh, and is opt-in using bash. PowerShell doesn't support any recursive wildcard for filename matching.

Don't confuse glob patterns with regex. While they may look similar, they are distinct technologies. The wildcards are different and glob patterns specifically relate to file paths, not arbitrary text. Glob matching is performed by the shell, whereas regex matching is run by separate utilities (e.g. `grep`).

Less is an interactive program. For example, you can read through pages in order, or search forwards and backwards through a file to match specific patterns. You might also encounter `more`, which is

an older, less-capable utility than `less`. Use `less` in preference.

In PowerShell you also have options to read, and page through content.

```
Get-Content my_file.txt # Prints all file content
Get-Content my_file.txt | Out-Host -Paging # Page the output
```

Editing Files

You can now navigate directories and create, read and delete files in your preferred shell. That's great, and you should practice as much as possible to commit this basic toolkit to memory. Editing existing files interactively is a little more complicated. GUI-based text editors can be complicated, highly extensible tools such as VSCode, and similarly powerful developer tools exist via a CLI interface.

There are plenty of occasions you'll need to interact with a text file without the luxury of GUI-based editors. Perhaps you're connecting to a remote server via **SSH** (more on that later in this module), or you need to modify a file inside a running docker container (discussed in the next module) . Or perhaps you're working on a

low-power integrated device with only a few shell tools available. We'll talk about two commonly-used CLI-based text editors: **nano** and **vi/vim** (there are others).

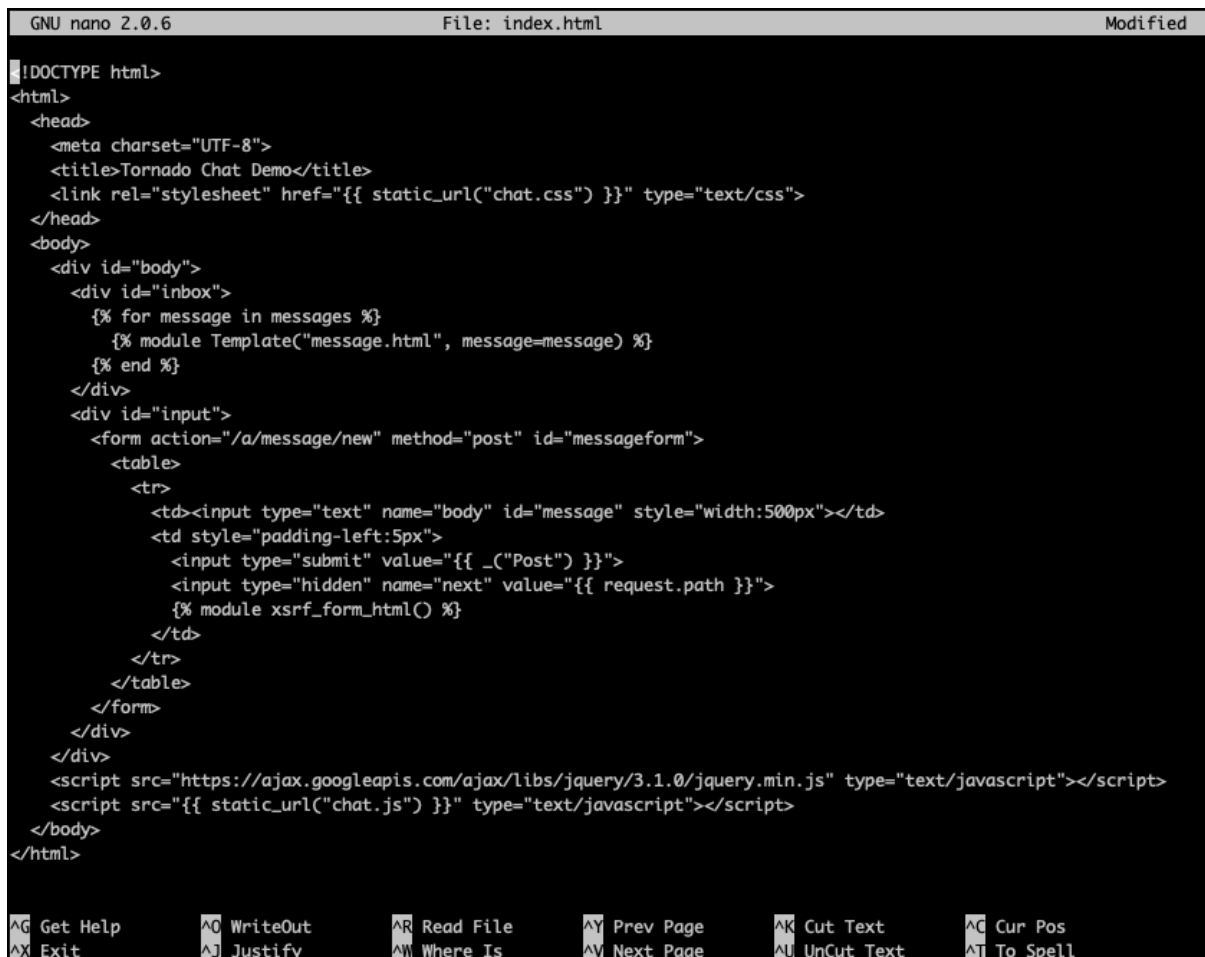
Windows PowerShell does not have equivalent standard CLI editing capabilities as standard. Nano and vim are available for download on the Windows platform. However, if you need to edit text files via the CLI in Windows, we strongly recommend using these UNIX tools via bash or zsh running in WSL.

Nano

Nano is a user-friendly CLI text editor that you'll find pre-installed on most UNIX-like systems. You can launch nano by running:

```
nano <file_path>
```

Once launched you'll be greeted by this familiar GUI, listing context-dependent keyboard shortcuts at the bottom and filling the rest of your terminal window with editor space. Nano is excellent for quick edits (e.g. updating a configuration value). Editing text works as-expected and is approachable to beginners in CLI tasks.



```
GNU nano 2.0.6 File: index.html Modified
!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Tornado Chat Demo</title>
    <link rel="stylesheet" href="{{ static_url("chat.css") }}" type="text/css">
  </head>
  <body>
    <div id="body">
      <div id="inbox">
        {% for message in messages %}
          {% module Template("message.html", message=message) %}
        {% end %}
      </div>
      <div id="input">
        <form action="/a/message/new" method="post" id="messageform">
          <table>
            <tr>
              <td><input type="text" name="body" id="message" style="width:500px"></td>
              <td style="padding-left:5px">
                <input type="submit" value="{{ _("Post") }}">
                <input type="hidden" name="next" value="{{ request.path }}">
                {% module xsrf_form_html() %}
              </td>
            </tr>
          </table>
        </form>
      </div>
    </div>
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.1.0/jquery.min.js" type="text/javascript"></script>
    <script src="{{ static_url("chat.js") }}" type="text/javascript"></script>
  </body>
</html>

^G Get Help      ^O WriteOut      ^R Read File     ^Y Prev Page     ^K Cut Text      ^C Cur Pos
^X Exit          ^J Justify       ^W Where Is      ^V Next Page     ^U UnCut Text    ^T To Spell
```


Vi/Vim

Vi is a very old terminal-based text editor released in 1976 for the Unix operating system. You'll hear it talked about, but rarely (if ever) use it. Instead, Vim (Vi iMproved), released in 1991, is an extensively-enhanced version of vi that is popular amongst developers and DevOps engineers. It's often considered the go-to option for editing text via the CLI thanks to its powerful keyboard shortcuts and extensive plugin systems.

Unfortunately, the minimal-interface approach of Vim, combined with its extensive use of slightly arcane keyboard shortcuts, can make it feel challenging for first-time users. This is not a section about Vim, but we'll run through a the very basics.

To launch vim, run the following from your terminal:

```
vim <file_path>
```

And you'll be greeted by the text-based interface, shown below:

```
!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Tornado Chat Demo</title>
    <link rel="stylesheet" href="{{ static_url("chat.css") }}" type="text/css">
  </head>
  <body>
    <div id="body">
      <div id="inbox">
        {% for message in messages %}
          {% module Template("message.html", message=message) %}
        {% end %}
      </div>
      <div id="input">
        <form action="/a/message/new" method="post" id="messageform">
          <table>
            <tr>
              <td><input type="text" name="body" id="message" style="width:500px"></td>
              <td style="padding-left:5px">
                <input type="submit" value="{{ _("Post") }}">
                <input type="hidden" name="next" value="{{ request.path }}">
                {% module xsrf_form_html() %}
              </td>
            </tr>
          </table>
        </form>
      </div>
      <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.1.0/jquery.min.js" type="text/javascript"></script>
      <script src="{{ static_url("chat.js") }}" type="text/javascript"></script>
    </body>
  </html>
~
~
~
~
~
"index.html" 33L, 1127C
```

Unlike most text editors, Vim has different modes. There are six modes, each providing distinct tools to the user:

- Normal Mode
- Visual Mode
- Select Mode
- Insert Mode
- Command-line Mode
- Ex Mode

By default vim starts in normal mode. Normal mode provides powerful keyboard shortcuts for navigating and searching in a file, but you can't change anything. Let's assume, for now, you want to make a quick text edit and save the file. You would do the following:

1. Find the content to change
2. Enter insert mode (press `i`)

3. Make your change
4. Return to normal mode (press `esc`)
5. Save the file with `:w`, or save and exit with `:wq`

This last command shows how commands in vim can be quickly chained together: `:wq` means write then quit. These power-user optimisations are the main reason for the tool's popularity.

To exit vim without saving any changes, you can run `:q!` from normal mode. The exclamation mark means 'force' - without it the operation will warn you about unsaved changes and require an additional approval step.

File Permissions

Everything you've done so far assumes you have the *permission* to create, edit and delete files. But that isn't always the case. Operating systems provide permission models to control *who* has access to *what*. This controls not only what human users of a file system can do, but also which resources processes have access to.

UNIX permissions model

Mac OS and Linux are all based on the same **UNIX permissions model**, so it's worth recapping how this works. This isn't the only permissions system at work, but it's the one you'll interact with most commonly. Other permission models such as **Access Control Lists (ACLs)** and extended attributes are often used alongside this permissions model,

but we won't dive into them here as they're less relevant to typical DevOps tasks.

The UNIX permissions model defines, for each file and directory in the file system, three permissions that a user can have:

- Read (*r*)
- Write (*w*)
- Execute (*x*)

As you might expect, having these permissions allows you to read from, write to, or execute a file respectively (execute being relevant for a file that happens to be a shell script or program you want to run). The meanings are clear enough for files, but what do the terms mean when granted on directories? Having read permissions on a directory grants visibility of the names of the files contained. Write permissions are needed to manipulate the contents (for example, creating a new file in a directory). Finally, execute permission on a directory is essentially a 'search' permission. Without execute permissions a user can't interact with the contents of a directory, even if they have full permissions to the contents itself.

The same permissions don't apply to everyone! For each file or directory the OS divides all possible users into three classes:

- The Owner (*user class*)
- The Group (*group class*)

- Everyone else (*other class*)

Each class can have different read/write/execute permissions on a file. By default, the owner is the user that created the file. Keep in mind that a user is not necessarily a human at the keyboard! Processes and applications run as a user. A 'group' is simply a collection of users. The group class can be used to grant a specific group elevated permissions on a file.

The permissions on a file or directory can be described in a couple of ways. Imagine a file where the owner has full access (read, write and execute), the group has read & write permissions, while others can only read the file. There are a couple of common notations used to represent these permissions:

- Symbolic Notation: `-rwxrw-r--`
- Octal Notation: `0764`

The first character of symbolic notation is not actually a permission - it's the file type (e.g. `d` is a directory, `l` is a symbolic link and `-` is a regular file). The final 9 characters in the symbolic notation represent, in order, the owner, group and other permissions.

Octal notation works by grouping the 9 permission characters into groups of three and interpreting each flag as a binary digit. This means permissions can be described via

a single digit 0-7 (where 0 is no permissions and 7 is read, write and execute).

	r	w	x	r	w	-	r	-	-
	User			Group			Other		
Binary	1	1	1	1	1	0	1	0	0
Octal	7			6			4		

In octal notation the first digit represents special modes that are only applicable to executable files. These are beyond the scope of this course.

We can view the permissions on a file using `ls -l`. For example, on Mac OS you may see results like those shown below. `__pycache__` and `ui` are directories (d) where everyone has execute permissions, group and owner have read permissions but only the owner has write permissions. Permissions on the file (`__init__.py`) are more restrictive - the owner can read and write, while the others can read.

```
$ ls -l
-rw-r--r-- 1 user_name group_name 0 16 Feb 09:51 __init__.py
drwxr-xr-x 3 user_name group_name 96 16 Feb 12:11 __pycache__
drwxr-xr-x 7 user_name group_name 224 1 Feb 15:22 engine
drwxr-xr-x 6 user_name group_name 192 17 Feb 14:39 ui
```

Modifying UNIX permissions

You'll sometimes need to change file permissions as part of DevOps tasks. Commonly you will need to add the execute permission to a downloaded script file, or perhaps you need to grant a process access to a particular library directory so it can adjust a configuration file as part of an application launch process.

To change file permissions you'll either need to be the file owner, or be able to act as the superuser (i.e. have the necessary credentials to invoke `sudo`). The command we'll use to edit file permissions is `chmod`.

You can call `chmod` using octal or symbolic permissions, and you can easily set permissions on particular classes. Here are a few examples, and you can find full docs by running `man chmod` in your shell.

```
chmod +x my_file    # Grant execute permissions to all
chmod g-w my_file   # Revoke write permissions from the group
chmod 0777 my_file  # Grant full permissions to everyone! WARNING: Dangerous
```

Sometimes you'll also want to change the ownership of the file. Remember, by default the owner is the user who created the file, and you'll need superuser privileges to change this. You can change a file's owner and group using the `chown` and `chgrp` commands, respectively.

Windows File Permissions

Windows doesn't use the Unix permissions model. Instead Windows permissions work via a customisable set of ACLs (**Access Control Lists**). You assign permissions to a **principal**. A principal can be a specific user or, more commonly, a group of users. Unlike the UNIX model, you can grant permissions on a directory to any number of principals. Windows files and directories by default inherit the permissions set on their parent directory.

Windows defines many more permissions than UNIX, and separates them into basic and advanced permissions. The basic permissions are:

1. Full Control
2. Modify
3. Read & Execute
4. Read
5. Write

The advanced permissions are:

1. ListDirectory
2. ReadData
3. WriteData
4. CreateFiles
5. CreateDirectories
6. AppendData
7. ReadExtendedAttributes
8. WriteExtendedAttributes
9. Traverse
10. ExecuteFile
11. DeleteSubdirectoriesAndFiles
12. ReadAttributes
13. WriteAttributes
14. Delete
15. ReadPermissions
16. ChangePermissions
17. TakeOwnership
18. Synchronize

A flexible number of principals and granular permissions make the Windows permissions model more powerful, and slightly more intimidating, to work with than the UNIX model.

You can view the permissions on a file or directory using the `Get-Acl` cmdlet, and set them using the `Set-Acl` cmdlet. You can also use the older `icacls` tool, which may be helpful if working via command prompt.

Making Web Requests

You can make network requests from the command line. As a DevOps engineer, you'll not need to do this manually very often (after all, we have browsers for that), but it'll be a common part of application deployment scripts and general infrastructure and tooling automation.

There are two open-source CLI tools widely used on UNIX-like systems to make web requests: `curl` and `wget`. `curl` is arguably the more powerful and flexible of the two (it has a much wider feature set, including a broader range of protocols), but either will suffice for making simple HTTP, HTTPS and FTP requests. At the most basic, either tool will issue a GET request and save the response to file:

```
$ curl -o python_curl.html https://www.python.org/
$ wget -O python_wget.html https://www.python.org/
```

For scripting web requests on Windows, you can also use `curl`! It's built into PowerShell, and has recently been added to command prompt as well. If you prefer a first-party cmdlet based PowerShell option, [Invoke-WebRequest](https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/invoke-webrequest?view=powershell-7) (<https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/invoke-webrequest?view=powershell-7>) should suit your needs.

Text Search & Manipulation

We're steadily building up a toolkit of core shell commands, and soon we'll learn how to combine these commands using pipes and scripts to automate meaningful tasks. But first, we need to introduce some common tools for automated text search and processing. These are parts of the standard UNIX toolkit.

Keep in mind that these are powerful tools about which a great deal has been written! You'll find additional learning resources on each of these in the additional material section. Used correctly, these tools can help you automate the vast majority of text processing tasks via the shell.

Finding Documentation

Command line programs can be extremely complicated, with dozens of options and sometimes entire programming languages embedded within them. How do we go about discovering these options, and learning the correct syntax for invoking a command?

One valid option is the internet! Websites like stack overflow (<https://stackoverflow.com/questions>) are full of helpful tips and tricks. However, you'll find that online instructions often relate to a different version of a tool. For example, many CLI utilities differ subtly between different Linux distributions. To work around this, you can use local documentation included with your CLI tools to get correct documentation for your specific versions.

Unix Shells

Unix tools and commands use 'man' pages (short for 'manual') to distribute documentation alongside tools. You

can browse the man pages for a particular command using the man tool. For example, try running `man git`.

You can even run `man man` to find out how to get more from your documentation. Man pages are often extremely long and detailed, and tools exist to summarise them or otherwise make them more readable. A popular option for shorter descriptions is tldr (<https://tldr.sh/>).

PowerShell

PowerShell includes the `Get-Help` cmdlet to look up documentation. You can also look up online documentation via `Get-Help -Online <cmdlet>`, which will open the results in your browser where possible.

It's sometimes necessary to manually update your local PowerShell help documentation. Use the `Update-Help` cmdlet for this.

grep

Grep is a command line text search tool that uses **regular expressions** (also known as regexp, or regex) to search for text patterns in files. It's a fact of life that Unix tools can have odd names! The name 'grep' comes from the earlier ed interactive line processor tool, where a command of the form `g/re/p` (where re is replaced with the regex to match) performs the same functionality as

grep. In fact, grep is a slice of the ed program, extracted and re-distributed as a standalone automated tool.

Regex lets you match a text pattern. For example, imagine we have a log file created by the chess application written earlier in this course. `chess.log` contains log entries with a structure shown below. This is typical of a simple log file - structured rows of text with a timestamp, and some information about what's happened.

```
...
2020-02-02 10:42:48,828 Player.WHITE Pawn to Square(row=3, col=6)
2020-02-02 10:42:53,813 Player.BLACK Pawn to Square(row=5, col=5)
2020-02-02 10:42:58,391 Player.WHITE Knight to Square(row=2, col=5)
2020-02-02 10:42:59,730 Player.BLACK Pawn to Square(row=4, col=4)
2020-02-02 10:43:05,468 Player.BLACK King to Square(row=6, col=5)
...
```

Using `grep`, you can quickly parse a file to find text matching certain patterns. The basic usage is `grep [pattern] [file]`. The `pattern` parameter is a regex pattern, and `file` is the file to look at.

Regex is a complicated topic, with lots of edge cases and special characters. Very few people master the topic - instead it's typical that users learn features as and when needed. For now we'll introduce some basic concepts, but note that this is only a small fragment of all regex functionality. We'll start with a few examples to give a flavour of what you can do:

```
# find rows that include 'King'
$ grep 'King' chess.log

# find rows relating to columns 3, 4 or 5
$ grep 'col=[3-5]' chess.log

# find moves made by the black player at 10:42:xx am.
# The -E flag switches to extended grep syntax.
$ grep -E '10:42:[0-9]{2}.*BLACK' chess.log

# find all Queen moves across all log files in the current directory.
$ grep 'Queen' *.log
```

These examples highlight some basic regex features. Most letters and numbers match themselves, i.e. the regex pattern `king` matches the strings `'king'`, `'kingmaker'`, `'speaking'`. Regex also contains many special characters used to construct more complex matches. These special characters are known as **regex metacharacters**. For example, the special characters `.` and `*` combined are extremely common. This regex means "match anything, zero-or-more times" and is often used as part of a larger regex that matches elsewhere on more specific fragments.

Regex	Description
.	Matches any single character
[ABC]	Matches any of A, B or C
[0-9]	Matches any digit in the range 0-9
{2}	Matches the preceding regex character twice
\	Escapes the following character, making it lose any special meaning
*	Matches the previous character zero-or-more times
+	Matches the previous character one-or-more times (egrep only)
?	Matches the previous character zero-or-one times (egrep only)
^	Start of line
\$	End of line

Not all regex implementations are the same! Regex functionality and syntax varies between modern programming languages and tools. There are three 'standards' you'll find (listed below), but bear in mind that individual implementations can and do add their own features.

- 1) POSIX basic
- 2) POSIX extended
- 3) Perl 5 (PCRE)

There's an important difference between basic and extended POSIX regular expressions. The extended grep syntax supports more metacharacters (for example the `?`, `+`, `{`, `}` and `|` special characters are part of the extended syntax only), and used in modern tools and programming languages. Using `grep` can be very confusing if you're not sure which syntax is being used!

Historically, `grep` used the basic syntax and another program, `egrep`, used the extended syntax. You can still do this, or simply run

`grep -E` to use the extended syntax. The extended syntax is usually preferred, but there are times when you really do want to match on lots of `?` and `|` characters. In that case, the basic syntax will require far fewer escape characters in your regex pattern!

You'll find regular expressions in almost all programming languages, and supported natively by many other programs (both CLI and GUI-based). You should be aware which syntax is used, and any other dialect-specific details. For example, the `sed` tool is like `grep` - you'll need to pass a `-E` flag to use extended regex syntax. `awk` on the other hand is based on the extended syntax by default.

Perl regex syntax is similar, but not the same as, POSIX extended! Many programming language regex implementations (including Python's) are based on perl's regex syntax.

sed

sed is a stream editor (hence the name) and, like grep, developed from the interactive line editor ed. That should give a hint to its purpose - it's an automated text processor that operates line-by-line on its input.

You can think of sed as an expansion of grep. grep searches lines of text, while sed can search *and manipulate* lines of text. It's a convenient tool for quickly creating a modified copy of a file, extracting key data, or automating repetitive tasks that you might otherwise do manually in a text editor. sed applies one-or-more commands sequentially to each line. Commands can modify, delete or totally replace lines, and can use line numbers or regex matches to limit commands to specific lines. More advanced use-cases perform complex multi-line processing, although be warned - there are often better tools for that use case!

To give a flavour, here's a couple of examples. Keep in mind that sed is non-destructive - it processes a file, and prints a modified version to screen. The source file is unchanged.

```
# 1. capitalise Python
sed 's/python/Python/g' my_file

# 2. Delete blank lines
sed '/^$/d' my_file
```

```
# 3. Extract code blocks from
markdown
sed -n '/^```/,/```/p' my_file
```

```
# 4. Print headings from a markdown
file (could be done with grep)
sed -nE '/^#\+/p' my_file
```

The interesting part in these examples is the command string (e.g. `'/^$/d '`) passed to sed. The command string gives sed instructions on how to process the input text (in this case, the contents of 'my_file'). You'll also notice the latter examples use a couple of **flags** (e.g. `-n`). Flags are a standard convention for toggling on or off certain behaviours in a command line tool.

The `-n` flag tells sed to **not** output text unless explicitly instructed by the `p` command. By default sed prints every line of text it operates on. The `n` flag is useful when using sed to filter lines out of a file, as in the examples above. The `E` flag instructs sed to use the extended POSIX regex syntax when parsing the command. In the example above, this is necessary for the `+` character to be interpreted as a metacharacter. Remember, sed uses basic POSIX regex syntax by default. Flags can be combined: `sed -nE` is a contraction of `sed -n -E`.

You may not understand the command syntax yet, but these commands

demonstrate a few key features that make sed surprisingly powerful. It can:

1. **Perform text replacement (using regex).** The first example uses the substitute command (s/) to replace 'python' with 'Python' everywhere in the file. In this case the replacement string is hardcoded, but it can also handle regex.
2. **Filter lines.** Using the -n flag, sed can selectively include or exclude certain lines. By default, sed prints all lines from the source file. Example 2 modifies this using the /d command to delete lines that match the regex ^\$ (i.e. an empty line). Example 3 shows how to invert this behaviour using the -n flag - in this case, only rows that match the regex are passed to the print command (p) and included in the output.
3. **Operate on multiple lines.** Example 3 uses two comma-separated regular expressions to define the start and end of a block of lines. When sed reaches a line matching the first regex (^```) it starts processing lines, and continues until it reaches a line matching the second regex (^```). In this case processing is done by the p command - it simply prints the line.

This combination of multi-line selection, and the flexible use of regex to both select lines and change them is extremely powerful. In these examples we've shown using a single sed command written in-line. Sed can also be run with multiple commands processed sequentially to build up complex operations. You can pass in multiple commands using the -e parameter or, preferably, by writing them in a file and passing the file path via the -f parameter.

awk

Awk is the final, and most complex, text processing tool we'll introduce here. Like sed, awk is all about automatically processing text files. It moves beyond sed's functionality by providing a domain-specific programming language and user-accessible memory for storing state between lines. This makes it much more flexible, but also more complicated.

A minimal AWK script is shown below. You pass awk a script and a text stream to operate on (e.g. a file). awk then runs the script for each line stream and sends the output to the terminal. The overall architecture is very similar to sed.

```
awk '{ print $0 }' my_file
```

This simply prints each line from the file (not very useful!), but shows a couple of key features.

1. awk scripts are wrapped in curly braces
2. The print function sends data to the output
3. The magic variable \$0 holds the value of the current line being processed

Passing an in-line script to awk rapidly becomes unhelpful, since awk scripts are usually multiple lines long. Instead you'll typically write them in a separate file and pass the filepath to awk instead, as shown in the next example.

Awk's particular strength is in processing structured data. It automatically splits input rows based on a separator (whitespace, by default) and provides the values to your script as variables \$1, \$2 etc. The total number of fields for the current row is available in the variable NF. This makes it ideal for handling tabulated data (e.g. CSV or TSV files), as well as structured text content such as log files.

awk's programming language supports a variety of flow control operations, can store variables (i.e. it has memory), and has built-in support for applying line processing rules based on regex matches of the current row. Together, these features can build remarkably powerful little scripts.

Here's an example, imagine you have a data file (my_file) that contains rows of text and rows of numeric data. You don't want the text, but need to sum all the numeric values. It's a constructed example, but let's up demonstrate what awk is all about. Here's an example file:

```
# contents of 'my_file'
Some header text.
Data follows.
1 2 3
onetwothree
4 5 6
10
```

You could solve the problem with the awk script below. We've not discussed awk syntax, but it should look familiar. There's a pre-processed step (BEGIN), a post-processing step (END) and two line-processing rules that are applied to rows that match the corresponding regular expressions. In awk and sed, a regular expression is bounded by forward slash characters (/)

```
# program.awk
BEGIN { total = 0 }
 /^[A-Za-z]/ {
    print "Skipping text line"
}
 /^[0-9]+/ {
    for (i = 1; i <= NF; i++) {
        total += $i
    }
    print "Processed values:", $0
}
END { print "Total", total }
```

Finally, here's how to invoke the script and see the output:

```
bash $ awk -f program.awk my_file
Skipping text line
Skipping text line
Processed values: 1 2 3
Skipping text line
Processed values: 4 5 6
Processed values: 10
Total 31
```

If you'd like to know more about the full range of functionality available in awk, you can look into the documentation, online guides and the [POSIX specification \(https://pubs.opengroup.org/onlinepubs/9699919799/utilities/awk.html\)](https://pubs.opengroup.org/onlinepubs/9699919799/utilities/awk.html) for the awk programming language.

Think carefully before using awk for file processing with complex logic. While awk provides some nice features for free, and is brilliant for simple manipulations, its syntax is limited and it shouldn't be used for implementing complex logic. You would be better served performing such operations in a general purpose programming language such as Python.

Text Processing in PowerShell

Text Processing in PowerShell works very differently to Unix-like shells. When writing bash commands (for example), the author will look to powerful external programs (e.g. grep, sed, awk). PowerShell has few equivalents, and instead provides built-in components that can be combined to perform similar operations.

PowerShell has powerful [comparison operators](https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_comparison_operators) (https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_comparison_operators) and cmdlets that can perform many common text processing operations including [regex matching](https://docs.microsoft.com/en-gb/powershell/module/microsoft.powershell.core/about/about_regular_expressions) (https://docs.microsoft.com/en-gb/powershell/module/microsoft.powershell.core/about/about_regular_expressions) and text replacement. These can be combined with cmdlets such as Where-Object, ForEach-Object and Select-Object to build powerful commands and scripts.

The Select-String cmdlet works similarly to grep. It can read lines of text from a file (or be piped input), and return lines based on a regular expression. For example, you could use it to remove blank lines from a text file using Select-String

```
# Find empty lines by matching a regex
$ Get-Content my_file | Select-String -Pattern '^$' -NotMatch
```

In PowerShell, every variable is a .NET object. That means we could solve the same problem by inspecting the length property on the string object, avoiding the need for this particular regex:

```
# Find empty lines by inspecting their length
$ Get-Content my_file | Where-Object {$_.Length -gt 0}
```

In the example above, we use the Where-Object cmdlet to filter the file contents based on the line length. The Where-Object cmdlet can be configured in many ways - here we've used a PowerShell **script block** { ... } to describe the operation to test on each object. In a script block the variable \$_ holds the value of the current input - in this case, an individual line from the file. We can use the -gt comparison operator to return include only rows with a length greater than zero.

PowerShell's regex matching could also be used to replace matches. The example below shows this using the -replace operator (which takes a regular expression to match), and writes the output to file.

```
# Capitalise all occurrences of "python" (a regex)
$ Get-Content my_file | ForEach-Object {$ _ -replace "python", "Python"} | Out-File Python.txt

# A shorter version
$ cat my_file | % {$ _ -replace "python", "Python"} > Python.txt
```

jq

The text processing tools we've discussed (Sed, awk and grep, along with built-in PowerShell tools) are designed to work with text data line-by-line. This arises from their origins in line-editing programs, and is suitable for many use-cases (e.g. CSV/TSV data files, written text, logs). However, those tools are not appropriate for processing complex multi-line structured data, for example XML and JSON.

Extracting values from JSON data is a common task. For users on UNIX-like environments, jq is a widely-used CLI tool that can help. The authors describe it as "like sed for JSON". It's a command-line JSON parser with a powerful query language. You can compose filters and create new JSON objects and arrays.

Let's try an example. Often you'll encounter JSON data in responses from web APIs. Let's use an example, pulling currency exchange. This example uses a file to store the results temporarily - we'll discuss this, and the use of pipes (|) in the next section.

```
# Get some JSON data
$ curl 'https://api.exchangeratesapi.io/history?base=GBP&start_at=2019-12-01&end_at=2019-12-31' > data.json
```

```
# Read and format the file entire with jq
$ cat data.json | jq '.'
{
  "start_at": "2019-12-01",
  "base": "GBP",
  "end_at": "2019-12-31",
  "rates": {
    "2019-12-17": {
      "CAD": 1.7333742389,
      "HKD": 10.2601831312,
      "ISK": 161.1837447491,
      "PHP": 66.5655826686,
      "DKK": 8.8180252041,
      "HUF": 389.0475291452,
      "CZK": 30.0243073583,
      "GBP": 1,
      ...
    }
    "2019-12-18": {
      ...
    }
  }
}
```

This is a typical JSON structure. There are many nested objects, and a lot of data that you're not interested in! Using jq, you can query this data structure and extract only the parts you need, as well as perform more complex re-processing of the data if required.

jq works through a series of filters connected by pipes. You can use filters to iterate through objects and arrays, construct and return new objects, and edit data. Filters can be composed together to build increasingly complex scripts. Take a look at the follow examples using JSON to query and manipulate JSON data:

```
# Basic: get the data for 2019-12-24
$ cat data.json | jq '.rates."2019-12-24"'
{
  "CAD": 1.704839068,
  "HKD": 10.0885038523,
  "ISK": 158.5353021641,
  "PHP": 65.8587913437,
  "DKK": 8.7348742591,
  "HUF": 387.8736861796,
  "CZK": 29.7955175196,
  "GBP": 1,
  "RON": 5.5873171758,
  "SEK": 12.2237031321,
  "IDR": 18116.2826043749,
  "INR": 92.3064782014,
  "BRL": 5.2898881134,
  "RUB": 80.4288403306,
  "HRK": 8.7048273766,
  "JPY": 141.6880034607,
  ...
}

# More complex: get the Canadian dollar exchange rate, keyed by date
$ cat data.json | jq '.rates | [keys[] as $k | {"\"($k)\" : .[$k] | .CAD}] | add'
{
  "2019-12-02": 1.7198244502,
  "2019-12-03": 1.7308685446,
  "2019-12-04": 1.7382106134,
  "2019-12-05": 1.7296081449,
  "2019-12-06": 1.7309035795,
  "2019-12-09": 1.7429776115,
  "2019-12-10": 1.7444363827,
  "2019-12-11": 1.7402813223,
  "2019-12-12": 1.7355723746,
  "2019-12-13": 1.7617473775,
  "2019-12-16": 1.7528022538,
  "2019-12-17": 1.7333742389,
  "2019-12-18": 1.7193580624,
  "2019-12-19": 1.7147626156,
  "2019-12-20": 1.7124969166,
  "2019-12-23": 1.7007747235,
  "2019-12-24": 1.704839068,
  "2019-12-27": 1.7140843416,
  "2019-12-30": 1.7159186931,
  "2019-12-31": 1.715796897
}
```


JSON processing in PowerShell

PowerShell contains `ConvertFrom-Json` and `ConvertTo-Json` cmdlets to transform data between JSON and .NET object representations. When parsing JSON data, PowerShell automatically creates a custom .NET object type for it. Once decoded, you can handle JSON data using native PowerShell features.

Our `jq` example extracting data for 2019-12-24 can be implemented in PowerShell:

```
$ Get-Content data.json |  
  ConvertFrom-Json |  
  Select-Object -ExpandProperty rates |  
  Select-Object -ExpandProperty '2019-12-24' |  
  ConvertTo-Json
```

Handling data via native PowerShell features is helpful - you don't need to learn a domain-specific language (DSL) as is common in Unix shell tools (e.g. `jq`), instead you can re-use common object operations. The disadvantage is that this approach is much more verbose, and less efficient to users familiar than a DSL-based alternative for experienced users.

Combining Tasks

You may have noticed that we're introducing lots of little tools that each do a single, clearly defined task then print the results to the terminal. This reflects a particularly relevant aspect of the 'Unix philosophy', summarised by Peter Salus in 'A Quarter-Century of Unix' (1994):

1. Write programs that do one thing and do it well
2. Write programs to work together
3. Write programs to handle text streams, because that is a universal interface

Many of the bash tools that are commonly used subscribe to these principles. This allows them to interoperate relatively seamlessly, and users can write command and scripts to pipe (|) data between multiple distinct programs, building their own tools to fit particular needs. You've already seen this in some code examples above. Let's show it off again:

```
echo 'lets try piping' | sed 's/try/have a go with/' | wc -w
```

Here the echo command produces a string 'lets try piping', which is manipulated by sed, before finally being passed to the wc program, which calculates a word count. The final result is sent to our terminal, because we haven't redirected the output from wc anywhere else! You use pipes to connect inputs and outputs together, combining individual programs into specific workflows.

Because everything in the unix shell communicates via text streams, it's also easy to source data from, and write data to, text files. The output from any command can be send be written to a file (>) or appended to an existing file >>. You can also use < to have a command read from a file, rather than standard input.

```
# Writing data to file (replacing the previous content of that file, if any)
$ echo 'hello world' > hello.txt

# Appending data to file (writing data to the end of the existing content
inside the file, if any)
$ echo 'hello again' >> hello.txt

# Reading from a file
$ cat < hello.txt
```

Standard Input, Output and Error

Let's dig a little deeper into how all this actually works.

Programs need ways of interfacing with the rest of the computer. For example, they need to connect with a keyboard to receive input, and know where to send their output (perhaps a file, or the terminal). Unix-like systems provide abstractions for this program IO, so that individual programs don't need to care about where their data has come from, or where it's going. Instead, CLI tools interact with their environment via three standard *text streams*:

- **Standard Input (STDIN).** Programs read input text from this stream.
- **Standard Output (STDOUT)** Programs write their normal output to this stream.
- **Standard Error (STDERR)** This is where errors go!

But what are these streams? Well, a text stream is a collection of lines of text, separated by newline characters. It's potentially infinite, and programs can read lines from a text stream (in the case of STDIN) or write new lines to a text stream (in the case of STDOUT and STDERR). This may sound a lot like interacting with files, and in fact the operating system exposes

them in the same way (using what's known as a *file descriptor*).

When you write `echo 'hello world' | wc -w`, here's what the pipe does:

- The `echo` command writes its input to STDOUT.
- The pipe (`|`) sends data from `echo`'s STDOUT and STDERR to `wc`'s STDIN stream.
- The STDOUT from `wc` isn't sent anywhere explicitly, so is printed to the terminal.

It's important to know about these standard streams, because you'll sometimes need to handle them explicitly. In the shell you'll usually refer to these streams by their file descriptor numbers, rather than names. Standard input is file descriptor 0, standard output is file descriptor 1 and standard error is file descriptor 2. You can handle them explicitly to fine-tune which outputs go where.

Here's a demonstration of the syntax. By default the output redirection operator (`>`) deals with standard output only. It is shorthand for `1>`, where the file descriptor is given explicitly. You can use other file descriptors to redirect other streams.

```
# Write output to a file (called 'output'), leaving errors to be printed to
the terminal (as happens by default)
$ program_a > output

# Write output to a file (called 'output') and errors to a separate file
(called 'error_file')
$ program_a > output 2> error_file

# Send standard output (file descriptor 1) and standard error (file
descriptor 2) to the same file (called 'all_output')
$ program_a > all_output 2>&1

# Suppress errors entirely by piping them to a special file that ignores
everything sent to it
$ program_a > output 2>/dev/null
```

To get your feet wet, try this practical example:

```
find / -type f -name '*.py' > results 2> errors
```

This command looks for python files recursively across your entire file system (it may take a while!). Your current user probably doesn't have read permissions on all directories, so there are going to be some permission errors as `find` traverses the file system. Running the `find` command without output redirects would print both `STDOUT` and `STDERR` to the terminal, which can be hard to process. By using stream redirection we can have one file that contains Python file paths and another that stores the errors encountered along the way.

If you want to learn more about stream redirection, take a look at the [bash manual](https://www.gnu.org/software/bash/manual/bash.html#Redirections) (<https://www.gnu.org/software/bash/manual/bash.html#Redirections>), which goes deeper into the syntax and tools available to you.

Executing programs

Let's take a step back and consider what is actually happening when you run a command like `echo`, `cd` or `find` from the shell. What are these commands, and how do they compare to a more fully-featured GUI application like VSCode?

Some commands are built-in to the shell, and others are external programs that the shell knows about because they exist on your **PATH** (we'll come back to this in a moment). From a user point-of-view, it doesn't really matter whether a command is built-in or provided as an external program, but you can find out using the `which` command. The results may vary depending on your system at shell (the examples below use `zsh` on a Mac).

```
$ which echo
echo: shell built-in command
/bin/echo
```

This tells me that the first occurrence of `echo` is built-in to the shell I'm using, but there's also a standalone `echo` program available at `/bin/echo`. My shell won't use that, but perhaps another shell doesn't have `echo` built-in, so a system-wide version is available.

But, wait a minute, what is `which`?

```
$ which which
which: shell built-in command
```

Built-in or not, all these commands are just programs that accept command line arguments. Some CLI programs are distributed with your system while others need to be installed via a package manager, but you will use them the same way. A common example of a routinely-used CLI program is `git`. You can also launch GUI-based applications from the command line. For example, VSCode, if installed correctly, can be invoked using the `code` command.

Environment Variables

Environment variables are process-wide variables that applications can use. We've already used them in various exercises, and we'll continue to explore their value when we discuss app deployment infrastructure. Environment variables can be managed via the command line. In `bash`, you can use the `export` keyword to define an environment

variable, and can access their values programmatically:

```
$ export MY_VAR=123
$ echo $MY_VAR
# Prints 123
```

The `PATH` environment variable is a particularly important one. It stores a list of directories where shells will look for programs you specify. You can run `git log` and it works because the shell can find the `git` program on your `PATH`. If `git` was not on the path, you'd need to provide a path to the program (`/usr/bin/git log`, in my case), or, preferably, add `git` to your `PATH`.

You'll often want to add new directories to `PATH` - perhaps you've installed a tool to its own directory, but want system-wide access via the command line. Fortunately, it's easy to modify environment variables by re-exporting an updated value:

```
export PATH="$PATH:~/my/app/bin"
```

This appends the `~/my/app/bin` directory to your `PATH`. `~` is your home directory; the shell will expand it to a full path. Path order is important - the first matching occurrence of a named executable is the one that will be used. Appending to `PATH` is safe: you'll never override a command that already exists. Sometimes you'll need to prepend a directory to `PATH` to ensure your version of a command takes precedence over a system-provided one.

Environment Variables Are Not Forever

Environment variables live in the process that defines them, and they're accessible from all child processes. But they don't last forever. If the process ends they are lost. If you define an environment variable in the shell, then quit the shell, the value is lost and you'll need to re-create it when you open a new shell.

To permanently set environment variables you'll need to add it to your **shell profile**. A shell profile is a script that runs automatically when your shell is invoked.

Shell	Profile Location
bash	~/.profile
zsh	~/.zprofile

Environment Variables on Windows

Windows uses a different architecture to persist updates to environment. They're stored in the Windows registry.

In PowerShell you can set a process-scoped environment variable similarly to UNIX:

```
$env:MY_VAR = 123
$env:MY_VAR
# Prints 123
```

You can define variables in your profile to automatically load them when a new shell process is launched. The `$profile` variable will show where PowerShell expects your profile to be.

As well as using profiles to set persistent variables, Windows supports user- and machine-scoped environment variables. To set these you'll need to use the Environment class in PowerShell.

For example, the script below updates a machine-scoped environment variable:

```
$path = [Environment]::GetEnvironmentVariable('Path', 'Machine')
$newpath = $path + ';C:\Path\To\My\Scripts'
[Environment]::SetEnvironmentVariable("Path", $newpath, 'Machine')
```

Windows supports several other methods for manipulating environment variables via the CLI. You can find out more [online](https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_environment_variables?view=powershell-7) (https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_environment_variables?view=powershell-7). It's also worth noting that, while this section is on shell tooling, Windows has good options for editing system and user variables via a GUI!

Chapter 2

Bash Scripting

In the previous section we introduced the shell, and went on a whistlestop tour of how to perform various common operations via a command line interface. We focused on individual commands, and introduced how to connect them together via pipes and redirecting text streams.

In this section we'll take the next step and introduce **shell scripting**. At its simplest, scripting is saving a sequence of shell commands to a file so they can be re-run again and again. Shell scripts also introduce syntax for variables, control flow, error handling and using functions to commonise common operations. This section focuses on scripting using bash.

Why are shell scripts useful?

Shell scripts are the backbone of automation. We've already discussed specific tools and technologies that help with automation in certain circumstances. For example, web APIs help us automate interactions with web applications and automated testing frameworks like pytest can help automate running unit tests. Shell scripts are broader than this, and are widely used to automate a variety of operations through software development, systems administration and DevOps:

- Machine configuration
- Development environment setup
- Application launch
- Test launch
- Scheduled backups

Why automate?

The benefits of shell script automation are similar to those we've already discussed in other areas. Automation can improve the accuracy of operations by reducing opportunities for human error. Automated tasks are also faster than doing the same task manually, and can save a great deal of time on often-repeated tasks.

Shell script automation is particularly suited for long-running or scheduled tasks - situations where human oversight is difficult or unfeasible. Automated tasks can often be scaled to run in parallel, and are best suited for tasks with little or no variation between re-runs.

Shell scripts or Python?

Python is often described as a 'scripting' language, meaning that a single `.py` file can be run directly by the Python interpreter, requiring no additional project boilerplate. This might sound superficially similar to a shell script, so why should you learn a new tool?

That's a very good question, and there are a few points worth making. Many businesses can and do use Python for general purpose scripting, although there are drawbacks. Let's discuss some key pros and cons.

Why Python is great for scripting

To be clear, everything you can do via a shell script could be done via Python (provided Python is available). You can even create and execute shell commands from Python if it suits your needs. In particular, Python has some key advantages over shell scripts:

1. **Python is portable.** A python script can run on any machine with an appropriate Python version installed. Mainstream shell scripting languages are shell-specific, and shells are usually closely tied to a single operating system, or family of operating systems.
2. **Python is readable.** Python's modern, language-like syntax is much clearer and easier to learn than most shell script syntaxes. It's also an extremely popular and widely-known language. It's easier to maintain Python scripts, particularly for applications programmers who may not often work with shell scripts.
3. **Python is scalable.** While python can be used to create a simple procedural

script, it is also a general-purpose programming language. If your script becomes increasingly complex, Python lets you refactor to use classes and methods, and even re-architect your overgrown script into an application of its own. Such options are far more limited when writing shell scripts.

4. **Python has a standard library.**

Python's standard library is huge, and provides Python scripts with an impressive toolbox to help with almost any task. As a script writer, you can usually assume that the standard library is available and ready for use. Key packages include `os` (<https://docs.python.org/3/library/os.html>), `pathlib` (<https://docs.python.org/3/library/pathlib.html>) and `subprocess` (<https://docs.python.org/3/library/subprocess.html>), the latter of which lets you run arbitrary shell commands in subprocesses.

Why shell scripts are still important

There are still a few reasons shell scripts are preferred.

- **Python isn't available everywhere.** To use Python for scripting you need to know that the expected version of Python, and the standard library, are

available on every machine you might want to run your script on. In contrast, you can be sure that bash will be available on all UNIX-like operating systems. In other circumstances, it might simply not be possible to run the Python interpreter for a particular platform. Perhaps you can't compile Python itself for that architecture or, more likely, you can't afford the disk space that a Python installation requires. Or maybe the tooling on a remote server is managed by a third-party, and they don't provide Python.

- **Python is more verbose.** Python is a general purpose programming language, while shell scripting languages are optimised for their particular use case. This means Python lacks some of the clean, concise syntax for common shell operations. Generally speaking, this is offset by Python's increased readability and flexibility for complex scripts, but it can feel like over-engineering to implement a one-line shell operation in 10 lines of Python!
- **Interacting with other programs.** The CLI is the primary external integration point for many modern DevOps tools, which makes shell scripting an excellent choice for automation. Python is more suited for automation where the integration can be done via a well-supported Python library or web API. These options may not exist, or have limited functionality compared to a CLI alternative. If you need to run CLI programs from Python, you can use the `subprocess.run(...)` method, but it's not nearly as neat as using a shell script.

Syntax

Let's look at how to write shell scripts using bash. Bash shell scripting is a programming language, just like any other. It has variables, keywords, and functions. You'll find some aspects of it familiar from Python, and in other ways it is wildly different (and sometimes a little obscure). The basic blocks of a shell script are the components we've already talked about: calls to external programs (e.g. git) or to built-ins (e.g. echo).

Bash scripts generally execute one command per line, and are mostly whitespace insensitive (exceptions will be noted when they crop up). You can include multiple commands on a single line by separating them with semi-colons (;).

Shebang

The **shebang** is a line at the start of a script that begins `#!`. The shebang says which executable, or interpreter, will be used to execute the script. *The shebang is essential*. A shebang lets a script writer ensure that their script is run from the appropriate interpreter, and it means script users don't need to worry about invoking a script using the wrong interpreter.

Consider the script below. It recursively renames JPEG images files in the current directory to standardise their file extensions

(to .jpg). The script uses some globbing patterns and the `zmv` built-in that are specific to `zsh`. The shebang at the top tells the user that the script must be executed using `zsh`.

```
#!/usr/bin/env zsh
autoload zmv

echo 'Standardising jpg file
extensions...'
zmv -n './(*/)(*)(#i)jp(|e)g'
'$1$2.jpg'
echo 'Complete!'
```

That means that the script can be invoked as shown below, and it will be run using `zsh` regardless of the shell used to invoke the script (e.g. bash). shell \$ cd path/to/my/photos \$./<script_file_name>

What happens if you run a script without a shebang? It's not recommended, and the behaviour will depend on the parent shell. For example, `bash` will run the script itself while `zsh` will revert to using `sh`. In both cases, the script above would fail.

Absolute Shebangs

Above, we used a shebang of the form `/usr/bin/env <exec_name>`. In the example `<exec_name>` was `zsh` because it is a Z-shell script, but it could be `sh`, `python3`, or something else, depending on the content of the file.

This shebang format is the one you should follow where possible. The appropriate interpreter is found from the user's PATH environment variable, which makes it portable across multiple systems, and flexible enough to handle setups like Python's virtual environments (where even the user may not know where their Python executable is!).

Alternatively, you can use an **absolute shebang** that gives a full path to the interpreter, as shown below.

```
#!/usr/bin/bash
...
```

This approach is generally discouraged, as bash may exist somewhere else on a particular system (i.e. your script has a hardcoded absolute path, and that's not very portable). However it can be necessary if you need to bypass the PATH variable on the client system and ensure the script is run by the same interpreter everywhere.

Variables

Like any programming language, shell scripts support variables. In bash (and related shells), you assign variables with `[name]=[value]` (without any surrounding whitespace). By convention, environment variables and shell variables have uppercase names, while regular user-created variables

should be lowercase. Variable names are case-sensitive, and valid characters are: a-z, 0-9, and the underscore (_).

You access a variable's value by prefixing the variable name with \$. You can also add braces (`${variable_name}`) for clarity, or where required to ensure the name is unambiguous.

```
#!/usr/bin/env bash
var=123

# Access the variable using $
echo $var

# Variables are always strings
echo $var + 3

# Variables are interpolated in
double-quoted strings
echo "Variable = $var (double
quotes)"

# ...but not single-quoted strings
echo 'Variable = $var (single
quotes) '

# Use curly brackets where necessary
to bound the variable name
echo "Variable = ${var}456"
```

Remember, the dollar size (\$) is needed to get the value of a variable. This is different from a language like Python. In shell scripts, the name of a variable is just a string and has no special meaning. The dollar instructs the shell to look up a value associated with that name.

```
$ var1=hello

# Incorrect
$ var2=var1
$ echo $var2
var1

# Correct
$ var3=$var1
$ echo $var3
hello
```

All the variables we've shown so far are read-write. Bash supports readonly variables.

```
readonly var=1
var=2 # error
```

As well as these **scalar variables**, bash supports **array variables**. Bash arrays are zero-indexed and are assigned using curly brackets.

```
$ my_array[0]=This
$ my_array[1]=is
$ my_array[2]=an
$ my_array[3]=array
```

You can also use the following shorthand syntax to define an array:

```
$ my_array=("This" "is" "an" "array")
```

After you have set any array variable, you access it as follows:

```
$ echo "First index: ${my_array[0]}"
First index: This
$ echo "Third index: ${my_array[2]}"
Third index: an
$ echo "All values: ${my_array[@]}"
All values: This is an array
```

Bash Arithmetic

Bash supports integer arithmetic, but that arithmetic needs to be expressed a little differently so it behaves properly alongside other bash syntax. For example, you can't just write:

```
a=1; b=2
echo $a+$b # Prints "1+2"
```

Instead, arithmetic needs to be wrapped up in double parentheses to be calculated correctly. The dollar sign (\$) is necessary if you want to return a value.

```
echo $((a + b)) # Prints 3
```

There are a couple of other ways to do integer calculations in bash, although this is the POSIX standard method and is generally preferred. Bash does not support floating point arithmetic (other shells, such as zsh, do). However, if you need to do any non-trivial maths as part of a shell script you may want to look into delegating that work to other programs (e.g. Python).

Array variables are particularly useful when combined with loop structures, which we'll introduce shortly.

Variable Scoping

Variables declared in bash are global by default. Global variables are accessible everywhere within the parent shell. This means that variables declared in functions or scripts are not limited to those scopes, and can also be read (and modified) outside their intended usage.

It's good practice to use local variables where possible. A local variable is declared with the `local` keyword, and is only accessible within the function scope that defines it. We'll introduce functions properly in a moment, but for now note that, in the example below, the value of `variable_two` is *not* accessible outside the function `set_variables`.

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

function set_variables {
    variable_one=global
    local variable_two=local
}

set_variables
echo $variable_one # prints 'global'
echo $variable_two # undefined
```

Remember that global variables are scoped to their parent shell. Be wary of declaring variables within pipes. Pipes create sub-shells. Subshells inherit the environment (i.e. variables) of their parent shell, but modifications to the environment within a pipe are not exported to the parent scope. Here's a minimal example:

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

parent_var=parent
ls | (echo $parent_var; subshell_var=child)
echo $subshell_var # undefined
```

Environment Variables

We discussed environment variables in the previous section. Environment variables are process-scoped variables, and can be set by using the `export` keyword before a variable definition. Environment variables are always strings, and by convention their names are all uppercase.

Other than their scoping behaviour, environment variables can be treated just like bash variables.

Conditionals

Conditional logic in bash is done via the `if` command. The reverse `fi` command signals the end of the conditional block, and the `then` keyword indicates the start of the conditionally-executed logic:

```
if BOOLEAN_EXPRESSION
then
    echo 'Expression was true!'
fi
```

If expressions can be nested, and bash also supports `elif` and `else` clauses within a conditional block for more complex conditionals.

```
if EXPRESSION_A; then
    echo 'Expression A was true!'
elif EXPRESSION_B; then
    echo 'Expression B was true!'
elif EXPRESSION_C; then
    echo 'Expression C was true!'
else EXPRESSION_C
    echo 'None of them were true'
fi
```

Logical expressions are enclosed in square brackets (one or two - they behave slightly

Syntax	Meaning
<code>[[-e file]]</code>	True if 'file' exists
<code>[[-z string]]</code>	True if 'string' has length zero
<code>[[A -gt B]]</code>	True if A is greater than B
<code>[[A == B]]</code>	True if A equals B
<code>[[A != B]]</code>	True if A does not equal B

differently) and bash supports a variety of comparison operators and built-in checks. Here are some common examples:

There are some short-cuts you can take when writing small conditionals. These can help with readability.

```
[[ BOOLEAN_EXPRESSION ]] && ( echo 'The expression is True' )
[[ BOOLEAN_EXPRESSION ]] || ( echo 'The expression is False' )
```

Single or Double Brackets?

Bash, Zsh and ksh support double square brackets for these test expressions, whereas the older Bourne shell (and the POSIX standard) uses single square brackets. You'll see both, and you can use either in bash. They behave subtly differently (you can read about the details [here](http://mywiki.woledge.org/BashFAQ/031)) (<http://mywiki.woledge.org/BashFAQ/031>). Without diving into the details (some are quite obscure), we recommend using double square brackets where possible, as in the example above.

You'll need to use single square brackets if your priority is to maximise portability.

Bash Brackets

Bash uses brackets for many different purposes! Here's a quick summary of some common ones:

Brackets	Meaning
<code>(...)</code>	Runs commands in a subshell
<code>((...))</code>	Perform integer arithmetic
<code>[...]</code>	An alias for the <code>test</code> built-in command
<code>[[...]]</code>	Bash's expanded test command
<code>{ ... }</code>	Function definitions, Grouping commands, text expansion

Many of these are also used in conjunction with modifier characters (e.g. `$`) to perform additional tasks. There's an excellent tour of bash bracket syntax available online (<https://www.assertnotmagic.com/2018/06/20/bash-brackets-quick-reference/>).

Loops

Bash features for loops, while loops and until loops.

For loops can be used to repeat some operation over a range of values, or iterate over the response returned from some other command or function:

```
for i in {1..5}; do
    echo $i
done

for file in $( ls ); do
    echo $file
done
```

We've not seen the `$(command)` syntax before. This is a **command substitution**. When run, the expression is replaced by the standard output returned by running the command inside the brackets. There's an alternative syntax using backticks: ``command``.

Remember, UNIX shells exchange data between commands as text. The for loop splits the response from `ls` on whitespace characters, then iterates over each word.

Bash also supports while and until loops. They're very similar. The while loop executes the nested commands while a condition is true, and the until loop executes the nested commands until a condition is true.

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

count=0

while [ $count -lt 3 ]; do
    echo $((count++))
done

until [ $count == -2 ]; do
    echo $((count--))
done

# prints:
# 0
# 1
# 2
# 3
# 2
# 1
# 0
# -1
```


Case statements

Bash supports case statements using the `case ... esac` syntax. A simple case statement is shown below. Each case statement contains a pattern to match and a series of commands to execute on a successful match (the two are separated by an unmatched closing bracket `)`). Each case can contain multiple commands, and end with a double semi-colon `;;`.

Below is an example case statement with some simple pattern matching. The call to `tr` converts the input letter to lowercase.

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

echo "Give me a letter: "
read letter

case $(echo "$letter" | tr '[:upper:]' '[:lower:]') in
  b) echo "That's the second letter of the alphabet!";;
  c) echo "And the third...";;
  a | e | i | o | u) echo "That's a vowel";;
  *) echo "I don't know that one";;
esac
```

These are just the basics. Pattern matching in bash is rather powerful, and there's a variety of neat tricks at your disposal. If you're interested in learning more check out the GNU documentation on [conditional constructs](https://www.gnu.org/software/bash/manual/html_node/Conditional-Constructs.html) (https://www.gnu.org/software/bash/manual/html_node/Conditional-Constructs.html) and [pattern matching](https://www.gnu.org/software/bash/manual/html_node/Pattern-Matching.html#Pattern-Matching) (https://www.gnu.org/software/bash/manual/html_node/Pattern-Matching.html#Pattern-Matching).

Functions

As your scripts grow, you'll want to refactor code (just like you would when writing Python). Bash supports functions, and the syntax is familiar to most programmers. You can define a function in one of two ways - they are equivalent:

```
function my_function {  
    commands...  
}  
  
my_identical_function() {  
    commands...  
}
```

Unlike most programming languages, you don't include function arguments in the function declaration. Instead you just access them when needed. Function arguments can be accessed via the variables \$1, \$2 etc. You can also access all the arguments using the special variable \$@, which is particularly helpful if you want to loop over them:

```
function print_a_thing() {  
    echo $1  
}  
  
function print_all_the_things() {  
    for arg in $@; do  
        echo $arg  
    done  
}  
  
$ print_a_thing "dogs" "cats" "chickens"  
# Prints: dogs  
  
$ print_all_the_things "dogs" "cats" "chickens"  
# Prints: dogs cats chickens
```

If you're a frequent shell user it's likely there are commands, or collections of commands, that you re-run pretty frequently. You can save yourself some time by defining them as functions in your ~/.bashrc or ~/.zshrc files. By doing so, your custom functions will be automatically loaded every time you start a new terminal session.

Taking User Input

Script Parameters

Often you'll want to run a script with custom arguments - varying a parameter or two each time. These work similarly to the functional arguments discussed previously.

```
$ ./my_script.sh parameter_a parameter_b
```

You can access these **positional parameters** passed to your script using the variables \$1, \$2 etc. \$0 contains the name of your script. In the example above \$1 will contain the string "parameter_a" and \$2 the string "parameter_b". The shell once again sets some useful additional variables arguments: \$# gives the number of command line arguments, and \$@ gives them all together. (Note: \$# and \$@ do *not* include the name of your script in their output.)

Built-in commands can help you parse positional parameters. This is particularly useful for more complex scripts that might take a combination of optional input flags and named arguments. Accessing those by positions would quickly get complicated! Fortunately, the `getopts` and `shift` built-ins can help.

```
$ ./my_script.sh -d 2 -r 5 -v
```

Consider the script above. The script takes two options with arguments (`-d` and `-r`), plus one option that takes no arguments, and is just a simple flag (`-v`). This is a common convention, both in shell scripts and external programs. It's much more readable than relying on positional arguments alone. But how should the script parse these options? It'd be a pain to do it manually by parsing \$@, particularly considering that the options may appear in different orders, or might be optional. Bash includes a built-in `getopts`, which can help parse single-character options:

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

duration=
retries=
verbose=

while getopts "d:r:v" option; do
    case "${option}" in
        d)
            duration=${OPTARG}
            ;;
        r)
            retries=${OPTARG}
            ;;
        v)
            verbose=1
            ;;
    esac
done

echo "Duration: $duration"
echo "Retries: $retries"
echo "Verbose: $verbose"
```

There are some important limitations here. Notably, `getopts` does not support long-format option names (e.g. `-r` vs `--retries`). To do so in a reliable cross-platform manner will require more custom script and validation, although this is a common operation and template scripts are widely available. Do not confuse the `getopts` built-in with the similarly-named `getopt` external tool that performs a similar function, but is not standard across all operating systems.

CLI Interaction

Sometimes it can be clearer, and more user-friendly, to have a script ask questions rather than rely on correct configuration of (potentially many) script parameters. In bash, you can accept input directly from script users using the `read` built-in. This command reads a line of text from standard input, and saves it to one-or-more variable(s).

```
#!/usr/bin/env bash
echo "What's your name?"
read name
echo "Hi $name!"
```

A word of warning! Scripts that require back-and-forth interaction with the user are much harder to automate than those configured via parameters or environment variables alone. For example, you should avoid using `read` in a script that might be run on an automated schedule. On the other hand, it can be a useful tool when writing scripts that perform one-time configuration jobs for a user (for example, configuring a local development environment).

Exit Codes

Shell scripts don't have errors or exceptions like you'd find in Python or other object-oriented languages. Instead, scripts return an exit code. Typically 0 means success and any non-zero value (1-255) means that something went wrong, and some further information should be written to `STDERR`.

You can abort a script and send an exit code using the `exit` built-in:

```
if ! [[ -f $1 ]]; then
    echo 'File not found!' >&2
    exit 1
else
    echo 'File exists'
    exit 0
fi
```

Shell Options

There are a number of different **shell options**, also known as **flags**, that you can apply to bash scripts. These options affect the way in which the script will run, and they can be set in-line or when the script is called. It's helpful to know what sort of things options can accomplish - using an appropriate flag can often save you a lot of headaches when trying to implement specific scripts!

There are quite a few options, and we won't go through them all here. To find out more about the available options, take a look at the relevant [GNU bash documentation \(https://www.gnu.org/software/bash/manual/html_node/The-Set-Builtin.html\)](https://www.gnu.org/software/bash/manual/html_node/The-Set-Builtin.html).

You can include shell options in the shebang (`/usr/bin/env bash <options>`) or by using the `set` built-in function. The `set` function is useful for modifying the current shell options while the script is running.

errexit (-e)

The most commonly used option is called 'errexit'. By default, bash scripts won't stop if there's an error (by error, we mean that a command returned a non-zero exit code). Instead, the shell will continue on with the rest of the script. Often this is not the desired behaviour, particularly for scripts written for DevOps automation.

We can tell bash to immediately abort a script if it encounters an error by enabling 'errexit'. There's a few way to do this. Options can be set or unset using `-o <flag-name>` or `+o <flag-name>` parameters respectively. These parameters can be passed to the interpreter directly (i.e. in the shebang), or using the `set` built-in from within your script.

Most options also have their own shorthand letter code (known as a flag). For example, running `set -e` is equivalent to running `set -o errexit`.

verbose (-v)

The verbose option (`-v`) is a great help for debugging scripts. This option instructs the shell to print every statement before running it. You can also use the `xtrace` option (`-x`) which works similarly, but also performs variable substitution on the printed output.

noclobber (-C)

By default, redirecting command output to a file will override (aka 'clobber') any existing file content. Noclobber helps reduce the risk of overwriting existing data. It's not a guarantee however! Noclobber only modifies the behaviour of stream redirection by the shell. It won't affect any file-processing done by external programs.

Scheduled Jobs using Cron

It's very common to have a task that needs to be done repeatedly, at particular times. On Unix-like systems you can use cron jobs to make this possible. Cron jobs allow you to ask the OS to run a specified shell command or script at particular times. For example, you might use cronjobs to schedule database backups, clear out temporary files or produce periodic system monitoring reports.

Cron jobs are defined per-user, and are managed via the `crontab` command line tool. They have a particular schedule syntax:

```
$ Minute(0-59) Hour(0-24) Day_of_month(1-31) Month(1-12) Day_of_week(0-6) Command_to_execute
```

Here are some examples of valid cron schedules:

```
# Run every minute
* * * * * <command-to-execute>

# every 5th minute
*/5 * * * * <command-to-execute>

# every hour at minute 30
30 * * * * <command-to-execute>

# every hour at minute 0, 5 and 10
0,5,10 * * * * <command-to-execute>

# every 4 hours
0 */4 * * * <command-to-execute>

# every day at 1 am
0 1 * * * <command-to-execute>

# every weekday
0 0 * * 1-5 <command-to-execute>
```

The command `crontab -l` will show your current cron schedule, and `crontab -e` will open the cron schedule for editing using the editor defined by your `VISUAL` or `EDITOR` environment variables. By default this will be `vim`. You can then edit existing scheduled jobs, or add new ones.

There are also some predefined schedule definitions:

Symbol	Description	Equivalent
@yearly (or @annually)	Run once a year at midnight of 1 January	0 0 1 1 *
@monthly	Run once a month at midnight of the first day of the month	0 0 1 * *
@weekly	Run once a week at midnight on Sunday morning	0 0 * * 0
@daily (or @midnight)	Run once a day at midnight	0 0 * * *
@hourly	Run once an hour at the beginning of the hour	0 * * * *
@reboot	Run at startup	N/A

Writing cron schedules can be error prone. It's advisable to use an external tool to validate your schedules, particularly if you're new to the technology. One such example is crontab.guru (<https://crontab.guru/>).

Cron-like syntax is widely used for scheduling beyond the `crontab` utility. It's worth noting that not all implementations of the syntax are identical. Some even include an additional 'seconds' component to allow higher resolution scheduling than the original `cron` syntax, which is limited to running once per minute.

Finally, An Example

In this section we've covered the core toolkit available to you when writing shell scripts using `bash`. You've learnt how to use control flow, variables and functions to build larger, re-usable shell scripts and how to automate them as cronjobs.

You've also met some of the nuances of `bash` syntax - command substitution, pattern matching, parameter handling. Shell scripting languages are full of little tricks that help expert users more efficiently manage their file systems, manipulate data and execute programs. Many of these tricks are not obvious, and it takes time to build up expertise in shell scripting. This is markedly different to general-purpose languages like Python which favour fewer, more explicit syntax features, prioritising readability over brevity.

We encourage you to try your hand at writing shell scripts routinely - little snippets to help automate common tasks to find in your everyday workflow. As you get more familiar with the tools you'll write increasingly sophisticated scripts, helping improve both your productivity and the reliability of your workflows.

We'll wrap up with a short example, bringing together a few of the concepts discussed in this section. The `bash` script below adds a license header to the files passed to it.


```

#!/usr/bin/env bash
# Adds a license header to selected files
# Usage: ./test_script -l <license_file> [files..]

# Exit immediately on error
set -e

# Parse options with getopt
while getopt "l:" option
do
    case "${option}" in
        l) license_file=${OPTARG};;
    esac
done

# Check license_file is a real file
[[ -f $license_file ]] || ( echo "No license file"; exit 1 )

# Add the license to each one and save it to a new file.
# By default, 'for' iterates over "$@"
for file
do
    if [[ -f $file && $file != $license_file ]]; then
        echo "Creating a licensed copy of $file"
        cat $license_file $file > "$file.license"
    fi
done

```

The script doesn't do any file path manipulation itself, instead relying on the calling shell to do path expansion as necessary. Some example usage patterns are:

```

# License all Python files in the current directories.
$ ./license_script -l ./license_file *.py

# License two specific files
$ ./license_script -l ./license_file main.cpp README.rst

# Recursively license all Python files
$ ./license_script -l ./license_file **/*.py

```

Chapter 3

Virtual Machines

What is a Virtual Machine?

A **virtual machine**, often abbreviated as VM, is an emulation of a computer system. It allows you to use your real, physical machine to simulate a different computer. We often refer to a VM as a **guest machine**, while the computer simulating the VM is called the **host**.

A sufficiently powerful host machine can run multiple VMs at the same time. Each VM can have different specs and run a different operating system; the only limitation is the physical hardware available to the host. Generally, programs running on a guest machine, including the operating system, are not aware they are running on a virtual machine. The process of creating and running the underlying virtual machines is a form of **virtualisation** and is carried out by a **hypervisor**.

Hypervisors

A hypervisor is a bit of software, firmware or hardware that allows a host machine to create and run virtual machines. They were originally developed in the 1960s as **bare metal hypervisors**, running directly on the hardware of mainframe computers. These are **Type 1** hypervisors, also called **native hypervisors**.

Type 2 hypervisors or **hosted hypervisors** are software based and run on top of the host machine's operating systems. Both types of hypervisor are in use today and the distinction between the two is not always clear.

Why we use virtual machines

Virtual machines have several advantages over a traditional, physical machine. Their 'hardware' can be upgraded easily without physically replacing any components: memory can be added or removed; extra cores added to processors; virtual hard disks can be resized programmatically - much easier than having to copy data to a new hard disk.

They are also easier to duplicate: the entire state of a virtual machine can be represented as a single file called an image. Images can be used to create new instances of virtual machines, all initially identical to each other. Images can also be used as snapshots; making it easy to back-up and restore a VM.

There are some drawbacks to running virtual machines. There are the overheads of virtualisation so they will always be a little less efficient than the equivalent physical machine. If you have very strict power consumption or performance targets then dedicated hardware might be best. However, for most use-cases the flexibility of virtual machines outweighs the downsides;

especially, as we will cover later in the course, in the cloud.

Types of virtual machine

There are two broad types of virtual machine with slightly different purposes.

System Virtual Machines

Also known as "Full Virtualisation VMs", these provide a substitute for an entire physical machine. They include all the virtualised hardware required to execute an entire guest OS in isolation.

This allows a single host machine to run multiple guest machines, each logically isolated from the others despite running on the same underlying hardware. The management of the virtual resources is performed by a hypervisor. Depending on the nature of the host machine a hypervisor might be software, firmware or even hardware based.

There is no requirement for a host machine to be a physical machine. It is possible to nest virtual machines inside other virtual machines. We will see practical uses for this later in the course. (There must, of course, be at least one physical machine for the virtual machines to run on.)

Process virtual machines

These virtual machines are focused on a specific task: providing a consistent environment for executing a program. This improves portability by allowing the same program to be run on a variety of different host machines without recompilation.

Well known examples of process virtual machines include the Java Virtual Machine (JVM) and .NET's Common Language Runtime (CLR). Programs written in Java compile to a special intermediate language. The JVM then takes responsibility for converting the program to machine code that will run on your host machine. This allows any program written in any JVM language (e.g. Java, Kotlin, Scala, etc) to run on any host machine - assuming there is a JVM targeting it. The CLR provides a similar function for Common Language Infrastructure languages (e.g. C#, F#, VB.NET, etc).

While both the JVM and CLR are hugely influential, for the rest of this module we will be talking about System Virtual Machines.

Use Cases for VMs

Use in local dev projects

Virtual machines have a variety of uses in local dev projects. A team all using the VM image can be sure they are running their code in the same, consistent environment; despite

all using separate development machines. This also allows you to test your code on a 'production-like' machine, even if your own machine differs significantly in power / OS etc.

Use in data centers / in the cloud

Virtual machines are also pretty much essential to cloud computing. Physical servers in data centres can be split into multiple virtual machines of different sizes, each isolated from other machines running on the same host. VMs belonging to different clients can run on the same physical host without being aware of each other.

We will discuss **cloud computing** in more detail later in the course.

Using a local VM

In this section we will run through some of the steps required to set-up a local VM. While you can follow along on your own machine there is no need, you will rarely (if ever) actually need to set up VMs like this.

Getting a Hypervisor

The first step is to get a hypervisor to handle the virtualisation. One well known tool is [Oracle VirtualBox](https://www.virtualbox.org/) (<https://www.virtualbox.org/>), an open source, cross-

platform Type 2 hypervisor. Microsoft Windows users can use [Hyper-V](https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/about/) (<https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/about/>), a Type 1 hypervisor. These usually require administrator permissions to run.

Creating a new Virtual Machine

Once you have a hypervisor installed creating a virtual machine is a fairly easy step. Your hypervisor will almost certainly include a wizard of some sort to guide you through the process.

This will typically involve choosing a name for your VM; assigning it some memory; creating a virtual hard disk for it and granting it access to some of the host machine's resources like optical drives and network cards. Some hypervisors (like Hyper-V) will also let you install an operating system automatically; others (like VirtualBox) only provide a barebones VM - you will have to download and install the OS yourself.

Once you've finished you'll be able to start running your VM. There are usually at least two ways of doing this: 'Normal' and **headless**. Running in 'Normal' mode causes the hypervisor to open a UI window so you can interact directly with the VM. This is particularly useful when running a VM with a graphical interface like Windows.

The 'Headless' mode starts the VM running in the background without opening a window for the UI. This saves the overhead of UI while still letting you run the VM. This is particularly useful for a webserver where its main purpose is to serve requests over HTTPS. In fact, server Ubuntu editions do not have a GUI at all. If you run an Ubuntu Server VM in 'Normal' mode it simply opens a terminal, command line like interface.

This raises the question, how do we securely interact with a headless computer? Linux based servers solve this problem by implementing a protocol called **Secure Shell**, almost exclusively referred to **SSH**.

SSH

So far we've been using our terminal to access a shell that communicates with a kernel on our local machine. SSH allows us to connect our terminal to a shell running on a different machine. We can use this to control our local, headless virtual machine or, indeed, *any* machine that we can open a connection to.

You can initiate a connection like so:

```
ssh user@hostname
```

Where `hostname` is the hostname (or IP address) of the remote machine you want to connect to and `user` is the username of the user you want to connect as. The user must already exist on the remote machine and you will then be asked for the user's password.

After successful authentication you will have started a new session on the remote machine. Your terminal will change to reflect this.

SSH Encryption

To ensure its connection is secure, SSH makes use of **public-key cryptography**, also known as **asymmetric cryptography**. A full description of this system is beyond the scope of this course but we will summarise briefly.

To use this system a pair of **cryptographic keys** is generated. These keys are large numbers with two important properties: 1) anything encrypted with one key can only be decrypted with the second key; and 2) if you have one key you cannot work out what the second key is. By distributing one key as a **public key** we allow anyone to encrypt a message that can only be decrypted using our second, **private key**.

When connecting to a SSH server using password authentication the SSH client uses the server's public key to encrypt the password, to allow a secure connection. Your client stores the server's public key in `~/.ssh/known_hosts` to help secure future connections to the same server.

Public-private keys can also be used to replace password based authentication entirely. This can be particularly useful when running scripts against remote servers. By placing a public key in the `~/.ssh/authorized_keys` file you can then connect to that server, as that user using only your *private key* for authentication.

Generating a new public-private key pair is very easy using the `ssh-keygen` tool. However, you should always be *extremely* protective of your *private key*.

You can now interact with the remote machine in the same way you interacted with your local machine's shell earlier in this module.

While the core function of SSH is to allow a secure connection between two machines, it is a very powerful and versatile tool. Some common use cases you should be aware of include securely copying files (e.g. using the `scp` command) and syncing files and directories (using `rsync`).

Important: even though SSH does require authentication it is ***not*** a good idea to allow your web server to accept SSH connections from anywhere in the world. As a security measure, if your machine is directly exposed to the internet it should *not* accept SSH requests.

The Ubiquity of Virtualisation

Virtualisation is a cornerstone of modern devops practices and tooling, and the hypervisors we've described above provide tools to manually create virtual machines. It is, however, unlikely that you will ever have to create a single VM like this. As we will see throughout the remainder of this course: a variety of tools, techniques and concepts have developed to make managing modern infrastructure significantly easier by harnessing the flexibility enabled by VMs and virtualisation.

Chapter 4

Automating VM Management

In the last section we introduced the concept of virtualisation and virtual machines. We also said that generally it was unusual to manually set up a single VM.

In this section we will look at some techniques to automate the setting up and configuration of virtual machines.

Shell Setup Scripts

One simple technique is to create a script to aid with the setup. This will often allow you to simplify the setup of your application down to only a small number (ideally just one 1) scripts that need to be run. You've already cover the basics of scripting in this module so you can imagine how a script to set up a machine to run a python app might look:

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

# Install Python
sudo apt install python3.8

# Make the directory for our app
sudo mkdir -p /opt/my-app
cd /opt/my-app

# Get the latest version of our application
git clone git@github.com:MyUserName/my-new-repository.git .
git checkout latest

# Set environment variables
export DATABASE_CONNECTION_STRING=<secret>
export API_URL=my-app-api.example.com

# Run our application
python3 start.py
```

This is obviously a simplified, but *plausible*, set-up script for an application. We specify the commands that must be run in order to create the correct environment. While this is quicker and more reproducible than doing this manually it still has some limitations:

- We still have to run each step everytime we set-up an environment
 - This could be very slow if there are lots of dependencies
 - Setting up lots of servers could take even longer
- The environment variables are hardcoded and secrets are visible
 - Do we need a different script for DEV, TEST and PRODUCTION environments?
- Perhaps, most importantly, we still need to create the VM manually to start with.

Let's look at some alternatives.

Images

Another simple technique that might help us is using VM images. As we previously touched on, a VM image is a 'snapshot' of the current state of the virtual machine. It includes everything about the setup of the machine at that point in time. By using a copy of a VM image it is easy for another person to create their own identical copy of your virtual machine. This means we can create a VM that is setup exactly as we want it, take a 'snapshot' and then spin up as many identical copies as we want from that image.

While this does make creating new VMs easier and quicker there are still some downsides.

The VM still needs to be created and updated manually (a script doesn't help us much here). This means we may have multiple versions of the image to manage. Images are binary files so it is difficult to source control them effectively. It's essentially impossible to tell the difference between two versions without booting up the VM and manually comparing the contents of each machine. This means you have to curate a separate set of metadata about each image and hope that it stays up-to-date.

Configuration Management Tools

In some situations we will be able to manage with the tools above. The drawbacks will be acceptable. However, it is very likely we will need a more sophisticated solution. Fortunately several people have encountered problems like this before and come up with helpful solutions: **configuration management tools**.

There are lots of different tools available, all with their own quirks and idiosyncracies. They do, however, all follow the same basic idea: we want to write the configuration for our servers once (in a readable text format)

and then let the configuration management tool apply this configuration to our servers automatically.

Storing all configuration as text files means it is easy to see what it contains: it's not as opaque as a binary image and can be documented in-line. It's also easy to source-control, providing us with all of the benefits of versioning. It can be easily stored alongside our existing code.

Configuration management tools can also make it much easier to configure multiple servers. As the tool is responsible for provisioning the virtual machines it can often be as simple as changing a number in a configuration file. The server creation process itself is automatic, accurate and repeatable.

Aside: Imperative vs Declarative

These tools can use two different approaches to defining configuration: **imperative** and **declarative**. The difference between these two approaches is quite important in understanding how the tools will work.

When using an imperative approach we have to define *how* to configure our servers. This is the approach we used in the scripts example above: first install Python3, then make a directory, then clone the source code. Imperative approaches give us a lot of

control, but also require us to be explicit about every step of the process.

The alternative declarative approach is to define *what* the final system is like and leave the *how* up to the tool. We would *declare* that we want an Ubuntu server with Python3 installed; the tool would then work out how to achieve that end state.

Generally we prefer a declarative approach to configuration management. It lets us focus on the outcome rather than the process of achieving it.

We will now look at examples of some configuration management tools.

Ansible

Ansible describes itself as an "IT automation system". We are going to focus on its configuration management functionality but it is able to do a lot more including provisioning cloud infrastructure, deploying applications and [more](https://docs.ansible.com/ansible/latest/index.html) (<https://docs.ansible.com/ansible/latest/index.html>).

When using Ansible you create files called **playbooks** to describe the infrastructure of your application. Ansible's automation engine then uses these files to configure your system. There is also an enterprise framework for controlling and managing automation via a UI and REST API.

Let us have a closer look at some of the terminology.

Inventories list the hosts that are to be controlled by Ansible, organising them into logical groups (e.g. web servers, database servers, network devices, etc.). Inventory lists can be static, or generated dynamically by running a script or calling an API. Here's an example of a static list of hosts:

```
[web]
web-1.example.com
web-2.example.com

[db]
db-a.example.com
db-b.example.com
```

Playbooks are plain-text YAML files that describe the desired state of a machine. They can be used to build entire application environments.

Playbooks describe a structured hierarchy:

- Playbooks contain plays
- Plays contain tasks
- Tasks call modules and trigger handlers

Tasks are executed in order, one at a time, against all machines specified by the play, before moving on to the next task. Tasks can also trigger handlers at the end of plays. Plays, like tasks, run in the order specified in the playbook, from top to bottom.

Variables allow you to alter how playbooks run and the configuration state they describe. Variables can be defined in a number of ways, including within playbooks themselves, in separate files or inventories, or via command line options. Variable values can also be "discovered" by having Ansible query properties about the machine(s) it is connecting to (these variables are called *facts*).

Ansible's ability to do things and interact with various services is determined by *modules*; many ship as standard and others can be installed (or custom ones written) as needed. Modules are small programs that control the

things you're automating, including system resources, installing packages, files, or other infrastructure (e.g. cloud services).

Roles are a special kind of playbook that are fully self-contained with tasks, variables, configuration templates and other supporting files.

Finally, *Ansible Tower* is an enterprise platform product offered by Red Hat that provides a web UI and REST API to allow more advanced management of Ansible jobs (i.e., playbook executions and other orchestration activities). It supports things like multi-playbook workflows with conditional execution, scheduled jobs and activity audits ("who ran what job when").

Ansible system model

At a high level, the Ansible system architecture is as follows:

- Your infrastructure is made up of nodes -- typically the servers that run the various parts of an application.
- The control node is responsible for orchestrating changes to the infrastructure, by running Ansible playbooks and other commands.
- Managed nodes are the machines that form the application infrastructure itself and receive instructions from the control node.

The controller communicates with managed nodes over SSH, directly running commands on those machines. This is in contrast to other automation platforms, which require a dedicated agent process to be installed and run on each server that needs to be controlled. By leveraging commonly-used tools, including SSH and Python, this minimises the additional libraries and software that need to be installed and maintained on each node.

Ansible applies changes to servers by connecting to your nodes and pushing out modules to them. Module programs are written in Python and designed to be resource models of the desired state of the system. Ansible then executes these modules (over SSH by default), and removes them when finished. When managing network devices (firewalls, switches, etc.), Ansible modules behave differently: because the majority of network devices cannot run Python, network modules are executed on the Ansible control node instead, where `ansible` or `ansible-playbook` runs.

Example: Ansible Playbook

Here's an example of an Ansible playbook to set up a basic webserver. It has one play, which consists of three three tasks and a handler:

```
---
- name: install and start apache
  hosts: web
  remote_user: root
  vars:
    http_port: 80
    max_clients: 200

  tasks:
    - name: install httpd
      yum:
        name: httpd
        state: latest
    - name: write apache config file
      template:
        src: /srv/httpd.j2
        dest: /etc/httpd.conf
      notify:
        - restart apache
    - name: start httpd
      service:
        name: httpd
        state: started

  handlers:
    - name: restart apache
      service:
        name: httpd
        state: restarted
```

Note that the three dashes (---) at the top of the file indicate that this is a YAML file.

The main aspects of the install and start apache play within this playbook are:

1. The `hosts` and `remote_user` keywords define which group of hosts Ansible should apply this play to, and which username to use when connecting to them.
2. Two variables are defined: `http_port` and `max_clients`
3. Each task has a name (e.g. `install httpd`) and a module (e.g. `yum`, `template`, `service`)
4. The module declared for each task refers to a specific action (or set of actions) that Ansible knows how to perform, and the relevant parameters are specified
 - For example, the package name to be installed by `yum` (`name: httpd`) and the version to use (`state: latest`).
5. A handler is defined to restart the Apache service (`httpd`), which is called if the `write apache config file` task changes the existing config file on the machine.

Chef

Chef is written in Ruby and Erlang. It uses a domain-specific language written in Ruby to define system configuration "recipes". Chef can be installed on all major operating systems (a key advantage over Ansible, which does not support running its main control node software on Windows).

Chef is also able to integrate with popular cloud-based platforms (Amazon Web Services, Google Cloud Platform, Microsoft Azure, etc.) to automatically provision and configure new machines.

Recipes describe a series of resources and their desired state: packages that should be installed, services that should be running, files that should be written, etc. Chef makes sure each resource is properly configured and corrects any resources that are not in the desired state. A resource corresponds to some piece of infrastructure, such as a file, a template, or a package. Chef Infra provides many commonly-used resources out of the box, but you can also use resources from community cookbooks, or write your own resources specific to your infrastructure.

A Chef *recipe* is a file that groups related resources, such as everything needed to configure a web server, database server, or a load balancer. A cookbook provides structure to your recipes and helps to

organise project files for your infrastructure as a whole.

Chef platform model

The Chef Infra platform is made up of the following components:

- *Chef Workstation* is a set of tools to write and test recipes and cookbooks.
- *Chef Infra Server* acts as a hub for configuration data, storing cookbooks, the policies that are applied to machines in your infrastructure, and metadata that describes each machine.
- Client nodes are the machines that are managed by Chef and run the Chef Infra Client (the agent software).

Periodically, the Chef Infra Client software running on each client node contacts the Chef Infra Server to retrieve the latest cookbooks. If (and only if) the current state of the node doesn't match what the cookbook says it should be, Chef Infra Client executes the cookbook instructions. This iterative process ensures that the IT infrastructure as a whole converges to the state described by the cookbook(s).

Chef DSL

Chef uses its own Domain Specific Language (DSL). This has some benefits and drawbacks:

Benefits

- You can compile it.
 - This allows you to check before deployment that the config is valid.
 - If it isn't, you catch this early and can be provided clear error messages.
- You *may* even be able to add some IDE support.
- With care, you can make Chef configuration files extremely readable.

Drawbacks

- Users have to learn a dedicated language.
- Syntax highlighting doesn't necessarily come for free.

Additional Material

Validating Shell Scripts

[ShellCheck](https://www.shellcheck.net/) (<https://www.shellcheck.net/>) is a popular static analysis tool that helps develop and validate shell scripts. It's available as a web tool, CLI utility and in many IDEs. It's even available as a [VSCode extension](https://github.com/timonwong/vscode-shellcheck) (<https://github.com/timonwong/vscode-shellcheck>).

PowerShell Aliasing

PowerShell is an extremely verbose scripting language, particularly when compared to its counterparts in the Unix ecosystem. This is a deliberate design choice - PowerShell scripts are meant to be readable and maintainable! To also support a concise scripting experience, PowerShell has a standard system for aliasing cmdlets and cmdlet parameters.

PowerShell defines a wide range of aliases. For example `r` is an alias for the `Invoke-History` cmdlet. You can see all defined aliases by running the `Get-Alias` cmdlet. PowerShell uses aliases to mimic common built-in Unix shell commands (for example `echo`, `cd` and `pwd`).

You can define your own aliases, or modify existing aliases, using the `New-Alias` and `Set-Alias` cmdlets, respectively. These changes aren't persistent by default - the aliases are scoped to your active shell. You can write alias definitions into your PowerShell profile file to automatically re-set them in future.

Cmdlet parameters can also have aliases. You can find any available aliases using the `Get-Command` cmdlet.

```
# Find all parameter details of the cmdlet
$ (Get-Command <cmdlet_name>).Parameters.Values

# An example, filtering the output
$ (Get-Command Get-ChildItem).Parameters.Values | Select Name, Aliases
```

Name	Aliases
----	-----
Path	{}
LiteralPath	{PSPath, LP}
Filter	{}
Include	{}
Exclude	{}
Recurse	{s}
Depth	{}
Force	{}
Name	{}
Verbose	{vb}
Debug	{db}
ErrorAction	{ea}
WarningAction	{wa}
InformationAction	{infa}
ErrorVariable	{ev}
WarningVariable	{wv}
InformationVariable	{iv}
OutVariable	{ov}
OutBuffer	{ob}
PipelineVariable	{pv}
Attributes	{}
FollowSymlink	{}
Directory	{ad, d}
File	{af}
Hidden	{ah, h}
ReadOnly	{ar}
System	{as}

As well as using pre-defined parameter aliases, you can replace any powershell parameter name with a truncated version, provided the shortened name remains unambiguous. Using a combination of cmdlet aliases, parameter aliases and providing truncated alias names, you can make PowerShell commands much more terse. This has its advantages and disadvantages. Remember, the full forms are generally more readable and maintainable.

Package Managers

This module has focused on built-in shell tools and system-provided programs. However, there's much more you can do by installing and using third-party tools. There are a variety of CLI package-management tools that make it easy to install, uninstall and update software from external code repositories.

Windows

PowerShell includes a [package management framework](https://docs.microsoft.com/en-us/powershell/module/package-management/) (<https://docs.microsoft.com/en-us/powershell/module/package-management/>) that package providers can register with. Out of the box, Powershell comes with NuGet and PowerShellGet, providing access to the [PowerShell Gallery](https://www.powershellgallery.com/) (<https://www.powershellgallery.com/>) and [NuGet repository](https://www.nuget.org/) (<https://www.nuget.org/>). You can install third-party package providers to supplement this, with [chocolatey](https://chocolatey.org/) (<https://chocolatey.org/>) being a popular option.

Mac

Apple do not bundle a package manager with Mac OS. There are a range of third-party options, with [homebrew](https://brew.sh/) (<https://brew.sh/>) being by far the most popular.

Linux

Most Linux distributions include a CLI package manager. You'll rarely have reason to look at third-party options. In the Debian family of Linux distributions (including Ubuntu) you'll use `dpkg` for package management, although users will typically interact via a higher-level wrapper (e.g. `apt`).

Other Linux and Unix varieties have their own package management solutions. We won't enumerate them here, as they are almost certainly already known to users of those systems.

Vagrant

We've talked about various server configuration tools (Ansible, Chef) that can help configure machines, both physical and virtual, in a production environment. But what about local development? A good development workflow needs a convenient way for developers to run and test their code in a production-like manner.

[Vagrant](https://www.vagrantup.com/) (<https://www.vagrantup.com/>) is a configuration management tool for local VMs, built on top of VirtualBox. It's a popular option for simulating a production environment locally, all the while automating many of the steps required if using VirtualBox alone.

In practice vagrant can take quite some effort to set up, but is often worth it. Once configured a simple `vagrant up` command will set up everything you need to run your code locally.

Windows Subsystem for Linux

It's common to need access to a Linux environment when developing on Windows. You may, for example, need to run certain compilers that don't exist on the Windows platform, or generally access the UNIX ecosystem of command line tools. There are a few long-standing tools for this, providing Linux emulation on Windows (e.g. `cygwin`, `git bash`).

Microsoft have recently developed the Windows Subsystem for Linux [Windows Subsystem for Linux \(WSL\)](https://docs.microsoft.com/en-us/windows/wsl/wsl2-about) (<https://docs.microsoft.com/en-us/windows/wsl/wsl2-about>), providing first-party support for running Linux within Windows. WSL version 2 was released as part of Windows 10 version 2004 (June 2020), and includes a Linux kernel running on a highly optimised VM. The VM is extremely efficient and fully managed by Windows.

WSL allows you to download and install multiple Linux distributions, running with a near-native experience. WSL and Windows

share a common file system, making it easy to switch between the two (although take care with line endings with some older Windows software!).

Older versions of WSL (known as WSL1) are based on an entirely different architecture, and have more significant drawbacks. WSL1 has more limited file system integration, and suffers on a performance front relative to WSL2. Due to the use of limited kernel emulation rather than full virtualisation, WSL1 was not able to support container software (e.g. Docker) - this is one of the major issues addressed by WSL2.

We recommend WSL2 as a standard tool in a Windows-based development workflow. It gives users the flexibility to work with tools from either OS as required. It renders older emulation tools (e.g. cygwin and git bash) largely redundant.

Secret Management

One challenging aspect of DevOps is deciding how to handle application secrets (e.g. passwords, database connection strings, private keys, etc) . There are many considerations and potential issues. For example, it's obviously bad practice to store your secrets in plaintext in source control. Anyone with access to the repository can then read them! But if not source controlled,

how should they be handled? There are some general approaches:

- **Encrypt secrets and store in source control.** This is a common approach, and there are tools out there that can help (e.g. [gitcrypt](https://github.com/AGWA/git-crypt) - <https://github.com/AGWA/git-crypt>). It can be tricky to set up, however, and the history can be invalidated by other practices such as key rotation.
- **Store secrets in configuration files.** For your exercise, you've likely stored secrets in a local configuration file. It's not source-controlled, but it's there when the application needs it. You can do the same on a production server, however you would want to encrypt the file as well.
- **Use an external secrets management service.** Cloud providers offer managed secrets hosting, for example AWS Secrets Manager, GCP Secrets Manager and Azure Key Vault. Such services are generally the preferred option.

Configuration management tools generally have built-in support for specific secret-management workflows. For example, both Ansible and Chef have their own tools to support source-controlled encrypted secrets files. Other tools in this space take different approaches (for example Puppet, which

favours integration with third-party secrets managers).

This is just a brief introduction to some high-level ideas around secrets management. We'll come back to secrets management in later modules when we talk about cloud infrastructure.

Are VMs obsolete?

Later in this course we'll introduce various managed cloud services and container solutions (e.g. Docker) that provide alternative ways to provision production infrastructure. As we'll see, these **immutable infrastructure** tools mean that, in many cases, we don't need to worry about configuring and updating servers at all. These services provide an alternative to virtual machines, and are often easier to setup and maintain.

VMs are not so widely run by end-users as they once were, however they are still extremely common in all but the latest technology stacks. Whether you manage VMs or not, it's important to have a background understanding of the technology, as virtualisation has enabled the evolution of highly scalable, on-demand cloud services modern DevOps workflows rely on.

If you're setting up a new software project from scratch, we'd advise looking into a

container-based infrastructure instead of using virtual machines directly. As we'll see, containers are more lightweight and considerably easier to manage. You can also investigate the offerings of cloud infrastructure providers, who can deploy fully managed virtual machines and container runtimes. Cloud deployments take care of many thorny issues, such as networking, backups and maintenance.