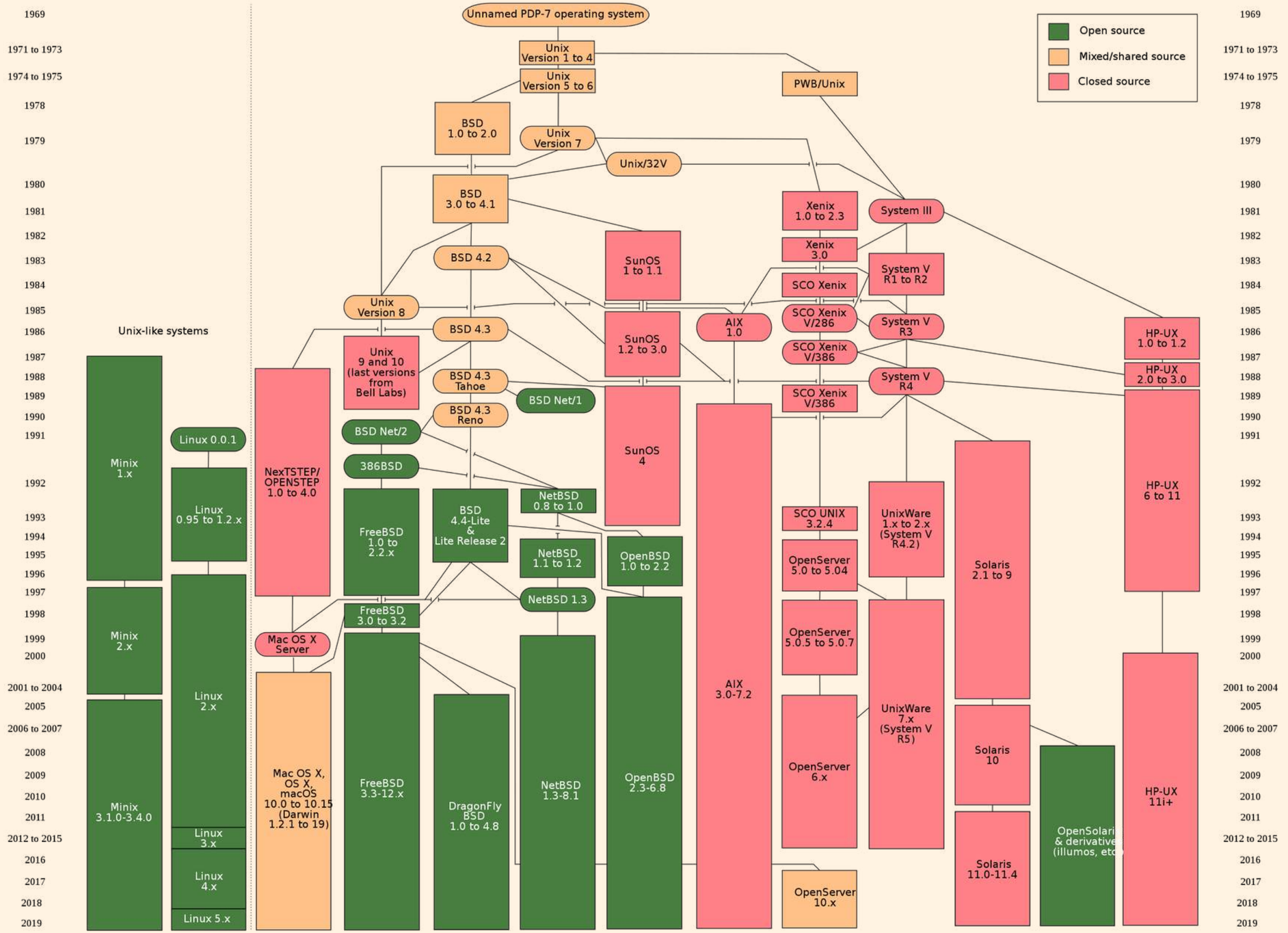
The World Of Operating Systems

Most operating systems can be grouped into two families:

- The Microsoft NT descendants including Windows, Xbox OS, and Windows Phone/Mobile
- Pretty much everything else has lineage going back to Unix, including Mac OS X, Linux, Android, Chrome OS, and even the PS4 OS



Eraserhead1, Infinity0, Sav_vas, CC BY-SA 2.0 via Wikimedia Commons

So What Is Unix?

Unix was an operating system developed at Bell Labs in the mid 1960s. Many of the innovations and design choices the original Unix team have lived on 50+ years later, including the idea of multi-user operating systems and hierarchical file systems.

Unix is the "grandfather" of many modern operating systems that we frequently use today.

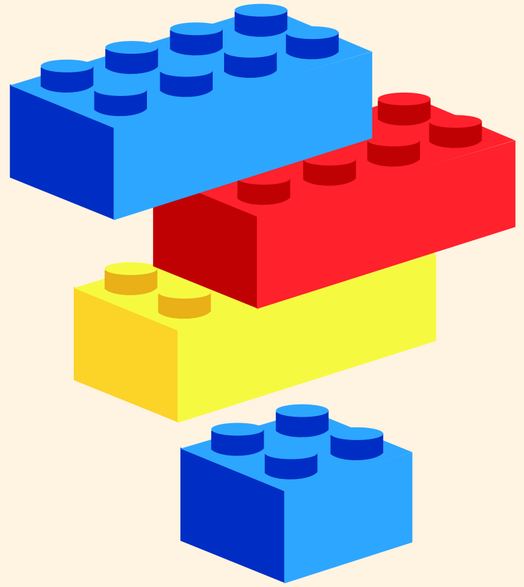
Peter Hamer, CC BY-SA 2.0 via Wikimedia Commons

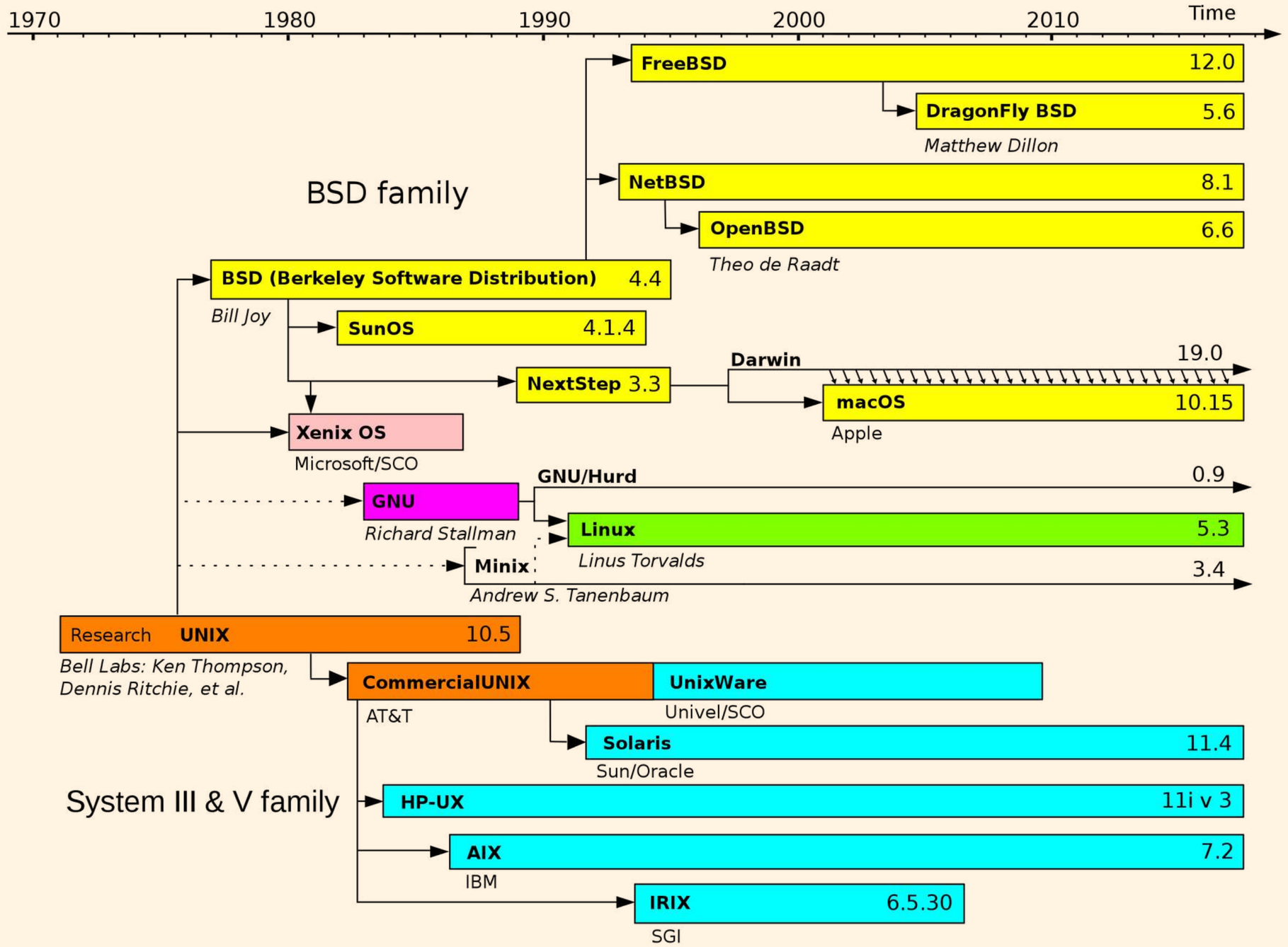
The Unix Philosophy

In the early days of computers, operating systems were tightly tied to specific hardware. Unix decoupled the two and was easily portable to other hardware.

Unix philosophy emphasizes modular software design and the creation of small individual programs that can be combined to perform complex tasks.

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




1970
1980
1990
2000
2010
Time
BSD family
FreeBSD 12.0
DragonFly BSD 5.6
Matthew Dillon
NetBSD 8.1
OpenBSD 6.6
Theo de Raadt
BSD (Berkeley Software Distribution) 4.4
Bill Joy
SunOS 4.1.4
NextStep 3.3
Darwin 19.0
macOS 10.15
Apple
Xenix OS
Microsoft/SCO
GNU/Hurd 0.9
GNU
Richard Stallman
Linux 5.3
Linus Torvalds
Minix 3.4
Andrew S. Tanenbaum
Research UNIX 10.5
Bell Labs: Ken Thompson,
Dennis Ritchie, et al.
CommercialUNIX
UnixWare
Univel/SCO
AT&T
Solaris 11.4
Sun/Oracle
System III & V family
HP-UX 11i v 3
AIX 7.2
IBM
IRIX 6.5.30
SGI

True UNIX

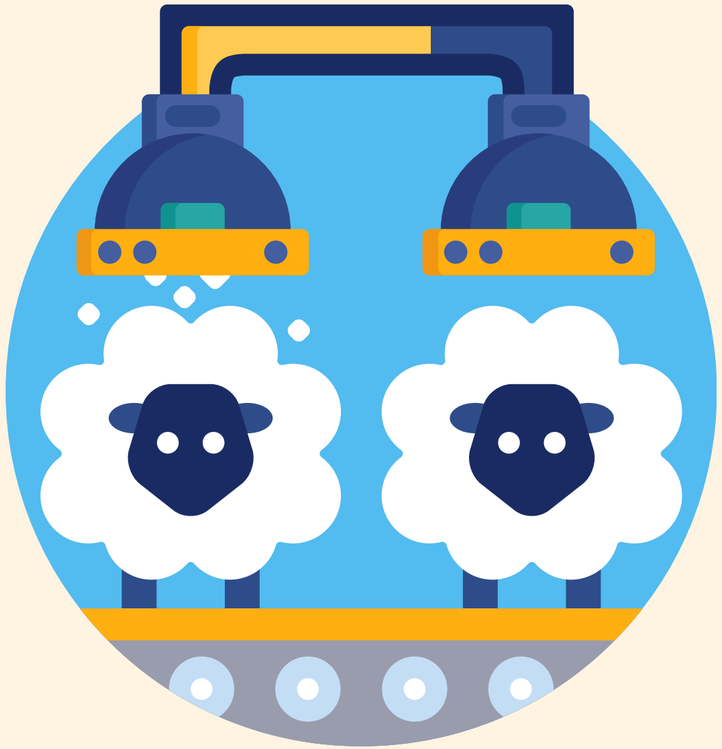
Today the name "UNIX" is a trademark of a global consortium called The Open Group. They maintain a set of standards called the Single UNIX Specification, which describes the core commands, features, interfaces, utilities, and more that define a UNIX operating system.

The Open Group will certify an operating system as fully UNIX compliant if it passes conformance tests. Companies must pay to be tested and must further pay to use the UNIX trademark.

Unix-Like??

Many operating systems are based on the original UNIX operating systems and are compatible with the UNIX standards, but are NOT considered UNIX because they have not been certified by The Open Group. Often this is because of financial considerations or ethical objections.

We call these operating systems Unix-like. They fully or mostly meet the specification but cannot legally use the UNIX name.



Free Software

The Free Software movement came about in the 1980s as a response to the proliferation of proprietary and restricted software. Think of "free speech" rather than "free as in zero price"

The movement's philosophy is that the computers and software should not prevent cooperation between users, and instead should have the goal of liberating everyone in cyberspace.

According to the movement's leader, Richard Stallman, "Users should have the freedom to run, copy, distribute, study, change, and improve the software"

GNU

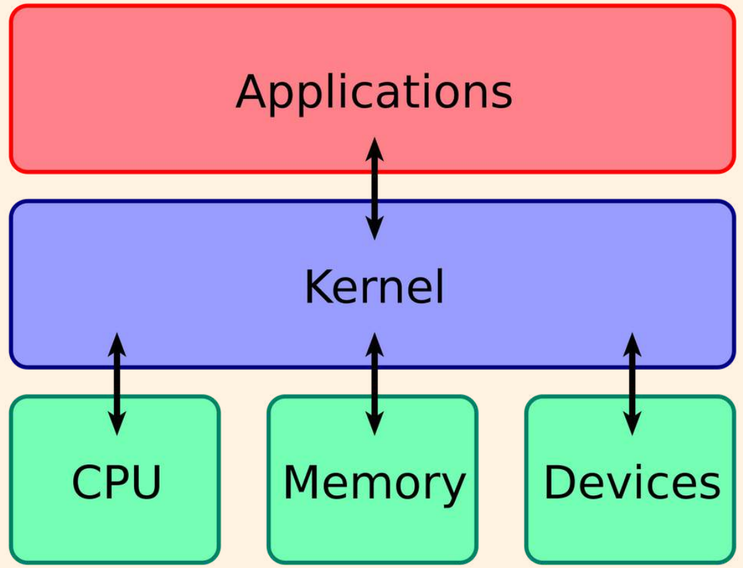
Richard Stallman was a leader in the group of developers who aimed to create Free Software alternatives to Unix.

In 1984 he began work on the GNU Project, with the goal of creating an operating system that included "everything useful that normally comes with a Unix system so that one could get along without any software that is not Free"

The Linux Kernel

Another developer, Linus Torvalds, was working on creating his own kernel known as Linux. The kernel is the part of an OS that facilitates interactions between hardware and software.

At the time, many GNU "pieces" were complete, but it lacked a kernel. Torvalds combined his kernel with the existing GNU components to create a full operating system.

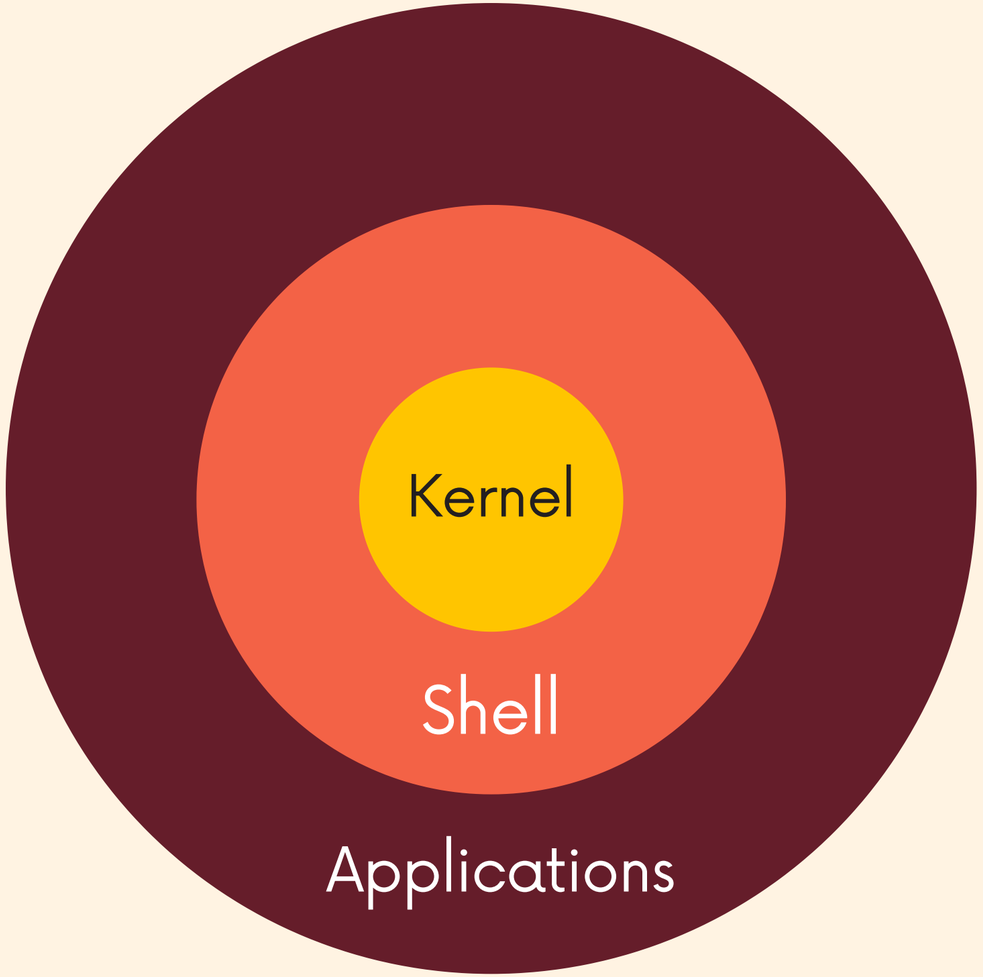


Bobbo, CC BY-SA 3.0, via Wikimedia Commons

Kernel??

A kernel is a computer program that forms the core of an operating system and manages critical tasks like:
- memory management
- task scheduling
- managing hardware

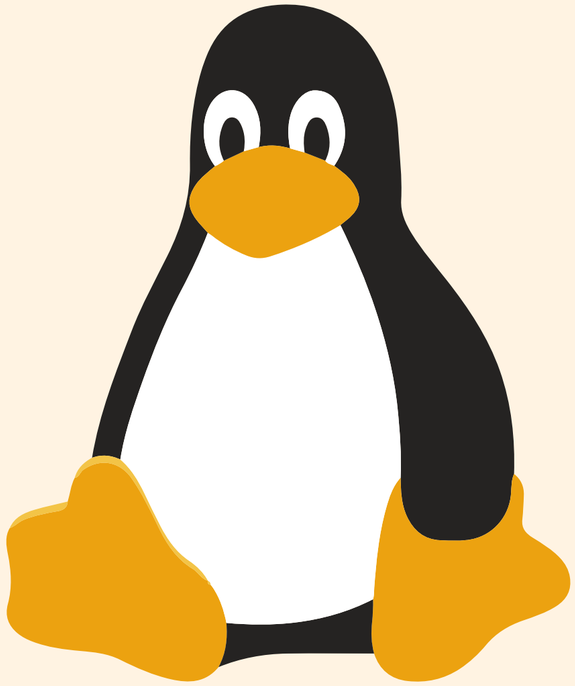
While a kernel is a critical piece, it is NOT the same as an operating system. An engine is the essential "core" of a car, but you can't drive an engine on its own!



Linux

Today, the term "Linux" refers both to the kernel created by Linus Torvalds AND all the software that is part of the Linux ecosystem.

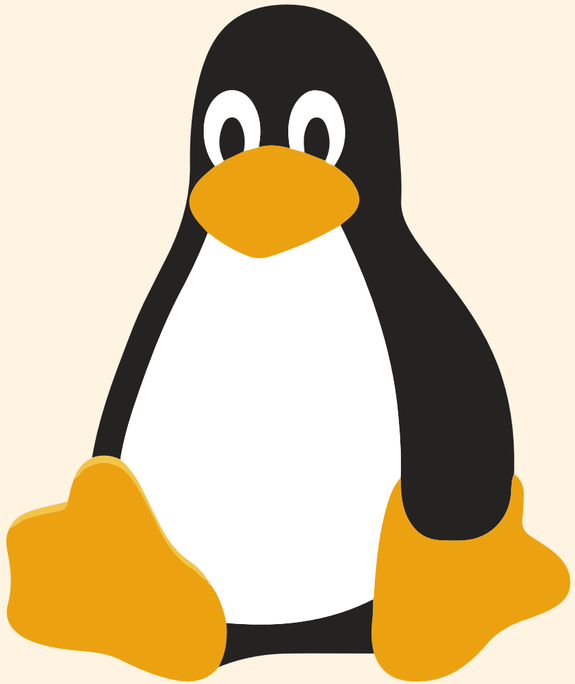
Some users feel strongly that the name GNU/Linux should be used instead, as it properly reflects the GNU Project's contributions.



GNU/Linux?

Some users feel strongly that the name GNU/Linux should be used instead, as it properly reflects the GNU Project's contributions.

- "Calling the whole system "Linux" leads people to think that the system's development was started in 1991 by Linus Torvalds. That is what most users seem to think. The occasional few users that do know about the GNU Project often think we played a secondary role — for example, they say to me, 'Of course I know about GNU — GNU developed some tools that are part of Linux"

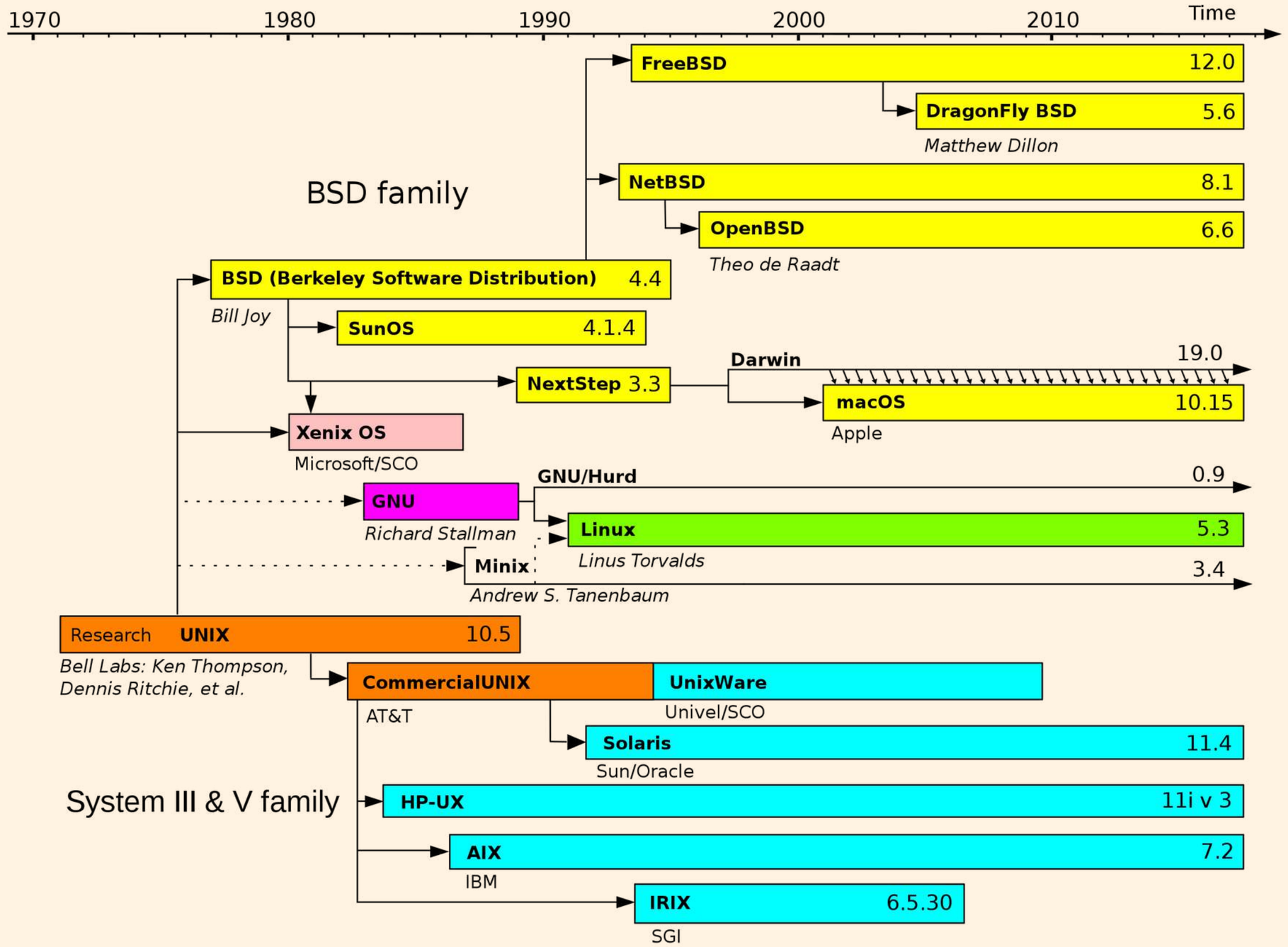


Linux Distributions

The Linux Kernel itself is not a full-blown operating system. When people talk about a Linux-based operating system, they are referring to Linux distributions.

Typically, a Linux distribution bundles together the Linux kernel, GNU tools, documentation, a package manager, a window system, and desktop environment.

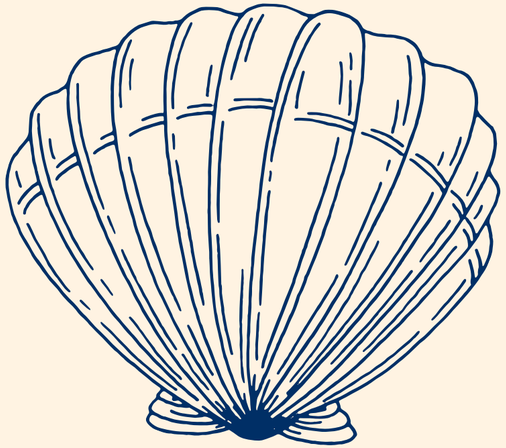
There are nearly 1000 Linux distros available. Some of the more popular ones includes Fedora, Ubunutu, Debian, and Slackware.


1970
1980
1990
2000
2010
Time
BSD family
FreeBSD 12.0
DragonFly BSD 5.6
Matthew Dillon
NetBSD 8.1
OpenBSD 6.6
Theo de Raadt
BSD (Berkeley Software Distribution) 4.4
Bill Joy
SunOS 4.1.4
NextStep 3.3
Darwin 19.0
macOS 10.15
Apple
Xenix OS
Microsoft/SCO
GNU/Hurd 0.9
GNU
Richard Stallman
Linux 5.3
Linus Torvalds
Minix 3.4
Andrew S. Tanenbaum
Research UNIX 10.5
Bell Labs: Ken Thompson,
Dennis Ritchie, et al.
CommercialUNIX
UnixWare
Univel/SCO
AT&T
Solaris 11.4
Sun/Oracle
System III & V family
HP-UX 11i v 3
AIX 7.2
IBM
IRIX 6.5.30
SGI

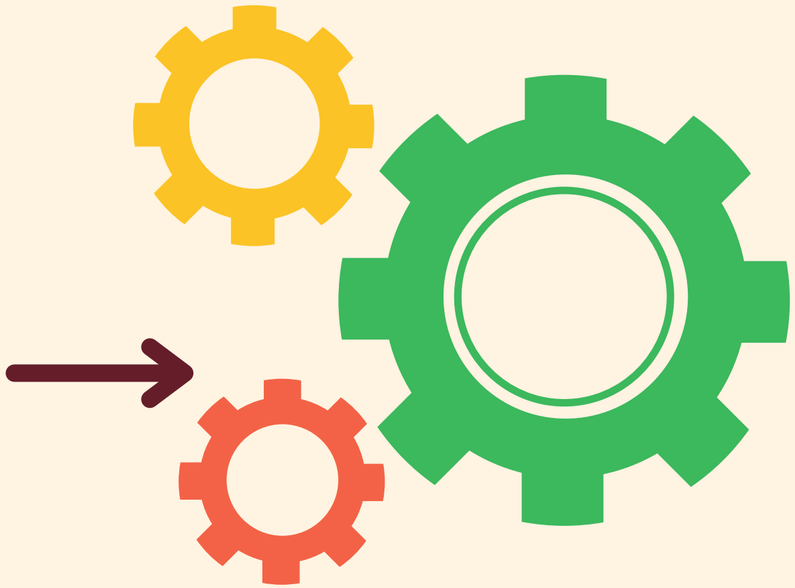
A shell is a computer interface to an operating system.
Shells expose the OS's services to human users or other programs.

The shell takes our commands and gives them to the operating system to perform.

It's named a shell because it is the outer layer around the OS, like the shell around an oyster!

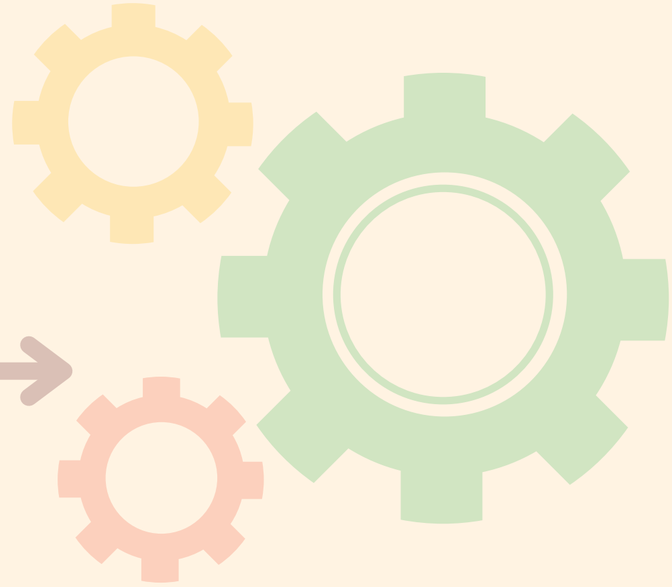


Shell



OS

```
❯touch group-{a..e}.txt
```

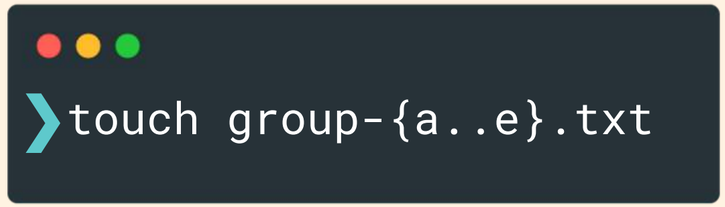


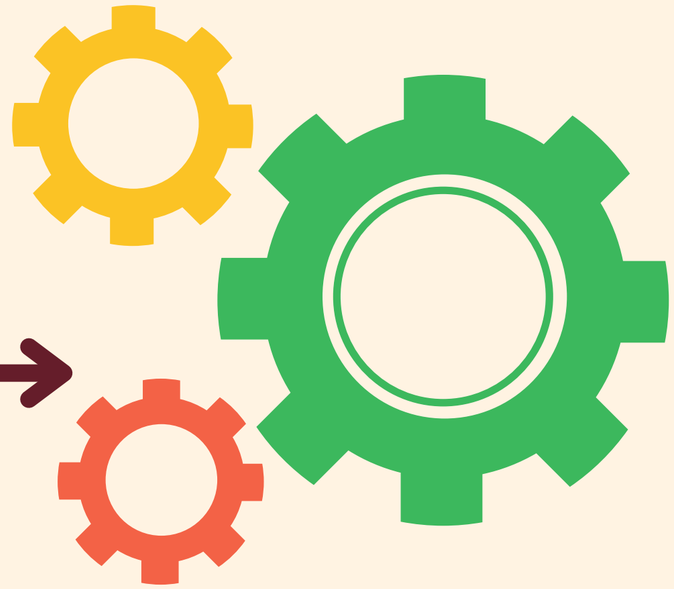
Terminal

Shell

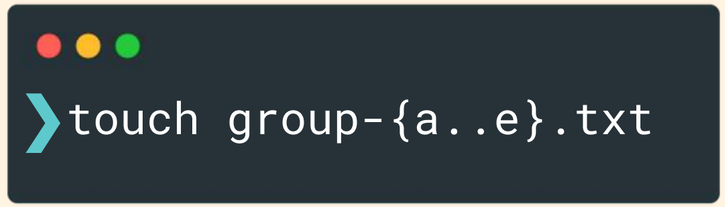
OS

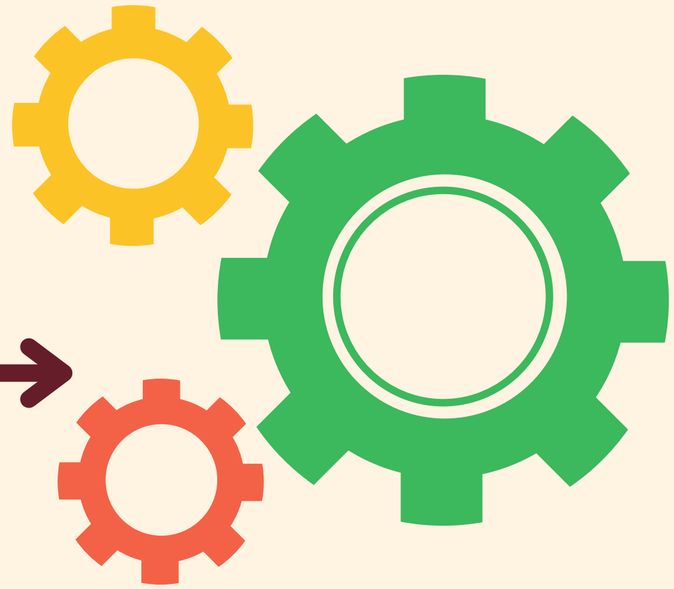
A terminal is a program that runs a shell.
Originally, terminals were physical devices,
but these days we work with software terminals.



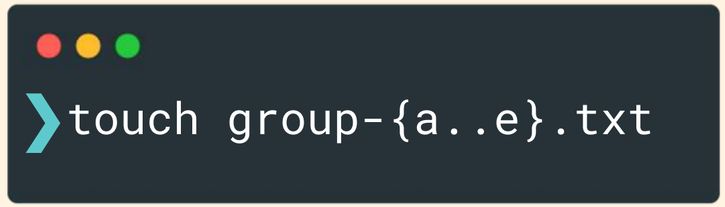


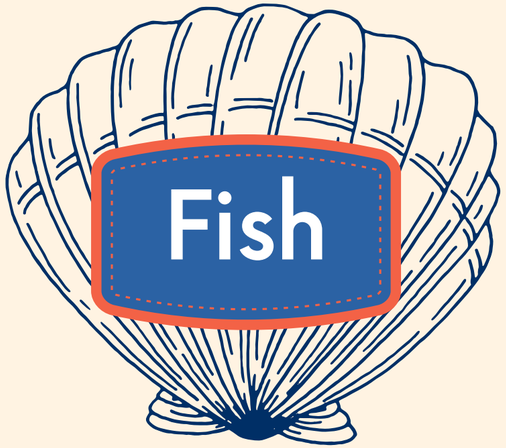
Terminal Shell OS

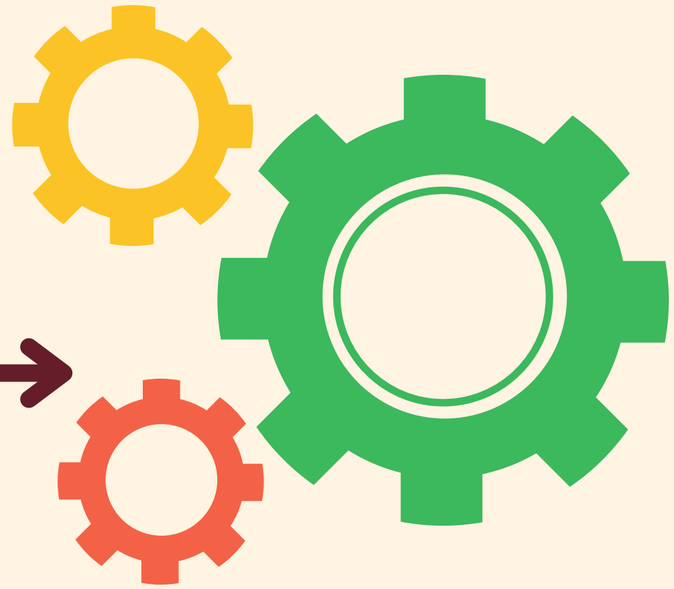




Terminal Shell OS





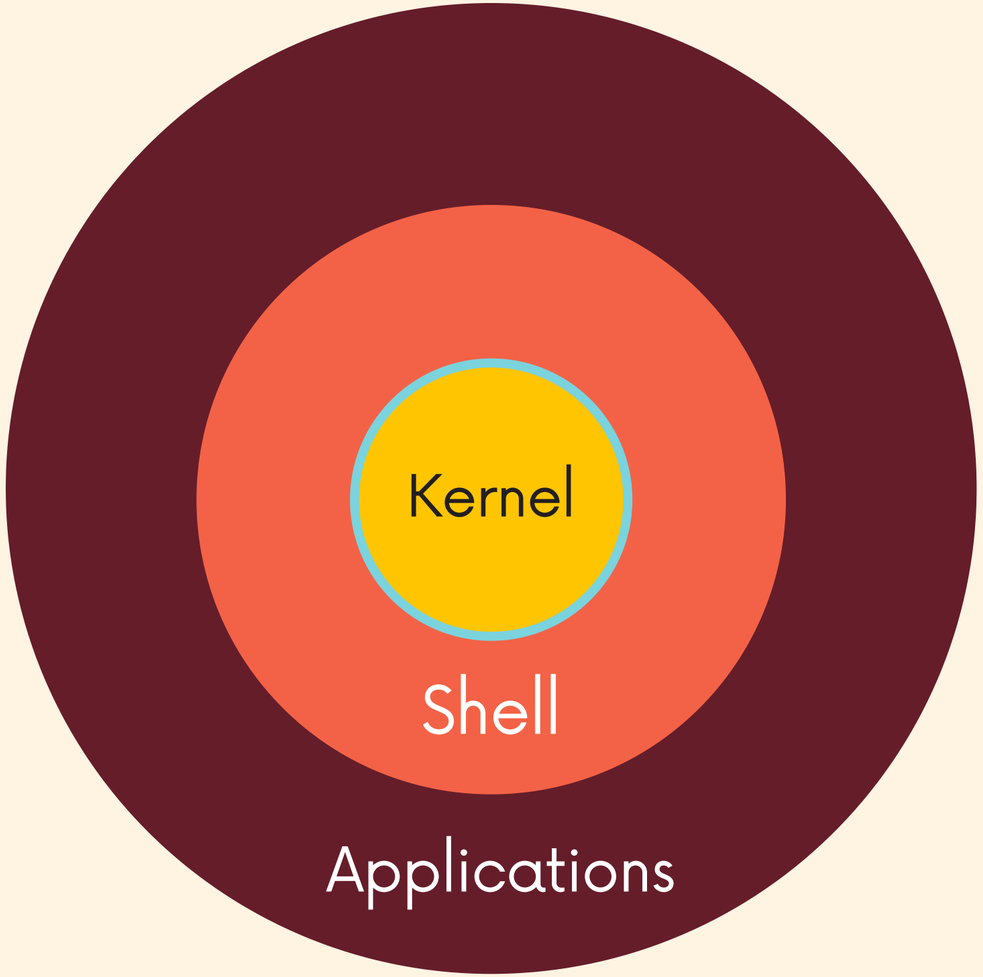


Terminal Shell OS

Kernel??

A kernel is a computer program that forms the core of an operating system and manages critical tasks like:
- memory management
- task scheduling
- managing hardware

While a kernel is a critical piece, it is NOT the same as an operating system. An engine is the essential "core" of a car, but you can't drive an engine on its own!

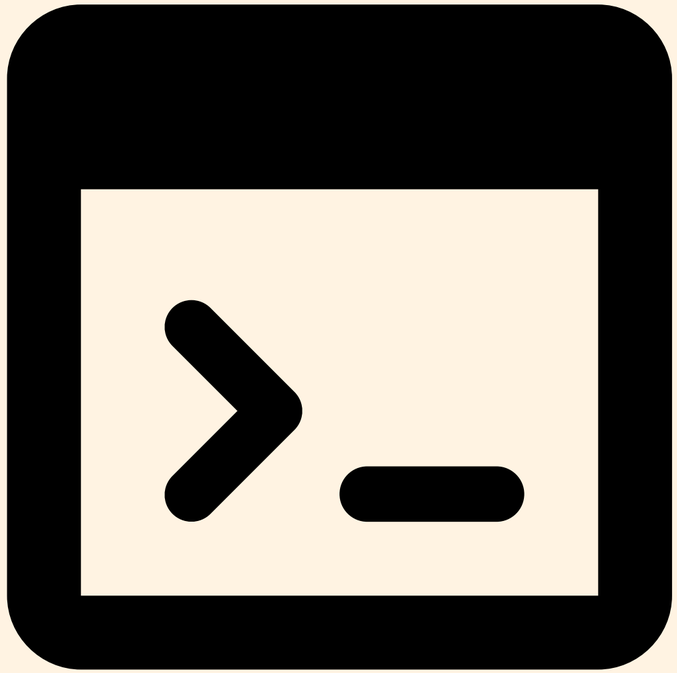


bash

On most Linux-based systems, the default shell program is **Bash.** There are many other shells, but Bash is currently the most popular.

The name "Bash" is an acronym for "Bourne-Again SHell", a punny reference to Stephen Bourne, the creator of Bash's direct ancestor shell, **sh**.

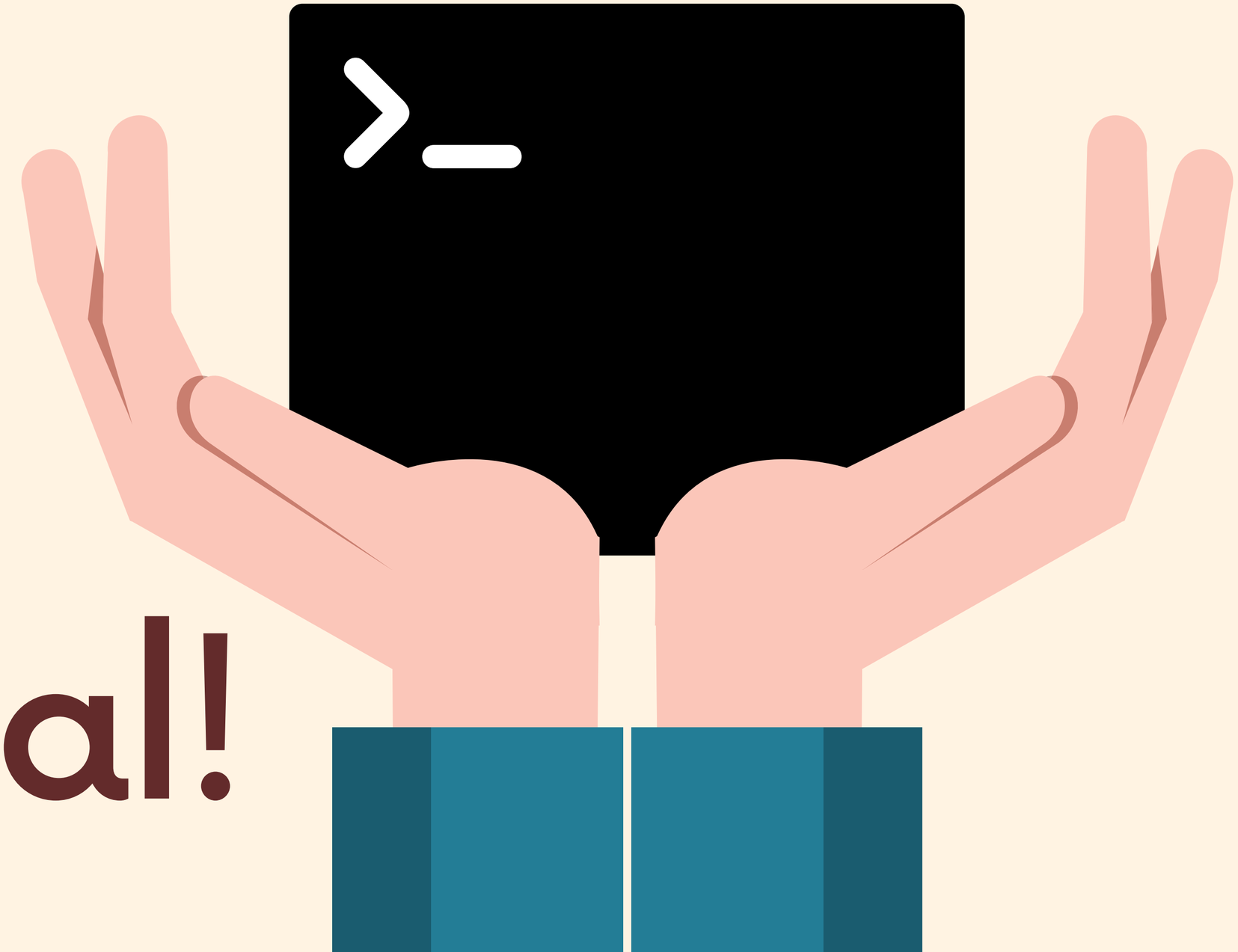
Bash runs on pretty much every version of Unix and Unix-like systems.



Command Basics

Step 1:

Open Terminal!



The Prompt

When we open up our terminal, we'll see our prompt which will likely include your **username@machinename**, followed by a ~ and then a dollar sign. We'll learn how to change the prompt later on.

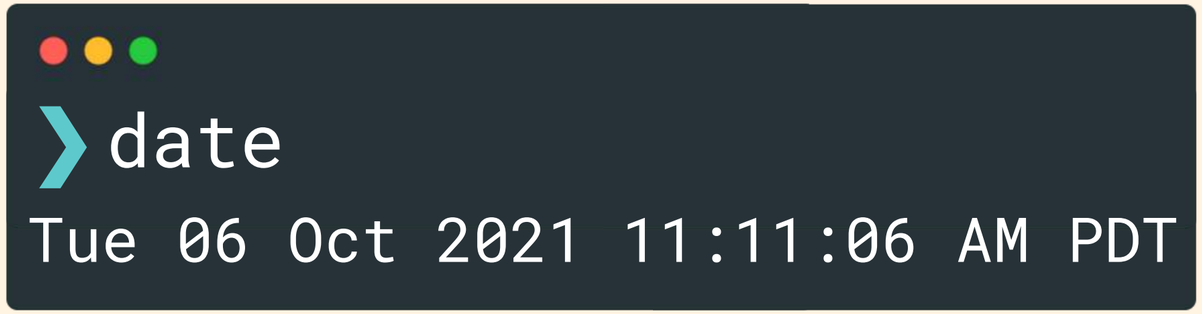
This prompt is what we'll see whenever the shell is ready to accept new input. **All we need to do is type some commands and hit enter.**

If we try typing some gibberish and hit enter, the shell attempts to find a command with that name before telling us "command not found".

Our First Command!

The date command may not be the most useful command of all time, but it's a great place to start.

Try typing date and then hit enter. You should see the current date printed out!



Case Matters!

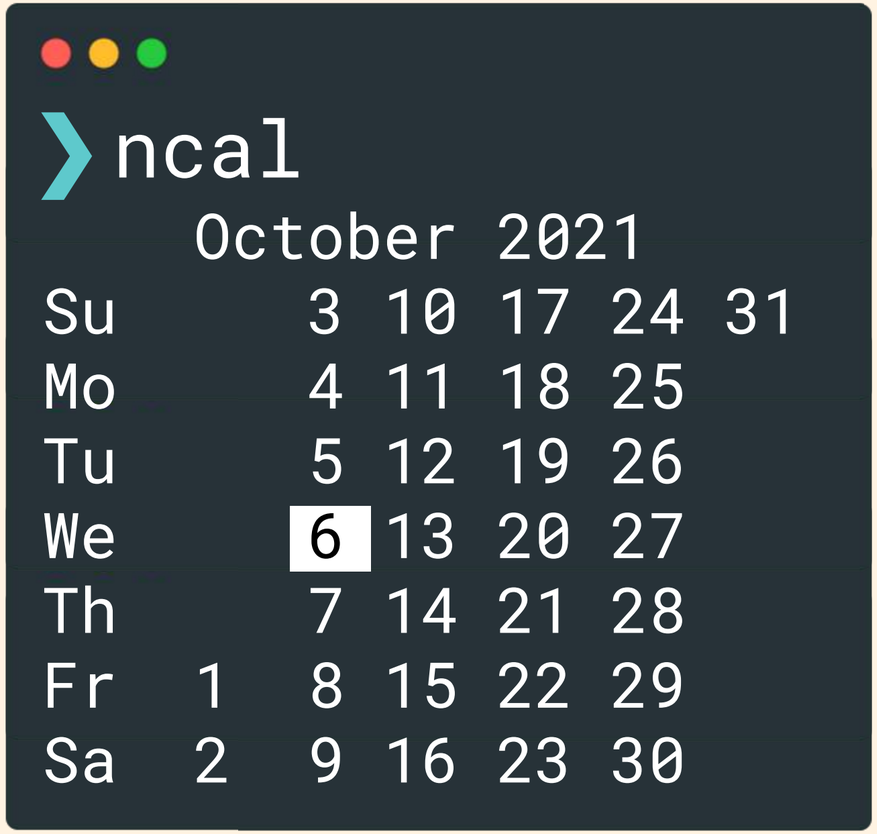
Commands are case sensitive, so Date is NOT the same thing as date.

* If you're using OS X, some commands are not case sensitive, but others are. It's safest to assume all commands are case sensitive.

Another Simple Command

Try typing ncal into your prompt. Hit enter and you should see the current month's calendar printed out.

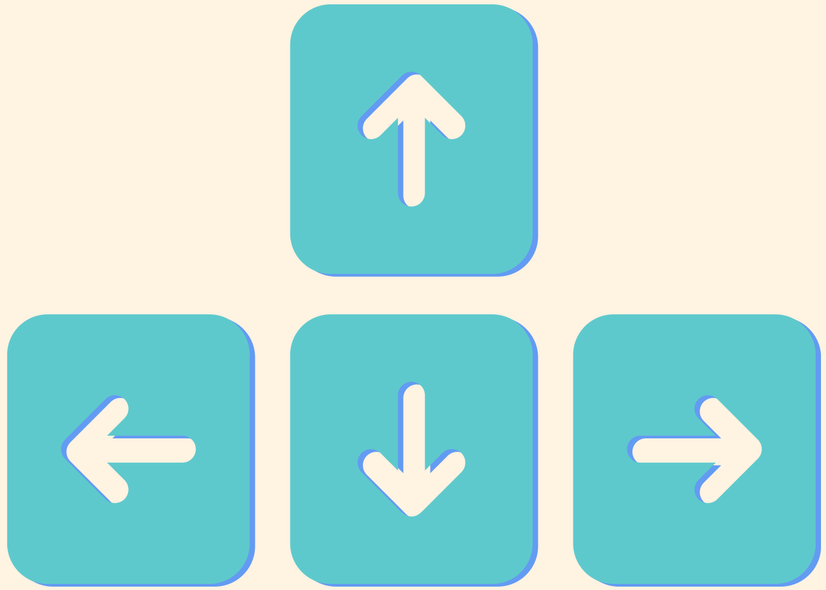
ncal stands for "new cal". There is also a "cal" command that does the same exact thing, but ncal adds some fancier functionality.



Use The Arrow Keys

In the terminal, we can use the left and right arrow keys to move through a line of text, one character at a time.

Use the up arrow to access previously entered commands, which can save you tons on typing!



Command Structure

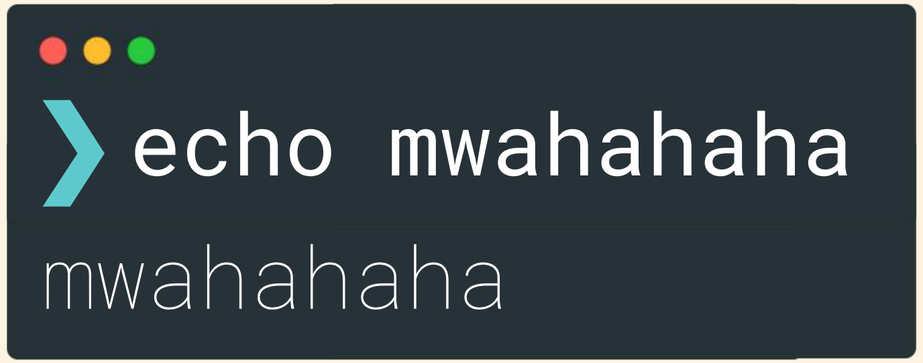
Most commands support multiple **options** that modify their behavior. We can decide which options to include, if any, when we execute a command.

Similarly, many commands accept arguments (the things that the command acts upon or uses)

Arguments

The terms "argument" and "parameter" are often used interchangeably to refer to values that we provide to commands.

The **echo** command is extremely simple. It takes the arguments we provide to it and prints them out. It echoes them back at us.

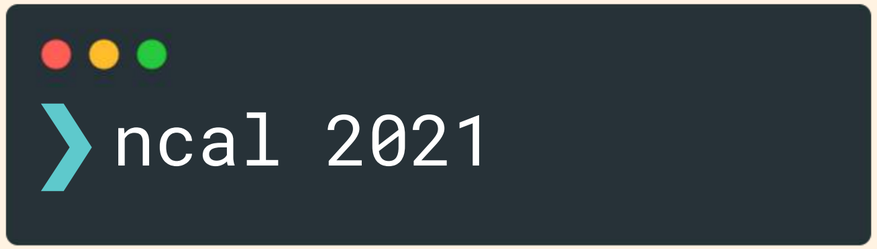


Arguments

The ncal command accepts values to control the specific month(s) and year it displays.

If we specify only a year, ncal will print out the calendar for that entire year.

If we specify a month and a year, ncal will print only that month's calendar.



Arguments

The **sort** command, which we will cover in depth later, accepts a filename. It prints out the sorted contents of that file.

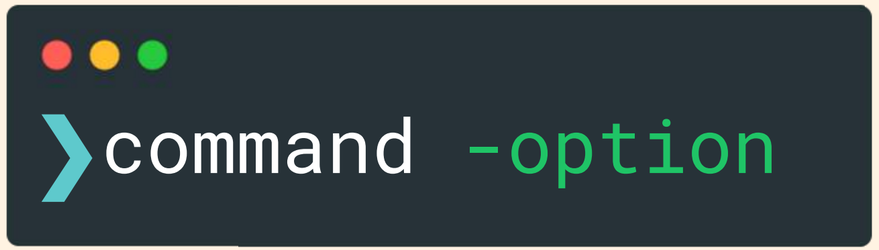
For example, **sort colors.txt** prints out each line of the colors.txt file, sorted in alphabetical order.

The specifics don't matter.
Focus on the PATTERN.

Options

Each command typically supports a host of options that we can choose to use when executing the command. These options modify the behavior of the command in predefined ways.

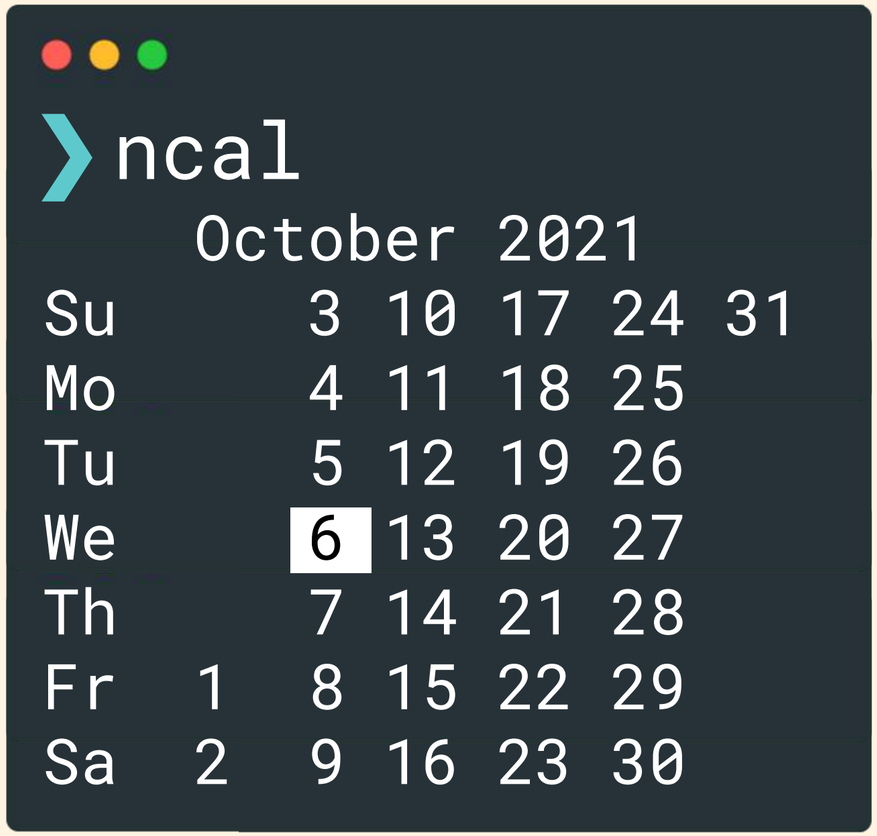
Options are prefixed by a dash, as in **-h** or **-3**.



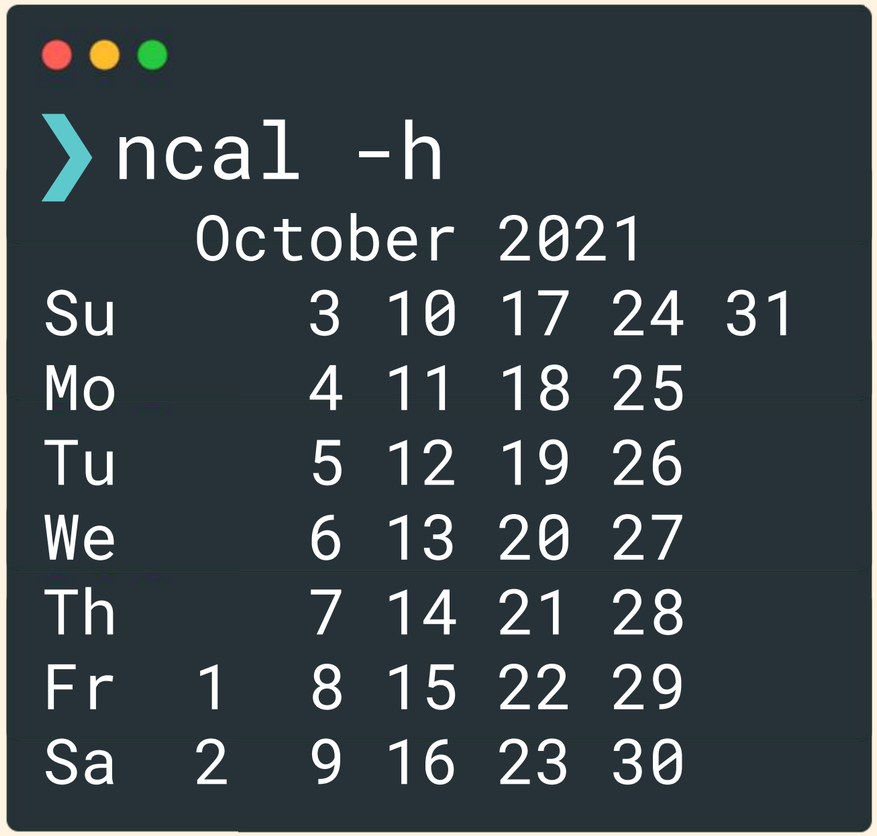
Please do not worry about remembering any of these options I'm about to show you.

Options

By default, the ncal command highlights today's date in the output.



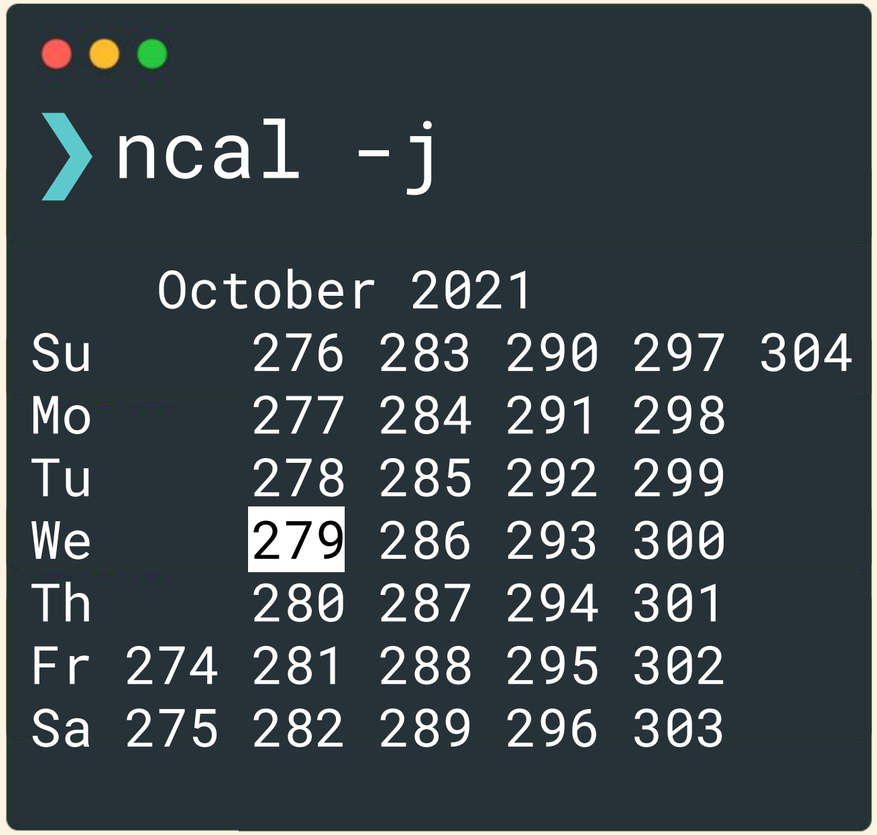
We can provide the -h option to turn off the highlighting of today's date.



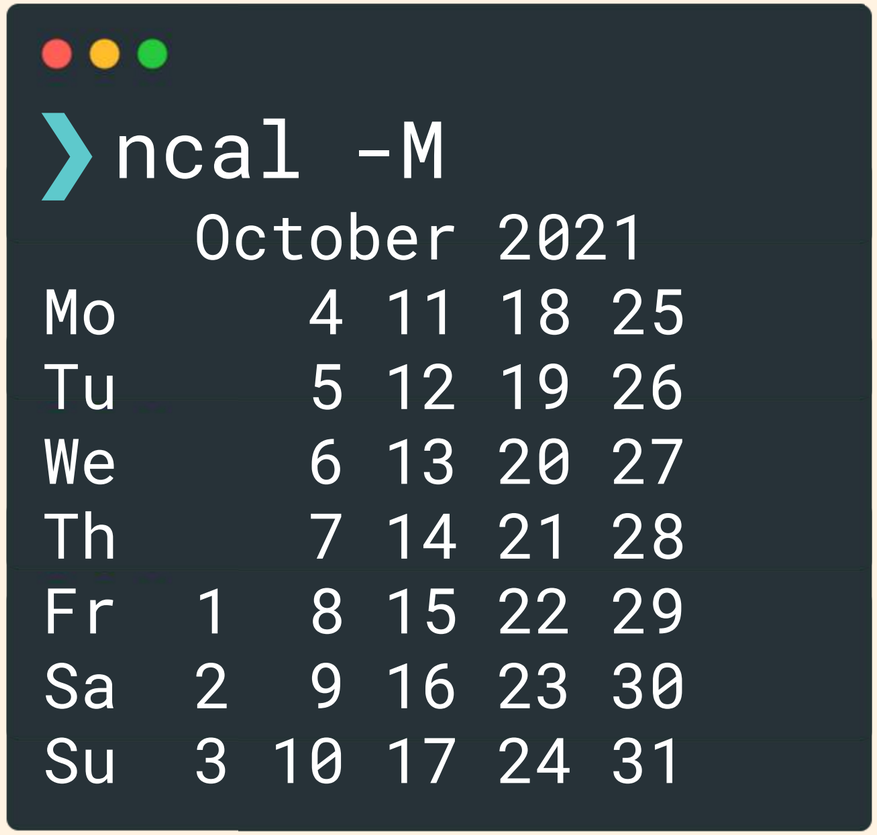
Case Matters

More Options

The **-j option** tells ncal to display a calendar using Julian days (days are numbered starting from jan 1st)

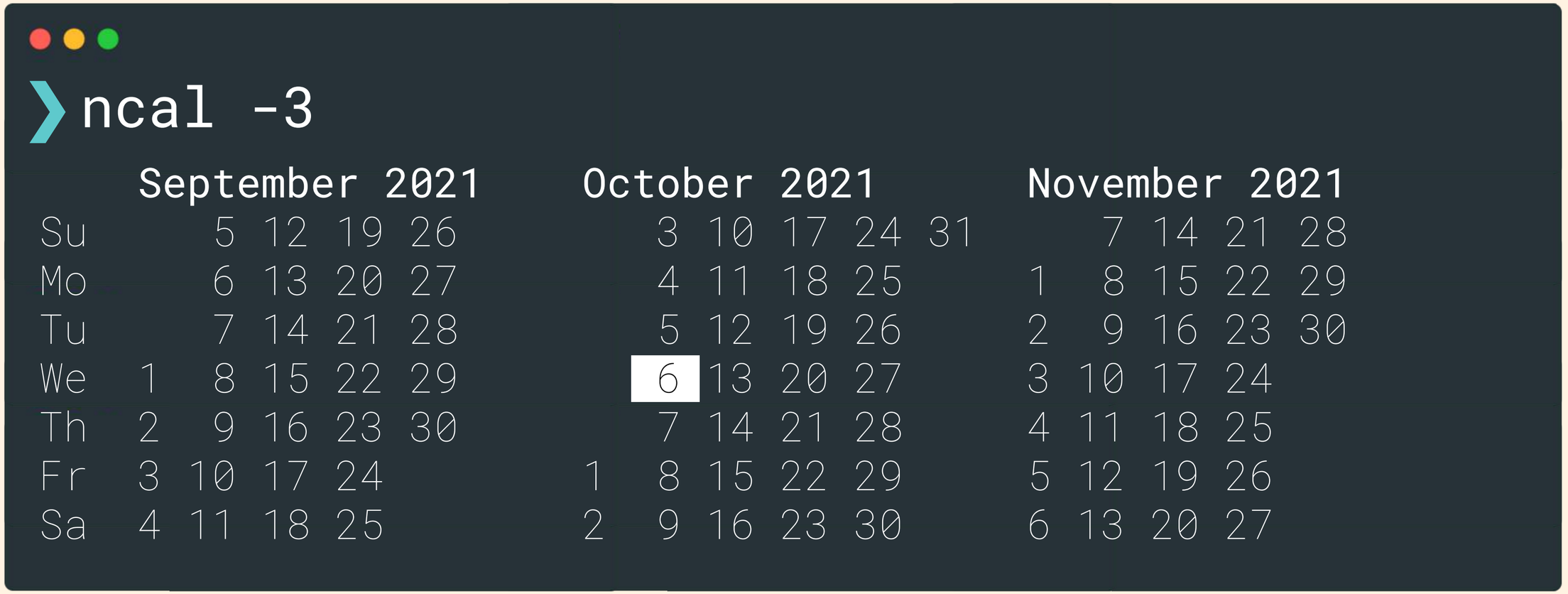


We can provide the **-M option** to tell ncal to use Monday as the first day of the week instead of Sunday.



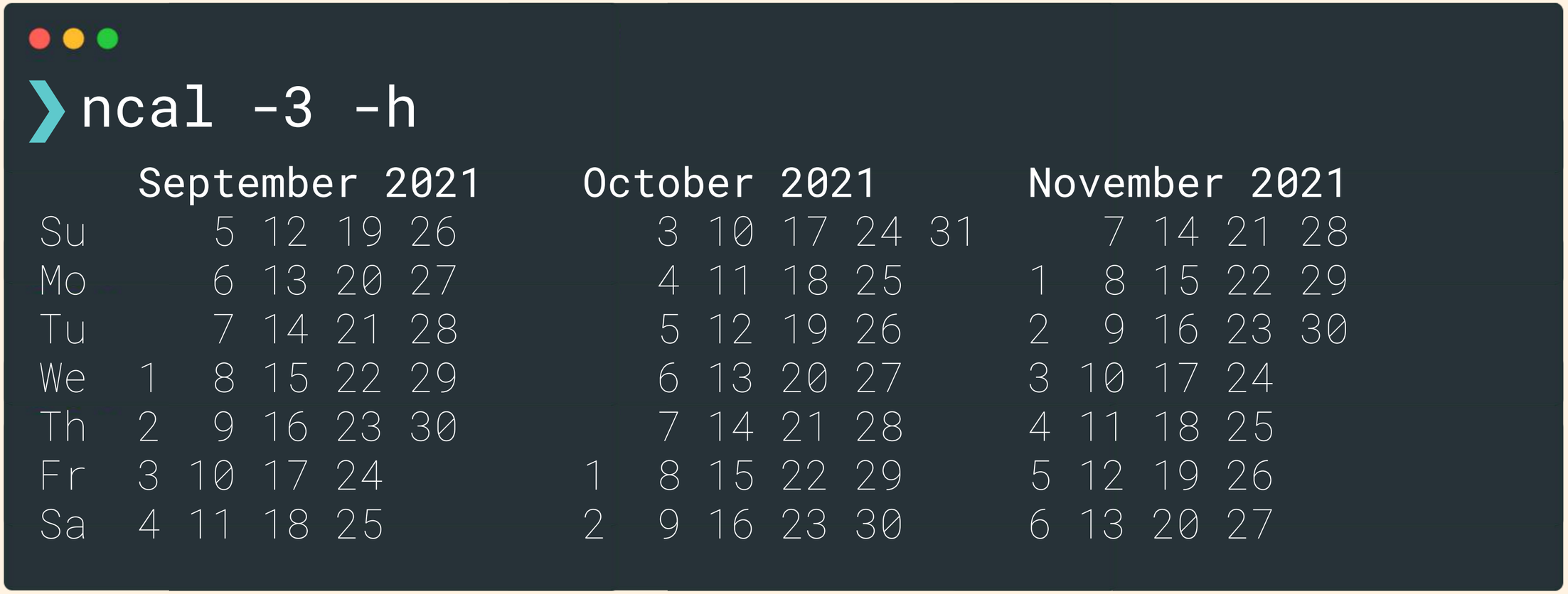
More Options

The -3 option tells ncal to display the previous, current, and next month



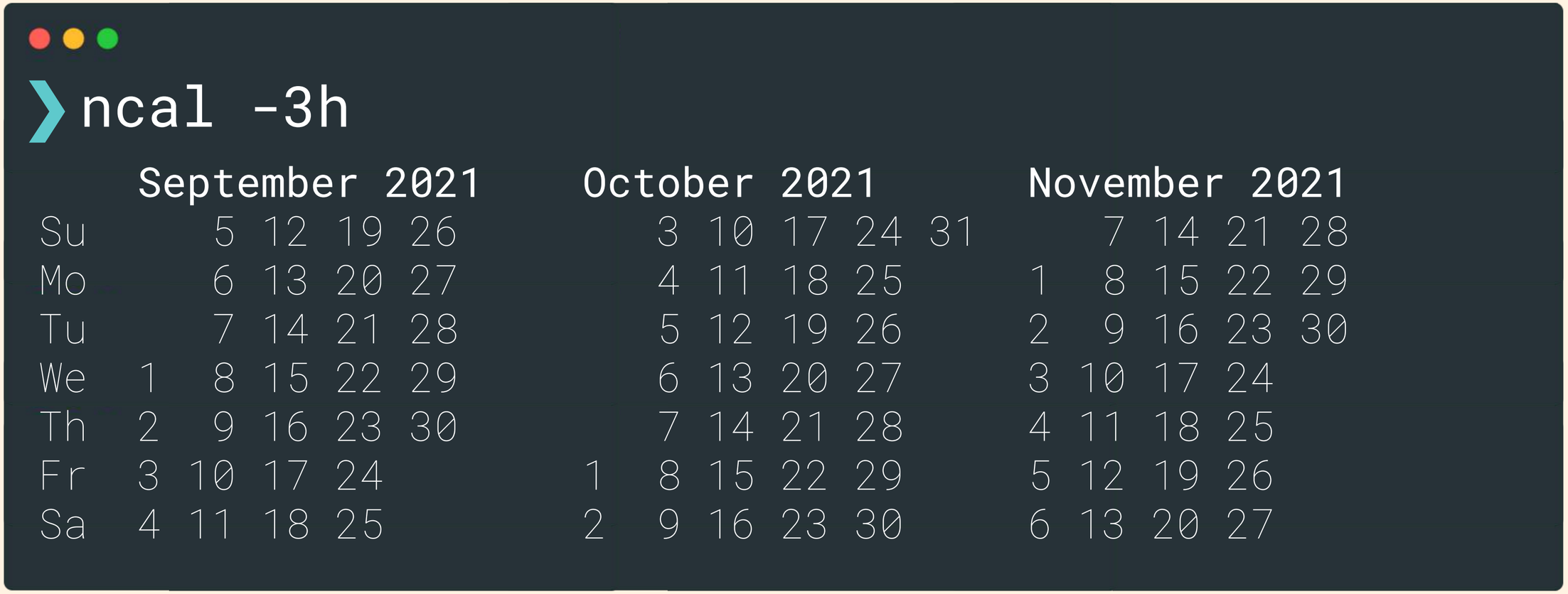
Combining Options

We can provide multiple options at once. This example uses the `-3` option to display the previous, current, and next month AND the `-h` option to turn off the highlighting of the current date.



Another Syntax

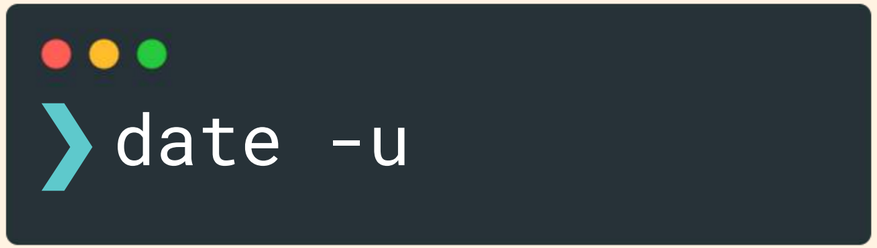
When we provide multiple options to a single command, we can use a shorter syntax where we only need a single dash (-) character

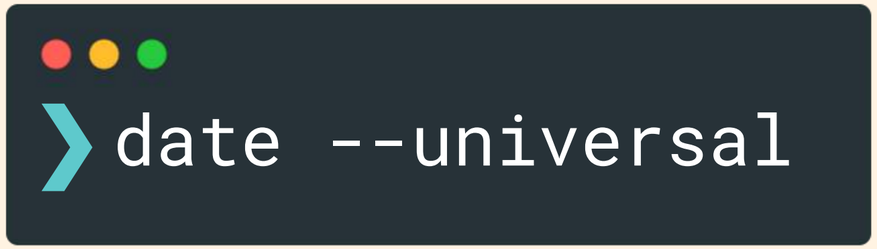


Long Form Options

All these short one-character options can get confusing! Some options also support equivalent long format options that are usually full words and are prefixed with two dashes instead of just one.

For example, the `date -u` option is used to print the date in Coordinated Universal Time (UTC). We can instead use `date --universal` to accomplish the same end result.





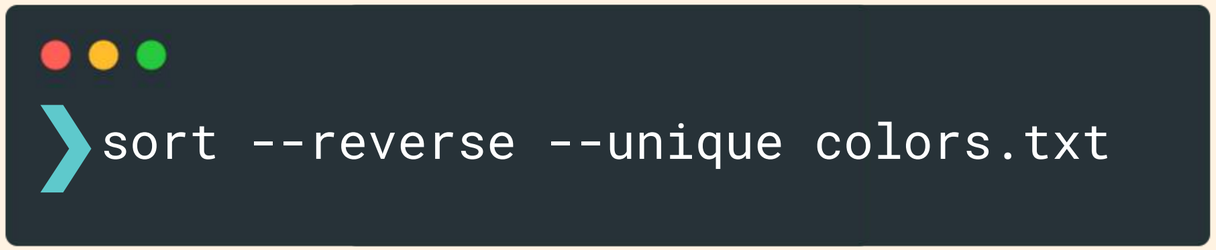
Long Form Options

Here's another example using the sort command (which we have not really covered yet)

The sort -r option will sort a files contents in reverse. If we prefer, we can use the longer form sort --reverse to accomplish the same thing.

Multiple Long Form Options

To use multiple long-form options in a single command, we must write them out separately with their own dashes (--). We cannot combine long-form options in the same way we can with single character options.

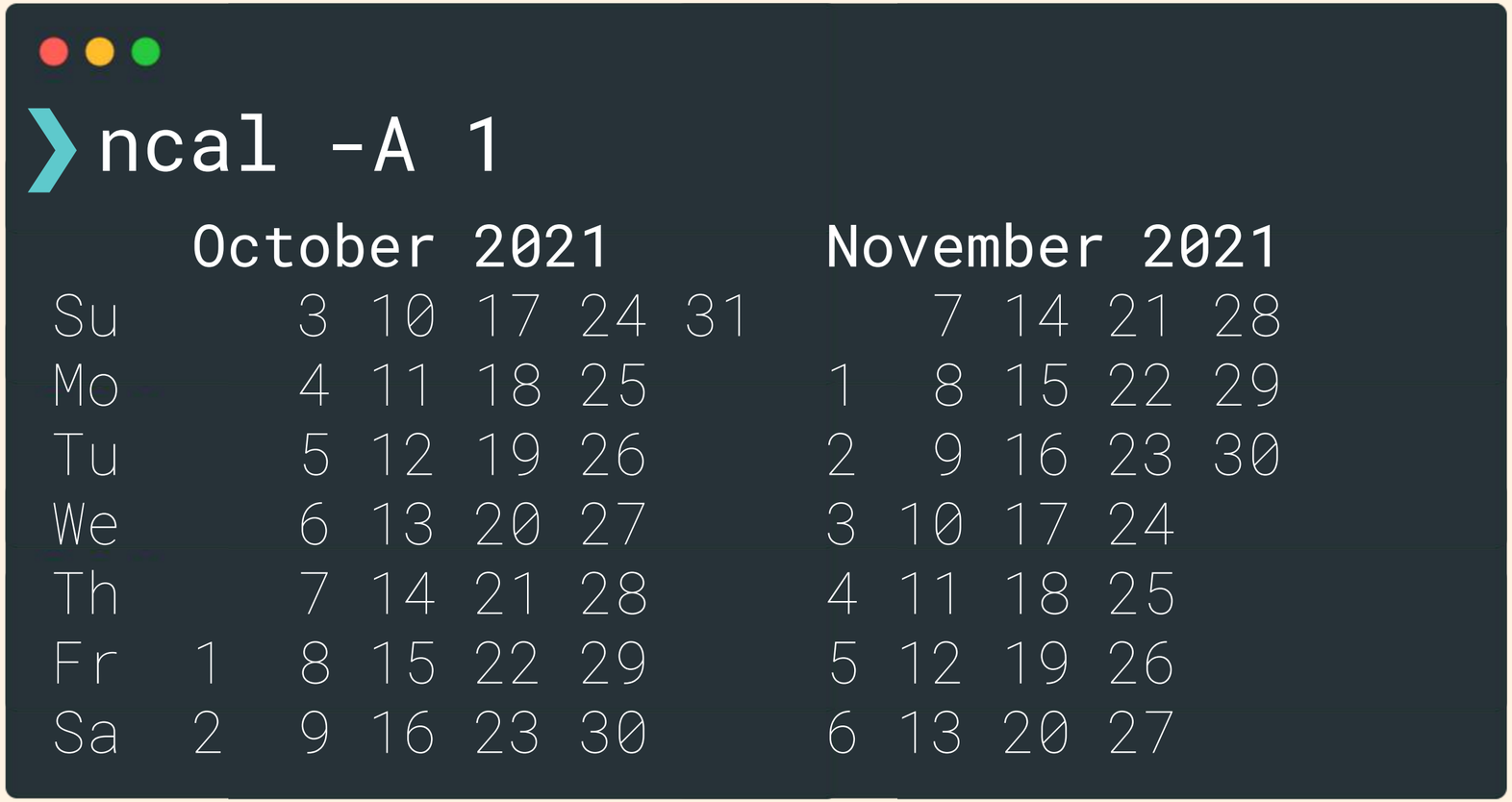


Options With Parameters

Some options require us to pass in an additional value. For example, ncal's **-A option** is used to display a certain number of months AFTER a specific date. We need to tell it how many months to display.

In this example, **ncal -A 1** prints out the current month (october) with one month afterwards (november)

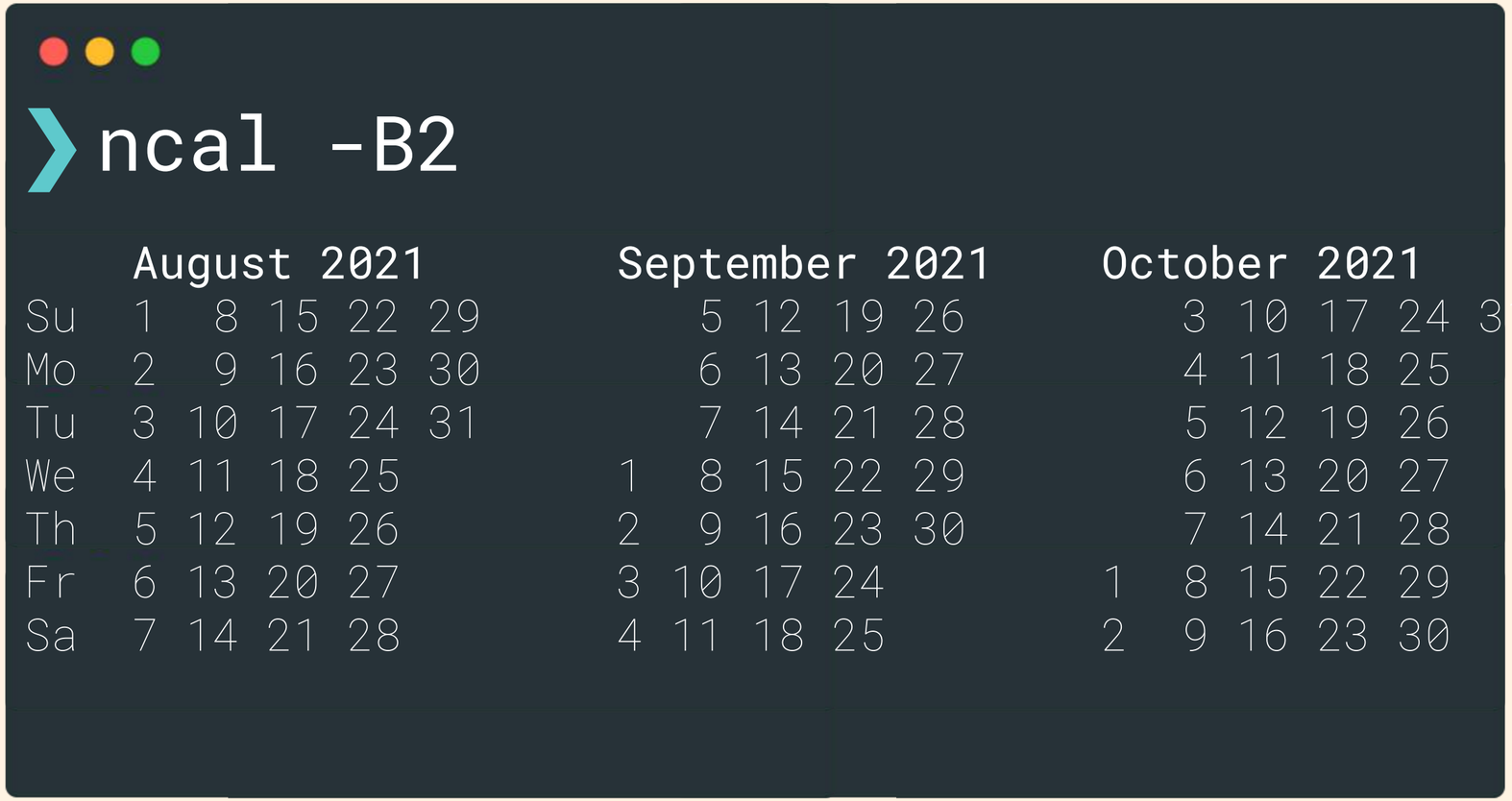
Note: this can also be written as **ncal -A1** (no space between A and 1)



Options With Parameters

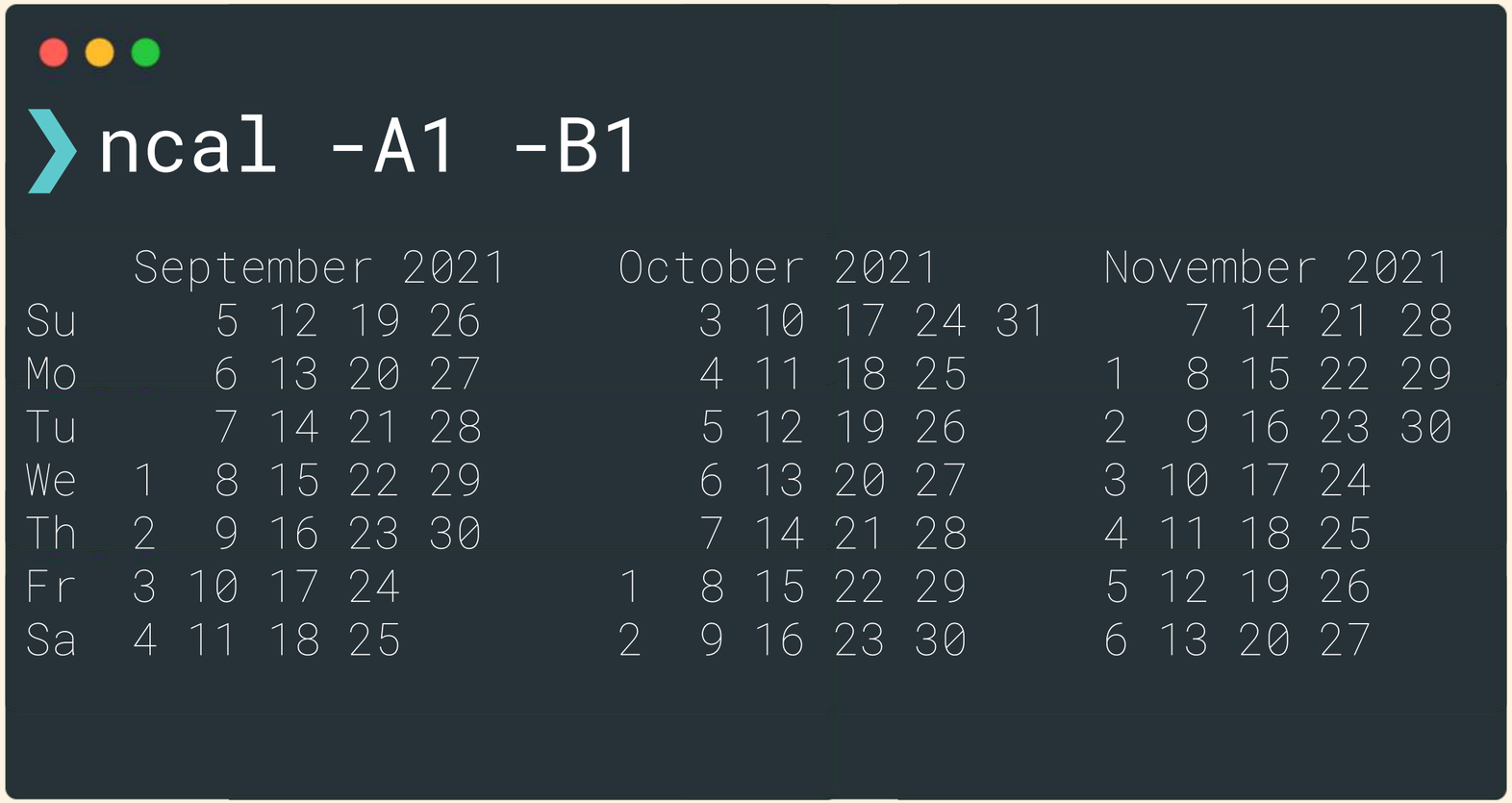
There is also a -B option to print a number of months BEFORE the specific date. We need to pass it a number of months.

In this example, **ncal -B2** prints out the current month (october) with the two previous months (august and september)



Options With Parameters

This example uses both the -A and -B options to print out 1 month before the current month AND one month after.



All Together Now

This example prints out the calendar for July 1969, with one month before (june) and two months after (august and september)

```
❯ ncal -B1 -A2 july 1969
     June 1969          July 1969          August 1969        September 1969
Su  1  8 15 22 29      6 13 20 27      3 10 17 24 31      7 14 21 28
Mo  2  9 16 23 30      7 14 21 28      4 11 18 25      1  8 15 22 29
Tu  3 10 17 24      1  8 15 22 29      5 12 19 26      2  9 16 23 30
We  4 11 18 25      2  9 16 23 30      6 13 20 27      3 10 17 24
Th  5 12 19 26      3 10 17 24 31      7 14 21 28      4 11 18 25
Fr  6 13 20 27      4 11 18 25      1  8 15 22 29      5 12 19 26
Sa  7 14 21 28      5 12 19 26      2  9 16 23 30      6 13 20 27
```

Don't worry about remembering the specific options!

Getting Help

-w ??

What is that?

man pages

The **man pages**, short for manual pages, are a built-in form of documentation available on nearly all UNIX-like operating systems.

The specific contents vary from one operating system to another, but at a bare minimum the man pages include information on commands and their usage.

man pages

To read the specific piece of documentation associated with a given command,
run **man command**

For example, to learn more about the ncal command we could run **man ncal**

This displays a bunch of information on ncal that we can scroll through. Type "q" to exit.

man pages contents

In general, each man page will follow this pattern:

- The title/name of the command with a short explanation of its purpose

- Synopsis of the command's syntax

- Description of the all the command's options

man pages synopsis

ncal [-31bhJeoSM] [-A number] [-B number] [-d yyyy-mm] [year]

Anything listed inside of square brackets is OPTIONAL. The only required part is **ncal.**

The above synopsis for ncal tells us that we can use the following options without providing any sort of additional parameter: -3, -1, -b, -h, -J, -e, -o, -S, and -M. To keep things brief, they are all lumped together as [-31bhJeoSM]

Then, we see other options in square brackets followed by their expected parameters: [-A number] means the -A option expects a number. [-d yyyy-mm] indicates that the -d option expects us to pass in a date in the format: yyyy-mm like 1980-04

Finally, at the end we see [year] which means that we can pass a year as a parameter

man pages synopsis

echo [OPTION]... [STRING]...

The above synopsis is for the **echo** command, which echoes text back at us.

An ellipsis (...) indicates that one or more of the proceeding operand are allowed:

[OPTION]... means that we can pass more than one option to echo

[STRING]... indicates that we can pass multiple strings to echo. For example, we can run **echo hello** and also **echo hello there you cutie little chicken pot pie**

man pages synopsis

cp [OPTION]... SOURCE DEST

So far, all of the operands we've seen have been optional, but some commands do require certain arguments in order to run. In a man page synopsis, required operands are NOT wrapped in square brackets.

In the above synopsis for the copy (cp) command, we see that we can optionally provide one or more options. SOURCE indicates that we must pass one source, and DEST indicates that we must pass a destination as well. Those two arguments are required.

Manual Sections

The manual is broken into 8 different sections, each covering a specific topic in depth:

1. User Commands
2. System Calls
3. C Library Functions
4. Special files
5. File forms
6. Games
7. Miscellaneous
8. System admin commands

Searching The Manual

```
❯man -k dog
rtkitctl (8) - Realtime Watchdog daemon control
wdctl (8) - show hardware watchdog status
```

We can search for a term within the manual using the **-k** option.

For example, to search the manual for "dog" we would run **man -k dog**

Types Of Commands

There are really four types of commands:

- An executable program, usually stored in /bin, /usr/bin, or /usr/local/bin. These are compiled binary files (hence bin)
- A built-in shell command. These commands are part of the shell (bash in our case)
- A shell function
- An alias

The type command will tell us... the type of a command

Which

To find the exact location of an executable, run **which command**

This only works for executables, not built-in shell commands or aliases.

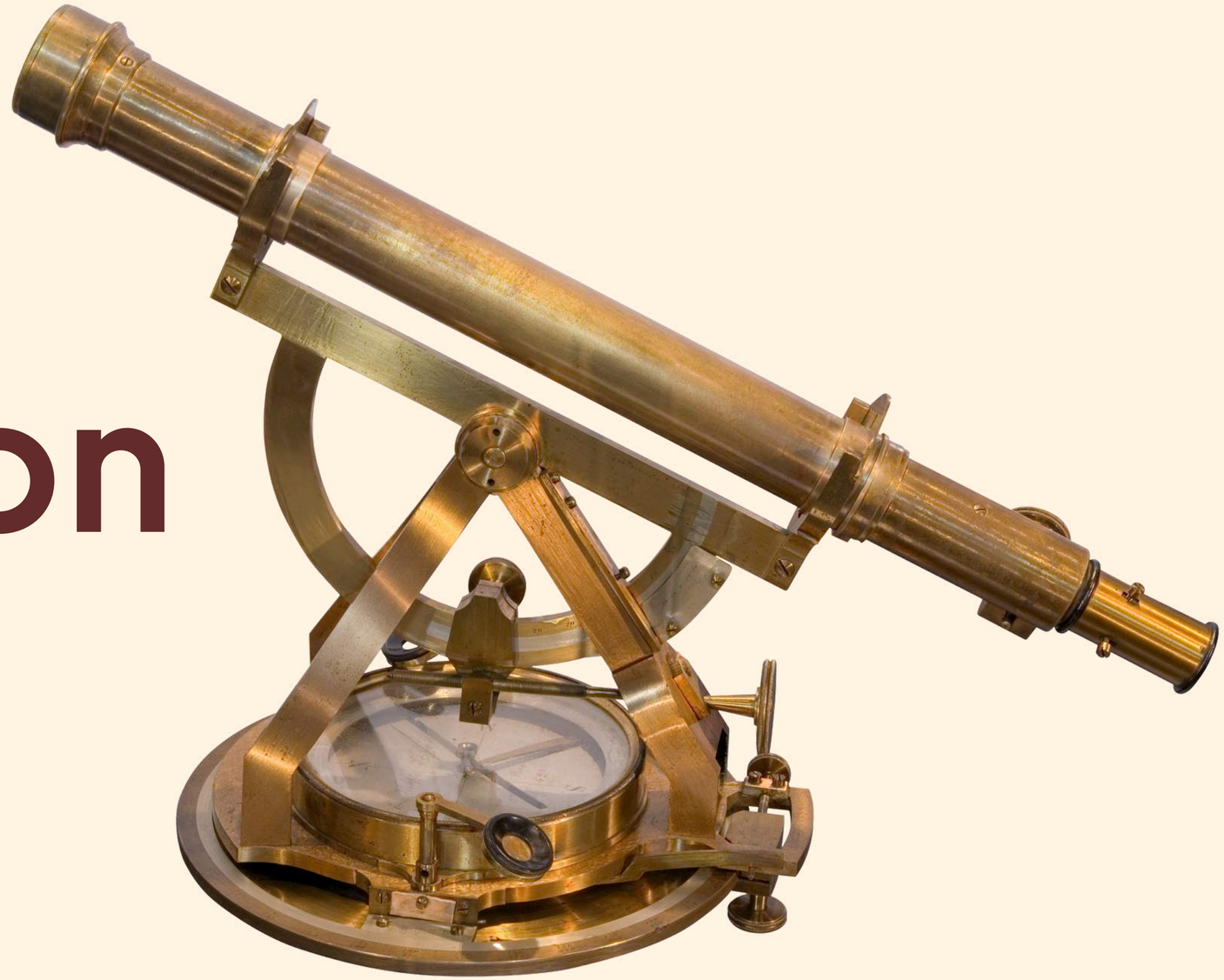
Help!

Some commands do not have man pages written for them, because they are commands that are directly built in to the shell.

We can find documentation for those commands using the **help** command.

For example, to learn more about the cd command we would run **help cd**

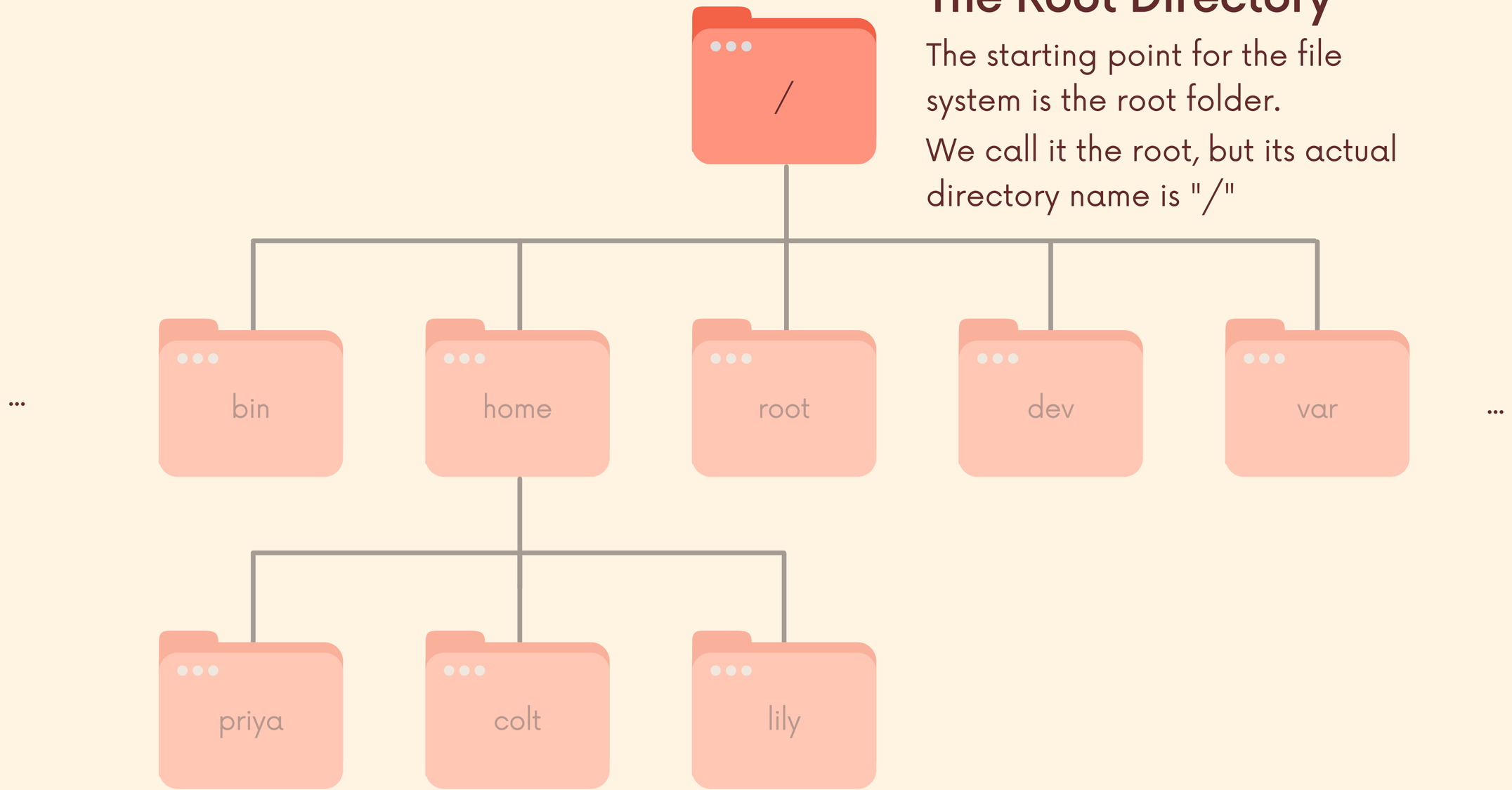
Navigation



The Root Directory

The starting point for the file system is the root folder.

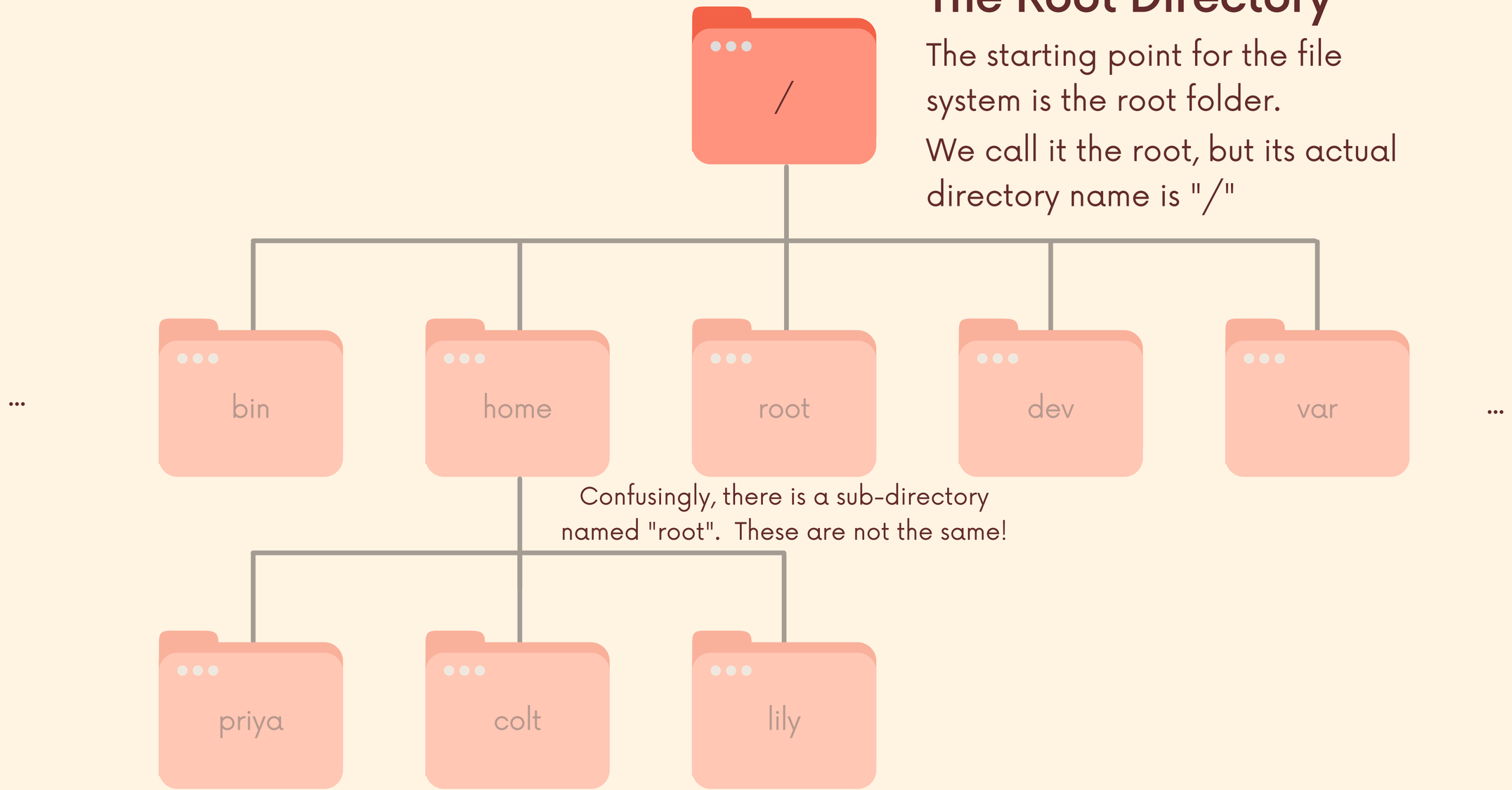
We call it the root, but its actual directory name is "/"

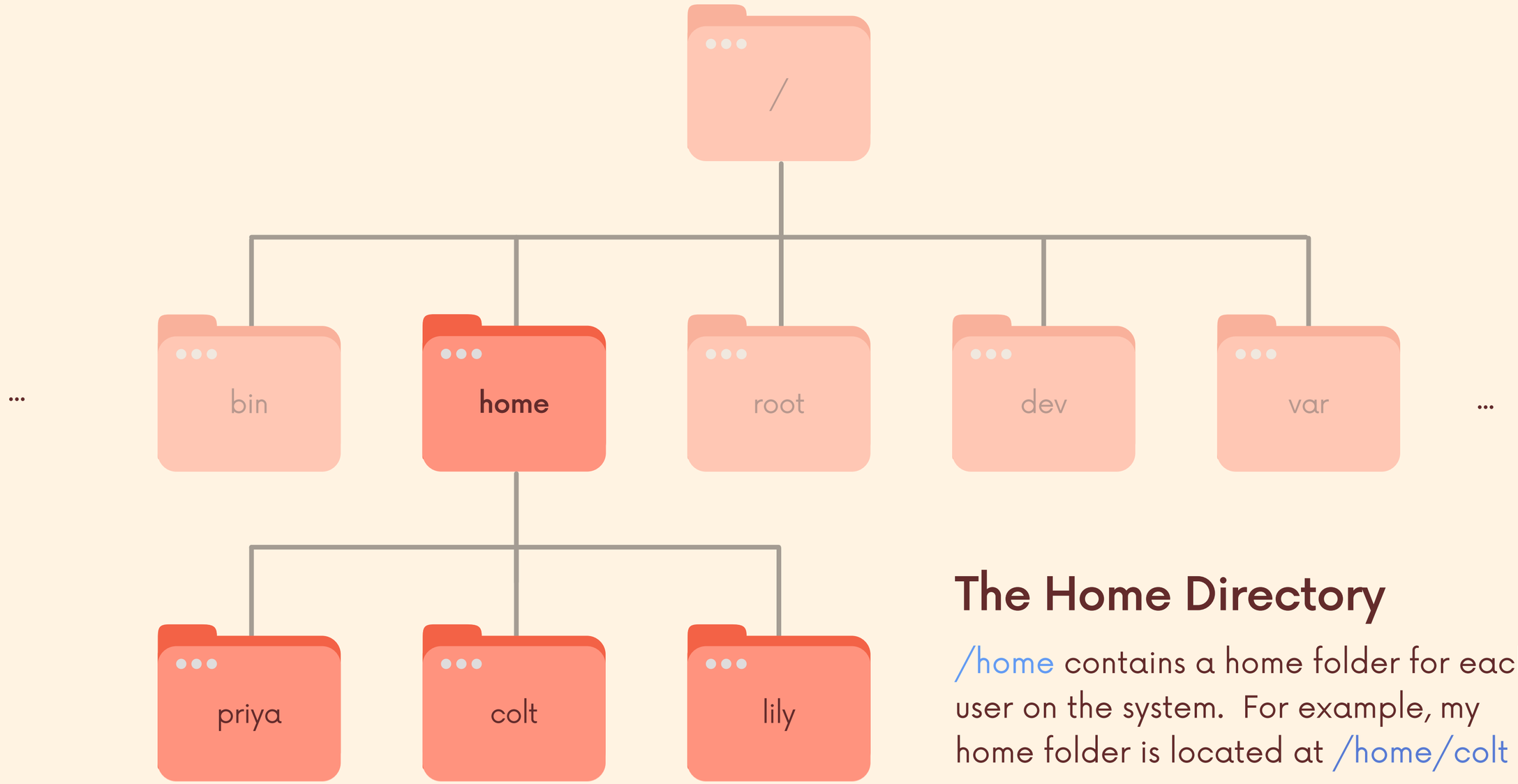


The Root Directory

The starting point for the file system is the root folder.

We call it the root, but its actual directory name is "/"





The Home Directory

/home contains a home folder for each user on the system. For example, my home folder is located at /home/colt

/ - root

~ - home

pwd

The **print working directory** command is super simple but very useful. Think of it as a "where am I" command.

It will print the path of your current working directory, starting from the root /

For example, if I were on my desktop and I ran pwd, I would see /home/colt/Desktop

"where am I?"

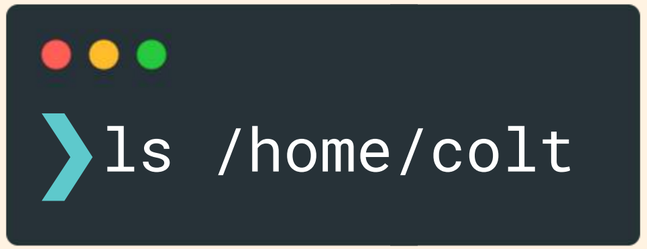
ls

The list command will list the contents of a directory.

When used with no options or arguments, it prints a list of the files and folders located in the current directory.

We can also list the contents of a specific directory using **ls path**. For example **ls /bin** will print the contents of the /bin directory

list contents of current directory



list contents of /home/colt

ls options

The ls command accepts a ton of options.
Two of the more commonly used are -l and -a

-l (lowercase L) prints in long listing format. It shows far more information about each file/folder.

the -a option will also list any hidden files that begin with . These are normally not listed.

We can combine options! This example prints detailed information for all files, including hidden files.

cd

The **cd** command is used to change the current working directory, "moving" into another directory.

For example... `cd chickens` would change into the chickens directory (assuming it exists)

`cd /home/colt` would take me my home directory

backing up

In Unix-like operating systems, a single dot (.) is used to represent the current directory. Two dots (..) represent the parent directory.

So we can use **cd ..** to move up one level, from our current directory into the parent directory.

"back up" one level into the parent folder

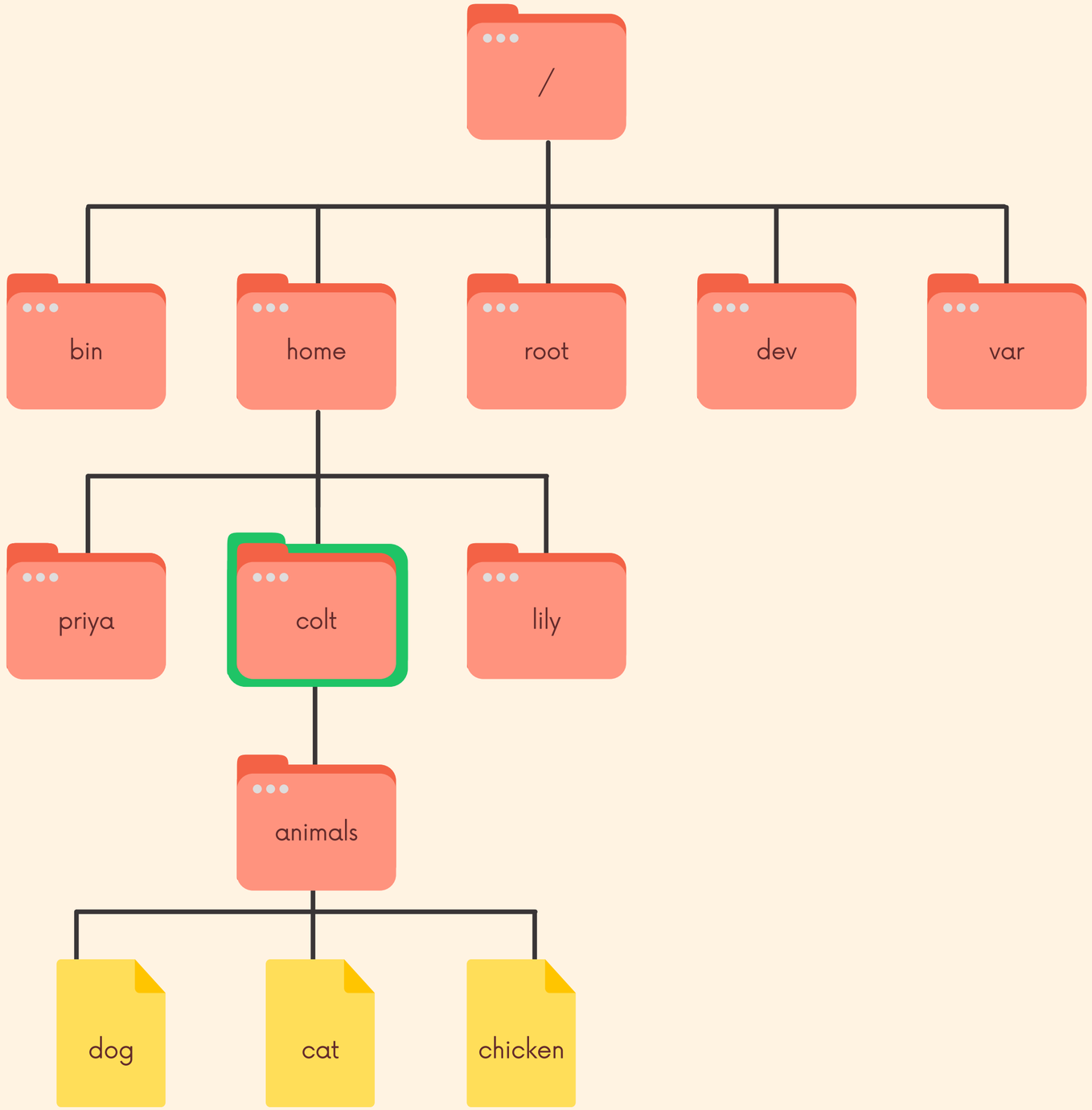
Relative Paths

When providing paths to commands like cd or ls, we have the option of using relative or absolute paths.

Relative paths are paths that specify a directory/file relative to the current directory.

For example, if our current directory is **/home/colt** and we want to cd into animals, we can simply run cd animals.

However, cd animals does NOT work if we are located in another directory like **/bin**. The relative path from /bin is ../home/colt/animals

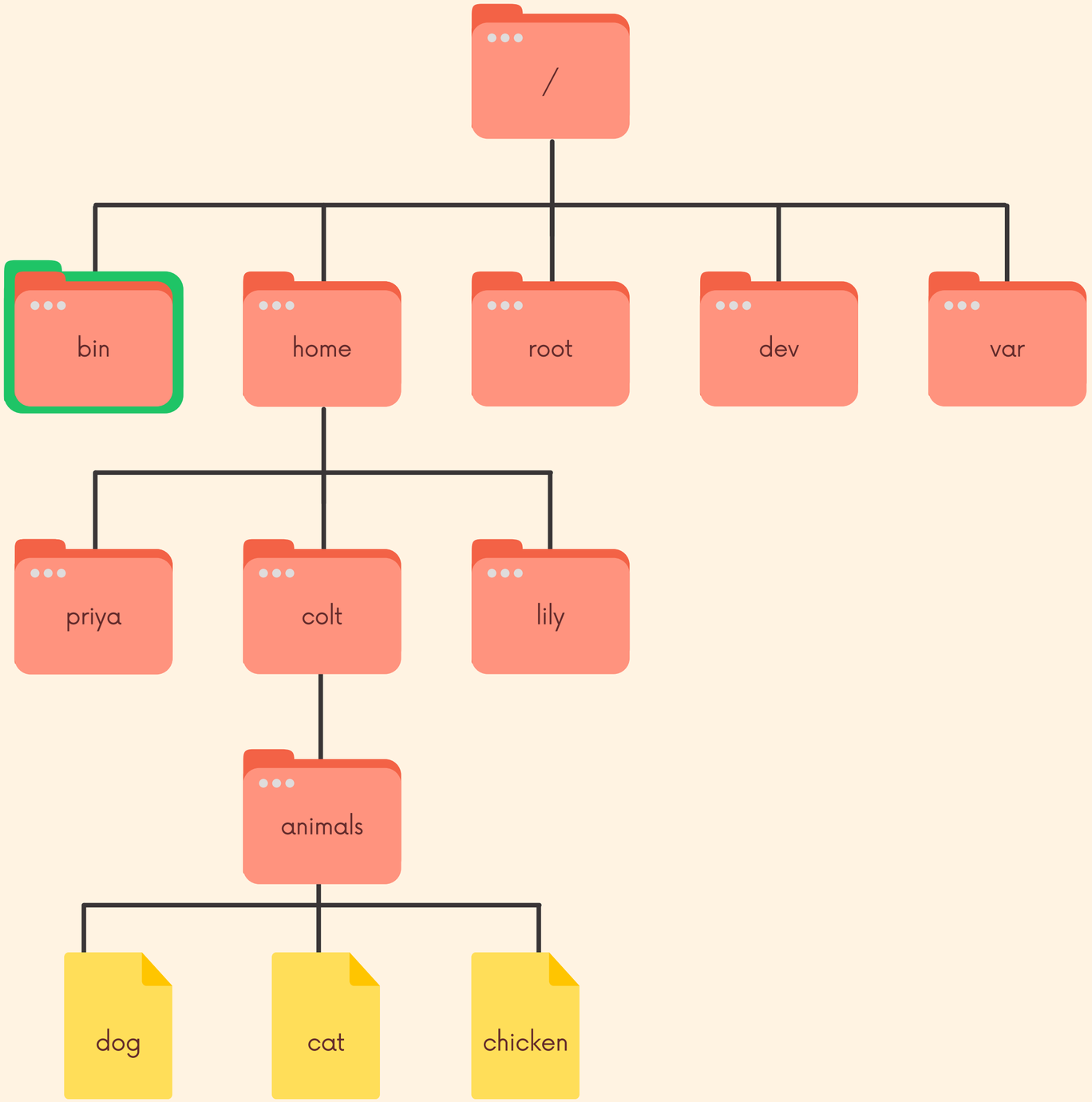


Absolute Paths

Absolute paths are paths that start from the root directory (they start with a /)

The absolute path to the animals directory is /home/colt/animals. We can use absolute paths to specify a location no matter our current location.

For example, from the /bin directory I could use **cd /home/colt/animals** to change into the animals directory.



Creating Files & Folders

touch

To create a new file from the command line, use the **touch** command. Provide a filename, and that file will be created for you (assuming it doesn't already exist)

For example, **touch chicken.txt** would create a chicken.txt file in the current directory.

If you try to use touch with a file that already exists, it will simply update the access and modification dates to the current time.

file

The file command can be used to determine the file type of a specified file. It's honestly not that important!

For example, running file contract.pdf will tell us the file type of contract.pdf
"conctract.pdf: PDF document, version 1.4"

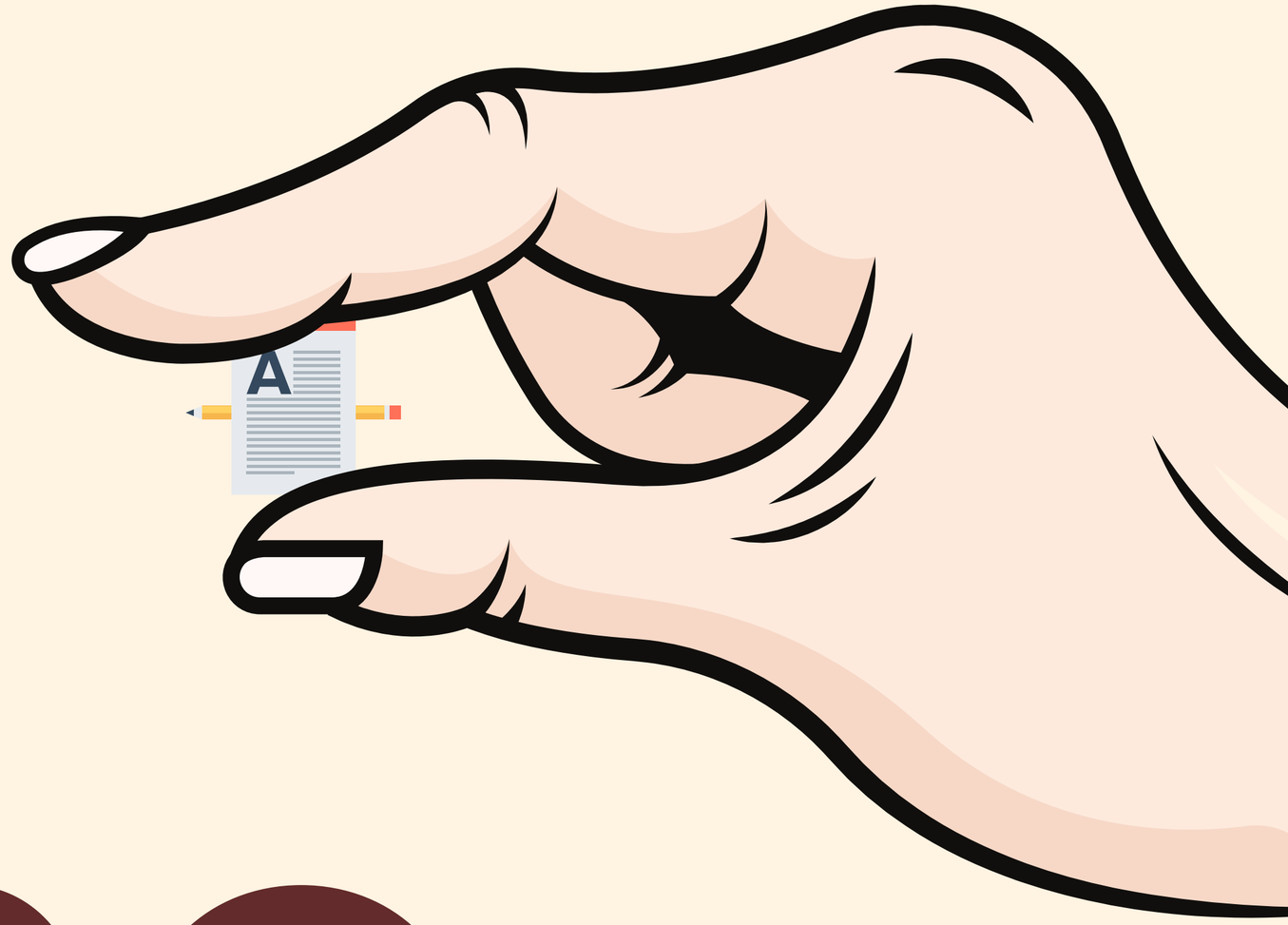
Note, it does not use the file extension to "decide" on the file type. We could have a pdf file named app.png

mkdir

To create new directories, we use the make directory (mkdir) command. We provide one or more directory names, and it will create them for us.

For example, to create two new folders, images and styles, we could run **mkdir images styles**

Nano



Nano

Nano is a simple text editor that we can access right from the terminal. It's far more accessible than other popular command-line editors like vim and emacs.

Nano includes all the basic text editing functionality you would expect: search, spellcheck, syntax highlighting, etc.

Nano

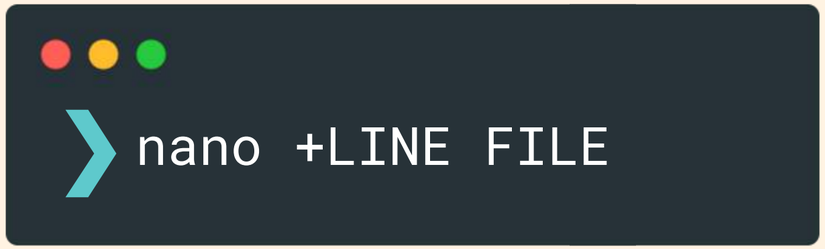
To open up a file using nano, run **nano FILE**. For example, to open up the file book.txt using nano, we would run **nano book.txt**

We can also use the same command to edit a file that doesn't yet exist (we can then save it and create the file).

Edit At A Specific Line

We can also provide nano with a specific line number to position the cursor at using **nano +LINE FILE**.

To open up the book.txt file at line 205, we would run **nano +205 book.txt**

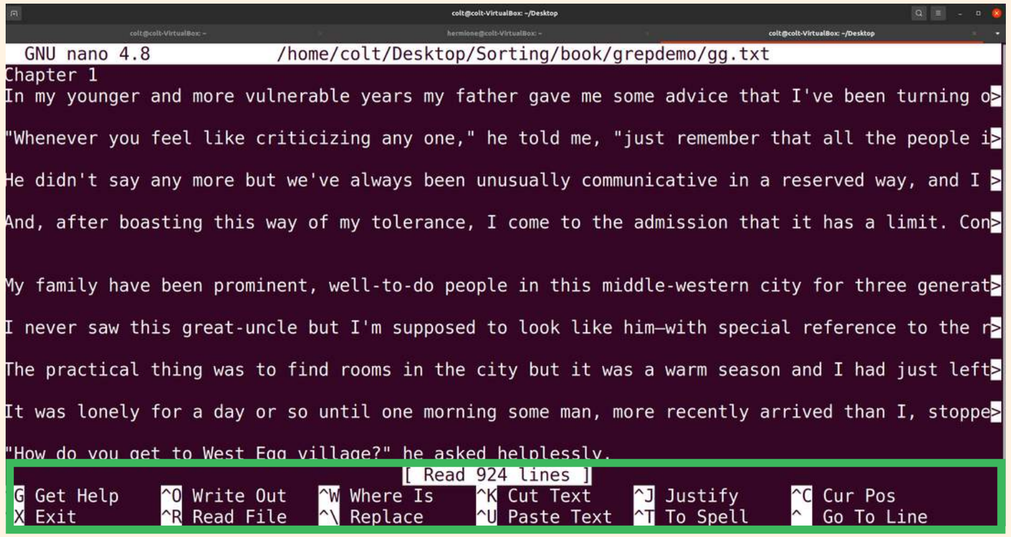


Editing With Nano

Some editors like vim require us to enter "write mode" before we can start editing a file, but with nano we can make changes right away.

We can move the cursor using the arrow keys.

At the very bottom, we'll see a **list of shortcuts** that we can use inside of nano. These are super useful!



Saving

To save, we need to "write out" using the **Ctrl+O** command. Then, nano will prompt us to enter the filename we want use (just hit enter to keep the original name).

Exiting

To exit, use **ctrl** + X

Searching

Use **ctrl-W** and then enter a search phrase to search FORWARD in the file from your current cursor location.

Replacing

To search and replace, use **ctrl+** and then enter the word you want to replace. Then enter the replacement and decide whether to replace specific matches or all matches.

Spellchecking

We can use spellchecking inside of Nano, but we have to enable it first in the Nano config file located at **/etc/nanorc**

Deleting, Moving And Copying

rm

We use the remove (rm) command to remove files from our machine.

For example, rm app.js would remove the app.js file.

Note: rm DELETES FILES, there is no undo or recycling bin to retrieve them from! They are gone!

deleting folders

To delete empty folders, we need to use the **-d** option with rm. For example, **rm -d cats** would remove the cats directory (only if it's already empty)

To delete folders that are NOT empty, use the -r option. For example, **rm -r chickens** would delete the chickens directory whether it's empty or not.

You definitely want to be careful when deleting directories!

moving stuff

Use the move command (mv) to move files and directories from one location to another.

When we specify a file or files as the source and a directory as the destination, we are moving the files into the directory.

For example, `mv app.css styles/` will move the app.css file into the styles directory.

moving stuff

Use the move command (mv) to move files and directories from one location to another.

When we specify a file or files as the source and a directory as the destination, we are moving the files into the directory.

For example, `mv app.css styles/` will move the app.css file into the styles directory.

renaming

We can also use the move command to rename files and folders.

If we specify a single file as the source and a single file as the destination, it will rename the file. For example, to rename the chickens.txt file to roosters.txt, we could run **mv chickens.txt roosters.txt**

If we specify a single folder as the source and the destination doesn't yet exist, it will rename the folder. If the destination folder does exist, it will move our source folder into the destination.

copying

We can use the copy command to create copies of files and folders.

To create a copy of sheep.txt called dolly.txt, we could run cp sheep.txt dolly.txt

To copy multiple files into another directory, use cp file1 file2 directory.

Shortcuts

Shortcuts

We can speed up our command-line experience by taking advantage of the many built-in shortcuts.

These shortcuts are designed so that your hands never have to leave your keyboard "home base"

They take a little bit of practice to get comfortable with, but it's worth the effort!

Clearing

Use ctrl-l to clear the entire screen

Line Jumping

Use **ctrl-a** to move the cursor to the beginning of the line.

Use **ctrl-e** to move the cursor to the end of the line.

Moving Characters

Use **ctrl-f** to move the cursor forward one character at a time (same as the right arrow)

Use **ctrl-b** to move the cursor backwards one character at a time (same as the left arrow)

Jumping Words

Use alt-f to move the cursor forward one word.

Use alt-b to move the cursor backwards one word.

Swapping

Use **ctrl-t** to swap the current character under the cursor with the one preceding it. This can be useful to correct typos made by typing to quickly!

Killing The Line

Use **ctrl-k** to kill the text from the current cursor location until the end of the line. →

Use **ctrl-u** to kill the text from the current cursor location to the beginning of the line. ←

Killing A Word

Use **alt-d** to kill the text from the current cursor location through the end of the word →

Use **ctrl-w** or **alt-delete** to kill the text from the current cursor through the beginning of the word ←

Reviving Text (Yanking)

When we kill text using commands like ctrl-k, ctrl-u, alt-d, and alt-backspace, the "killed" text is stored in a memory in an area known as the "kill-ring"

We retrieve the most recently killed text using ctrl-y.

History

Bash keeps a record the command we have previously entered. We can see the actual file at **~/.bash_history**.

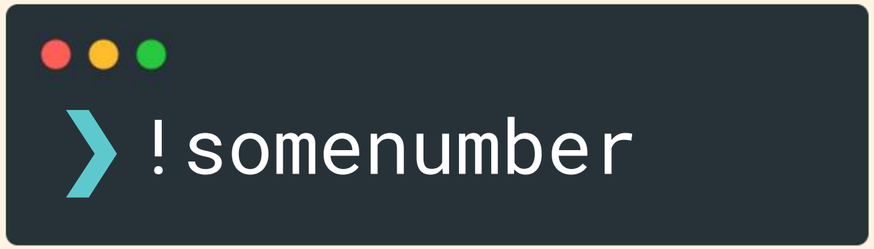
You can scroll through the history one command at a time using the up and down arrows.

We can also use the **history** command to view the entire history, though it's generally easier to manage if we pipe the output to less.

History Expansion

We can easily re-run an earlier command if we have its line number from the history.

For example, to run the 73rd command in the history, we could run !73



Searching History

Often it's easiest to find an earlier command by searching using a small portion of the command that you remember.

Type **ctrl-r** to enter "reverse-i-search". As you start typing, bash will search the history and show you the best match.

Hit ctrl-r to cycle through potential matches.

Working With Files

less

The less command displays the contents of a file, one page at a time. We can navigate forwards and backwards through the file, which is especially useful with very large files.

less somefile.txt will display the contents of somefile.txt using less.

less navigation

When viewing a file using less...

- press space or f to go to the next page of the file
- press b to go back to the previous page
- press Enter or Down arrow to scroll by one line
- to search, type forward slash / followed by a pattern
- press q to quit

cat

The cat command con**cat**enates and prints the contents of files.

cat <filename> will read the contents of a file and print them out. For example, **cat instructions.txt** will read in from the instructions.txt file and then print the contents out to the screen.

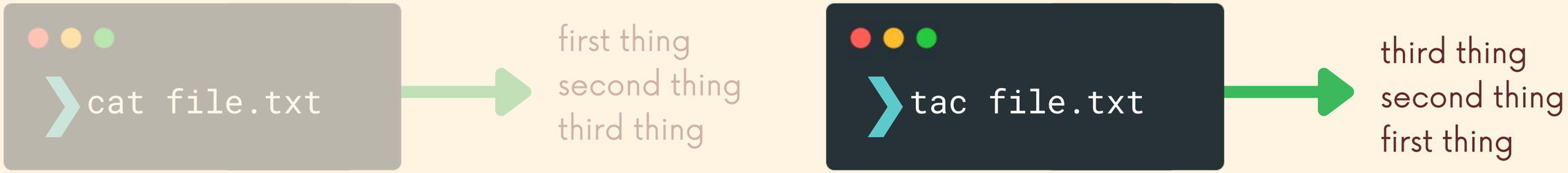
cat cont'd

If we provide cat with multiple files, it will concatenate their contents and output them.

cat peanutbutter.js jelly.css will output peanutbutter.js first and immediately after print the contents of jelly.css

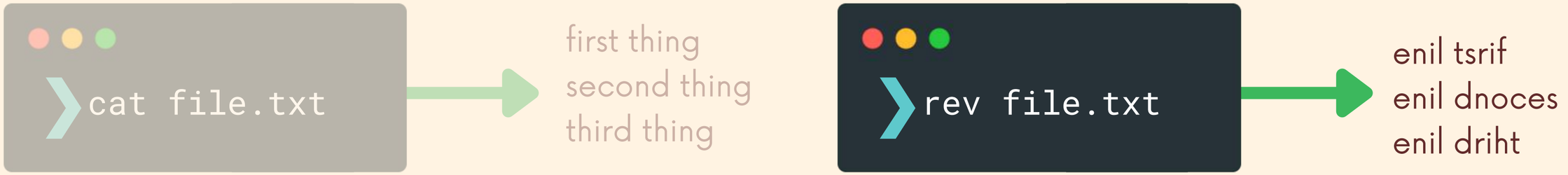
tac

tac (cat spelled backwards) will concatenate and print files in reverse. It prints each line of a file, starting with the last line. You can think of it as printing in reverse "vertically"



rev

the rev command prints the contents of a file, reversing the order of each line. Think of it as a "horizontal" reverse, whereas tac is a "vertical" reverse.



head

The head command prints a portion of a file, starting from the beginning of the file. By default, it prints the first 10 lines of a file.

head warAndPeace.txt would print the first 10 lines of the warAndPeace.txt file

head contd

We can also specify a number of lines for head to print using the -n option (or --lines) followed by an integer.

head -n 21 warAndPeace.txt would print the first 21 lines of the warAndPeace.txt file

We can also use an even shorter syntax to specify a number of lines: head -3 filename.txt will print the first 3 lines of the file.

head contd contd

We can also provide a number of bytes to print out, rather than lines using the `-c` option.

`head -c 8 warAndPeace.txt` would print the first 8 bytes of the warAndPeace.txt file.

tail

The tail command works similarly to the head command, except it prints from the END of a file. By default, it prints the last 10 lines of a file.

`tail warAndPeace.txt` would print the last 10 lines of the warAndPeace.txt file

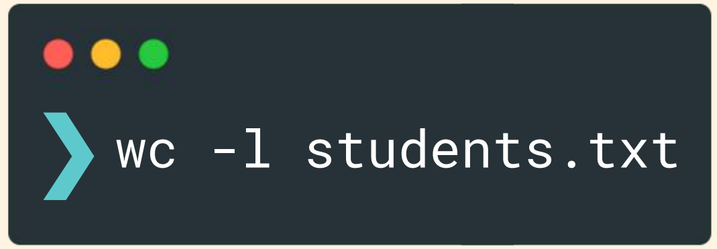
The same `-n` and `-c` options we saw with head also work with the tail command.

wc

The word count command can tell us the number of words, lines, or bytes in files. By default, it prints out three numbers: the lines, words, and bytes in a file.

We can use the -l option to limit the output to the number of lines.

The -w option limits the output to the number of words in the file.



sort

The sort command outputs the sorted contents of a file (it does not change the file itself). By default, it will sort the lines of a file alphabetically.

sort names.txt would print each line from names.txt, sorted in alphabetical order.

sort cont'd

The -r option tells the sort command to sort in reverse order.

sort names.txt -r would print each line from names.txt, sorted in REVERSE alphabetical order.

sorting numerically

The -n option tells the sort command to sort using numerical order.

sort -n prices.txt would print each line from names.txt, sorted in numerical order.
We could also reverse it with sort -nr prices.txt

uniques only

The `-u` option tells the sort command to ignore duplicates and instead only sort unique values

sorting by field

We can specify a particular "column" that we want to sort by, using the -k option followed by a field number.

In this example, sort data.txt -nk 2 tells sort to use the numeric sort and to sort using the 2nd field.

```
pencil 0.50
flowers 9.99
pie 4.99
soda 0.99
```

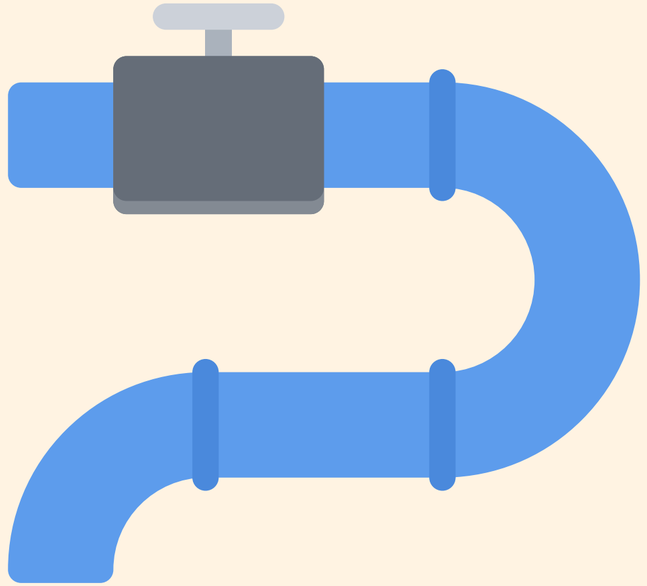
```
pencil 0.50
soda 0.99
pie 4.99
flowers 9.99
```

Redirection

Standard Streams

The three standard streams are **communication channels** between a computer program and its environment.

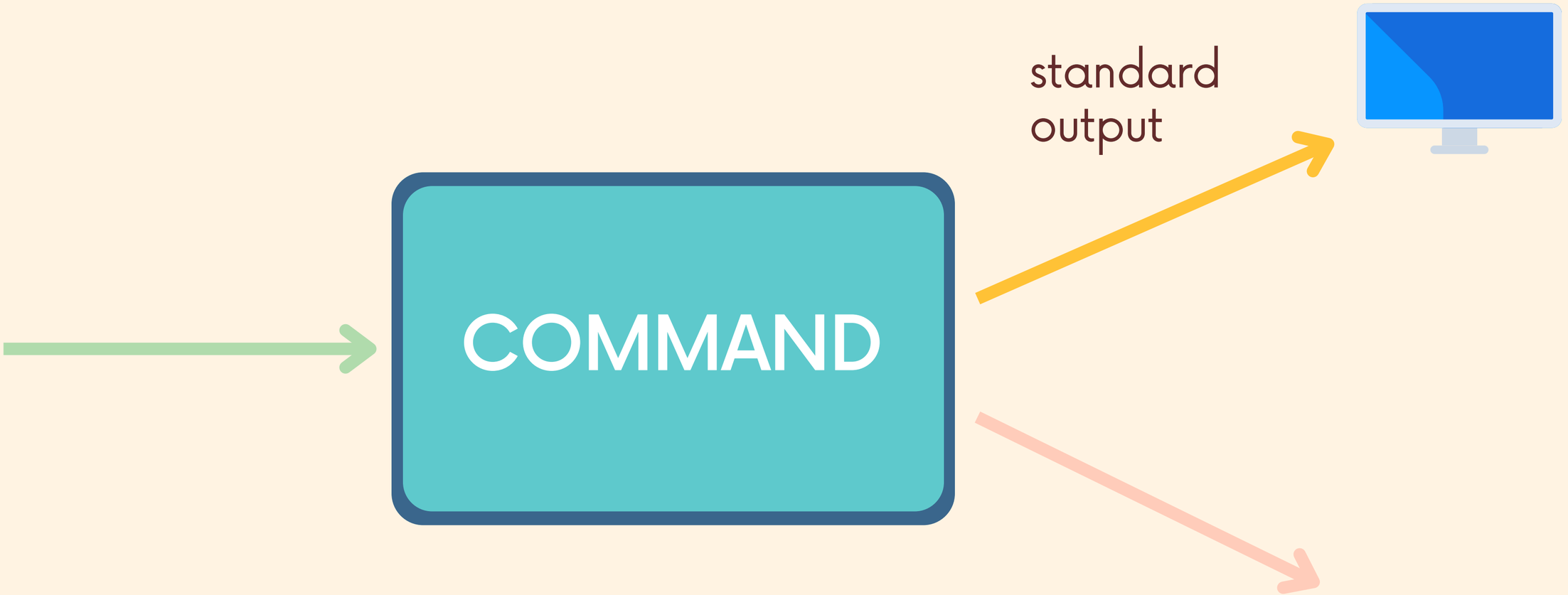
They are:
- Standard Input
- Standard Output
- Standard Error

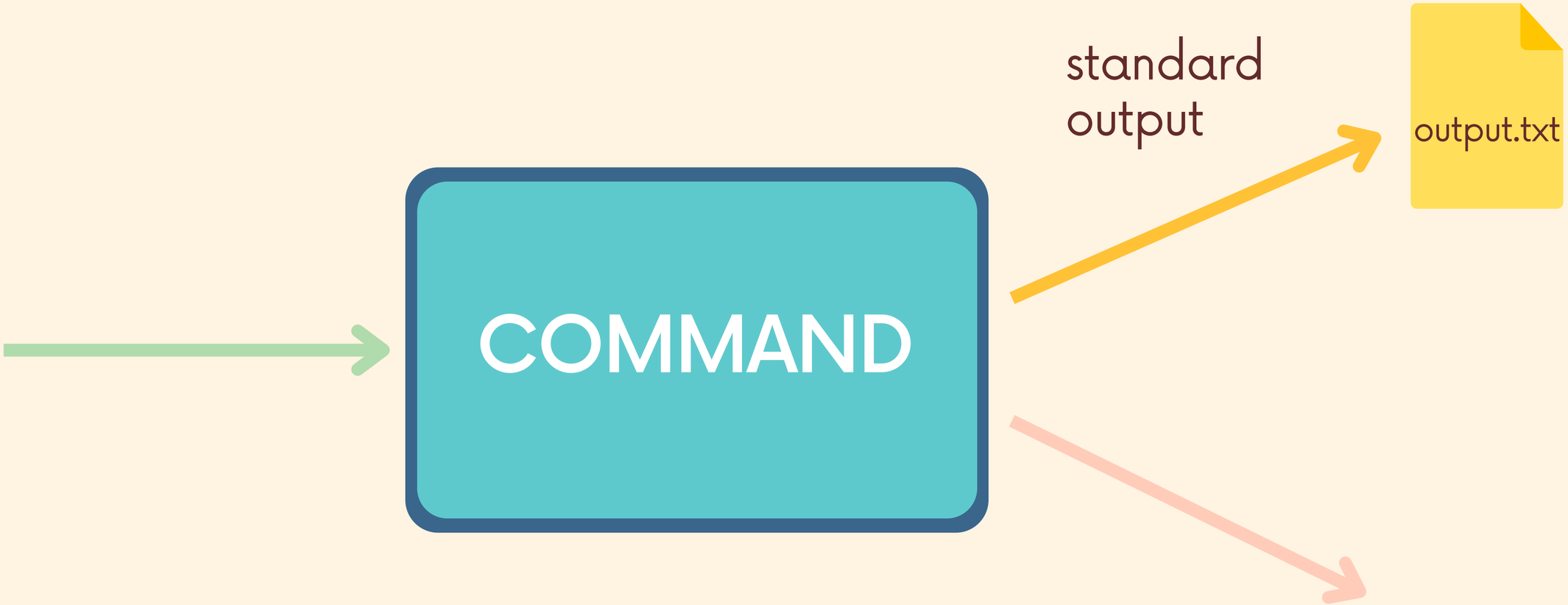


Standard Output

Standard output is a place to which a program or command can send information.

The information could go to a screen to be displayed, to a file, or even to a printer or other devices.


standard
output
COMMAND


standard
output
output.txt
COMMAND


standard
output
COMMAND

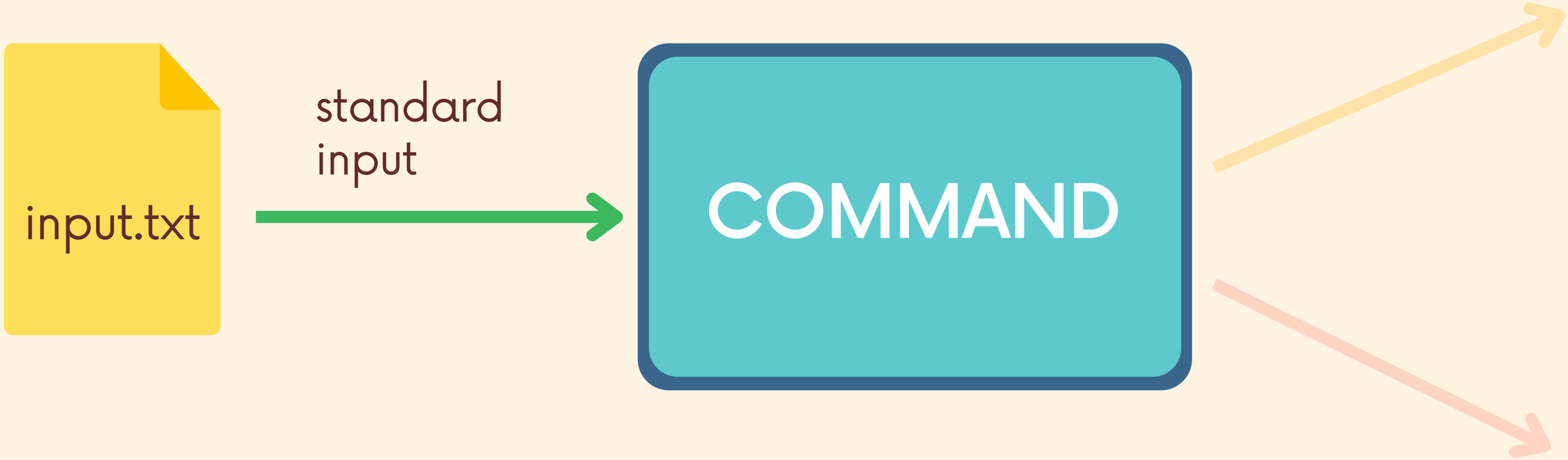
Standard Input

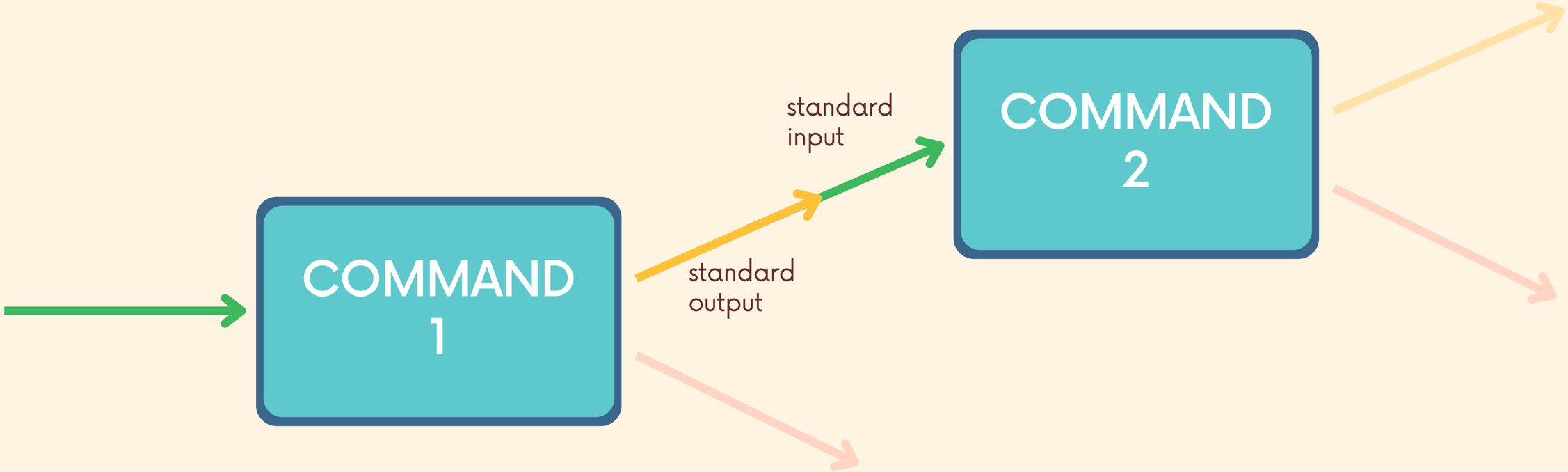
Standard input is where a program or command gets its input information from. By default, the shell directs standard input from the keyboard.

The input information could come from a keyboard, a file, or even from another command!

standard
input

COMMAND


input.txt
standard
input
COMMAND


COMMAND
1
standard
output
standard
input
COMMAND
2

Standard Error

Commands and programs also have a destination to send error messages: standard error.

By default, the shell directs standard error information to the screen for us to read, but we can change that destination if needed!

redirection

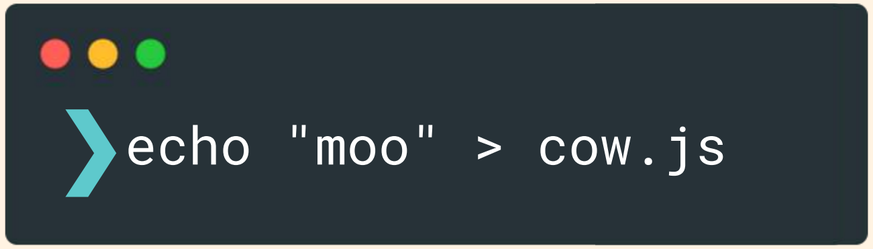
"redirection" describes the ways we can alter the source of standard input, and the destinations for standard output and standard error.

redirecting output

The redirect output symbol (>) tells the shell to redirect the output of a command to a specific file instead of the screen.

By default, the date command will print the current date to the screen. If we instead run date > output.txt the output will be redirected to a file called output.txt

redirecting output



This example redirects the output of echo

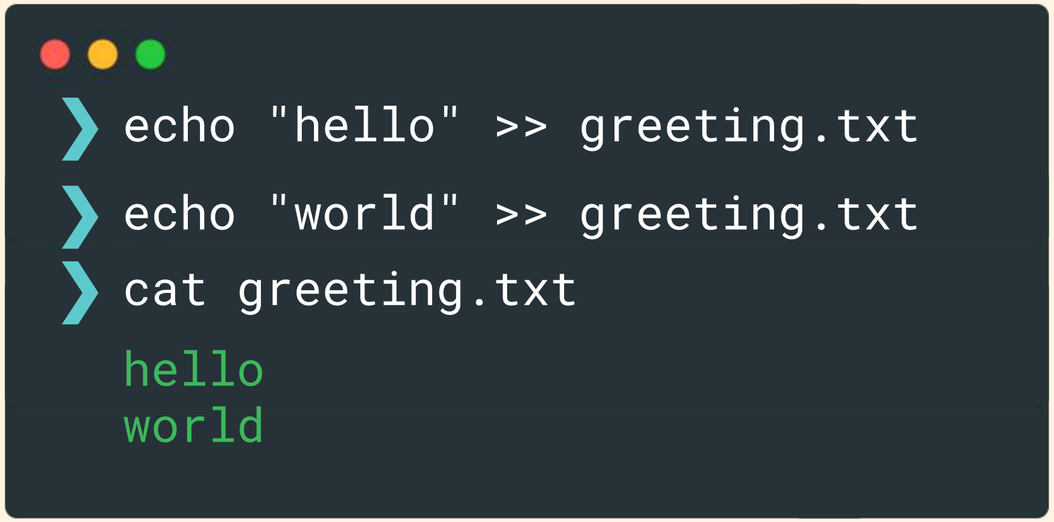
This example saves the output of ls -l to a file.

*Note the > symbol needs to occur after any options and arguments!

Appending

When we redirect output into a file using > any existing contents in the file are overwritten. Sometimes this is not what we want!

To instead keep the existing contents in the file and add new content to the end of the file, use >> when redirecting.

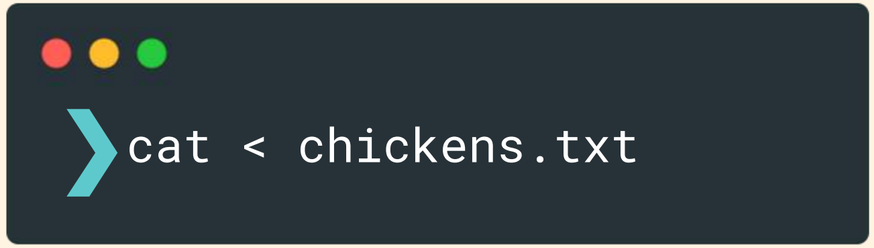


redirecting input

To pass the contents of a file to standard input, use the < symbol followed by the filename.

For example, we could pass the contents of the chickens.txt file to the cat command using
cat < chickens.txt

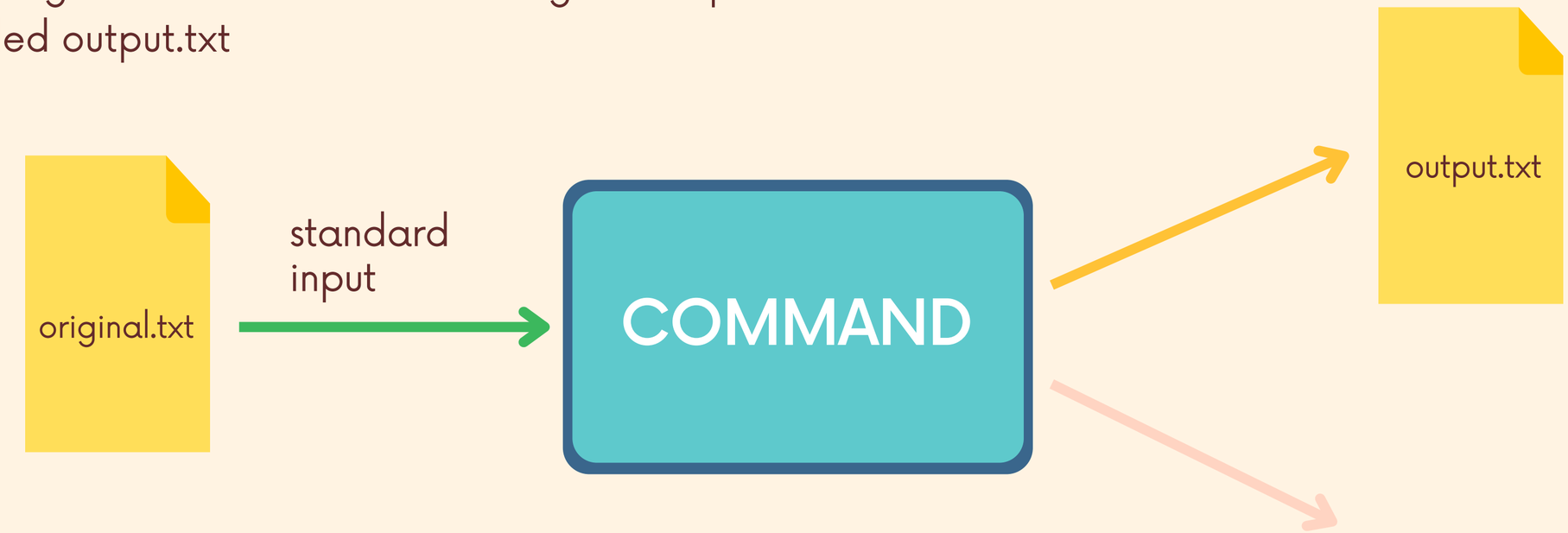
cat (and many other commands) are set up to accept filenames as arguments directly, but we can also redirect to standard input manually.



combo!

We can redirect standard input and output at the same time! In this example, we are using cat to read in the contents of original.txt and then redirecting the output to a file called output.txt

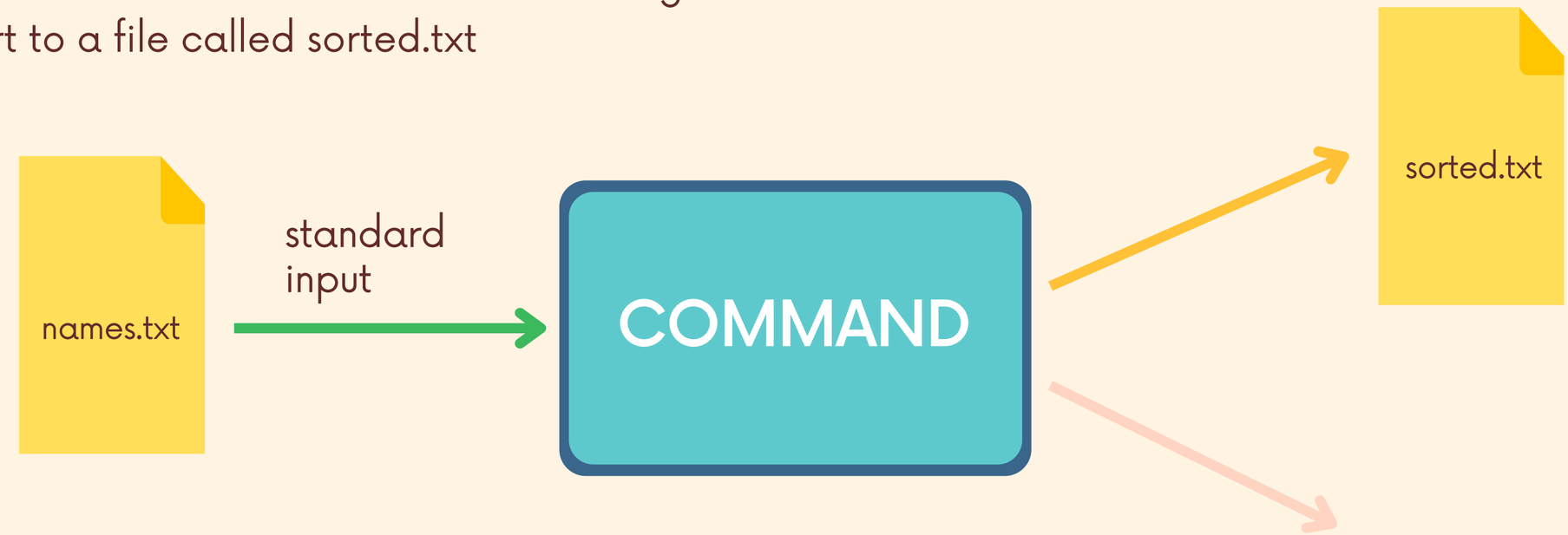
```
❯cat < original.txt > output.txt
```



combo! again!

We can redirect standard input and output at the same time! In this example, we are redirecting the names.txt file to the sort command. We are also redirecting the output of sort to a file called sorted.txt

```
❯sort < names.txt > sorted.txt
```



redirecting standard error

By default, error messages are output to the screen, but we can change this by redirecting standard error.

The standard error redirection operator is `2>`

If we ran a command like `cat nonexistentfile` (where the file really does not exist) we would see an error printed to the screen. We can instead redirect standard error to a file with `cat nonexistentfile 2> error.txt`

Why 2> ?

Each stream gets its own numeric file descriptor,
and for standard error the number is 2.

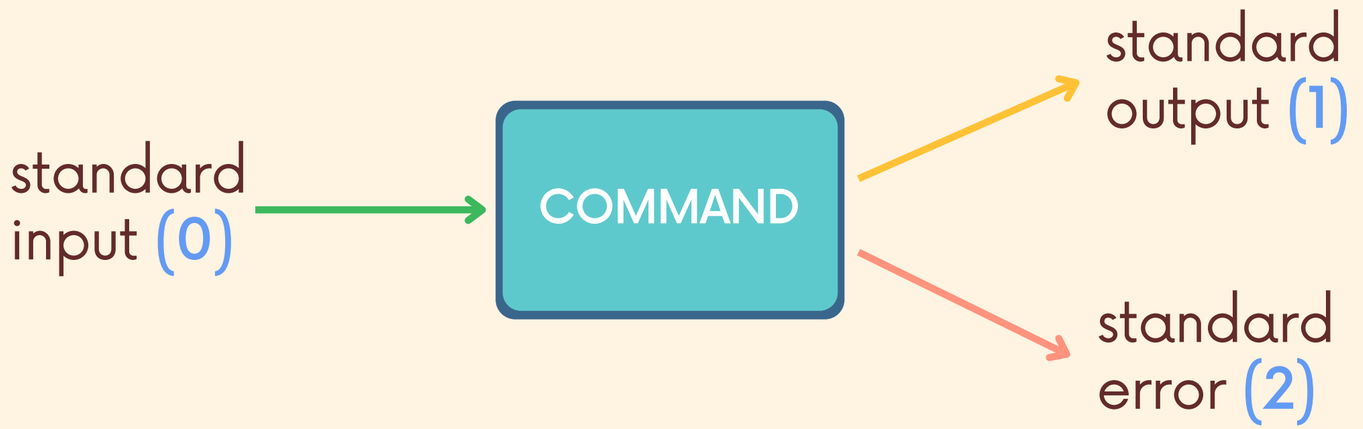
defaults

The > operator actually defaults to using 1 as the file descriptor number, which is why we didn't need to specify 1> to redirect standard output

Similarly, the < operator uses a default file descriptor number of 0, so we don't need to specify 0< to redirect to standard input (though we can!)

```
❯date 1> now.txt
```

```
❯date > now.txt
```

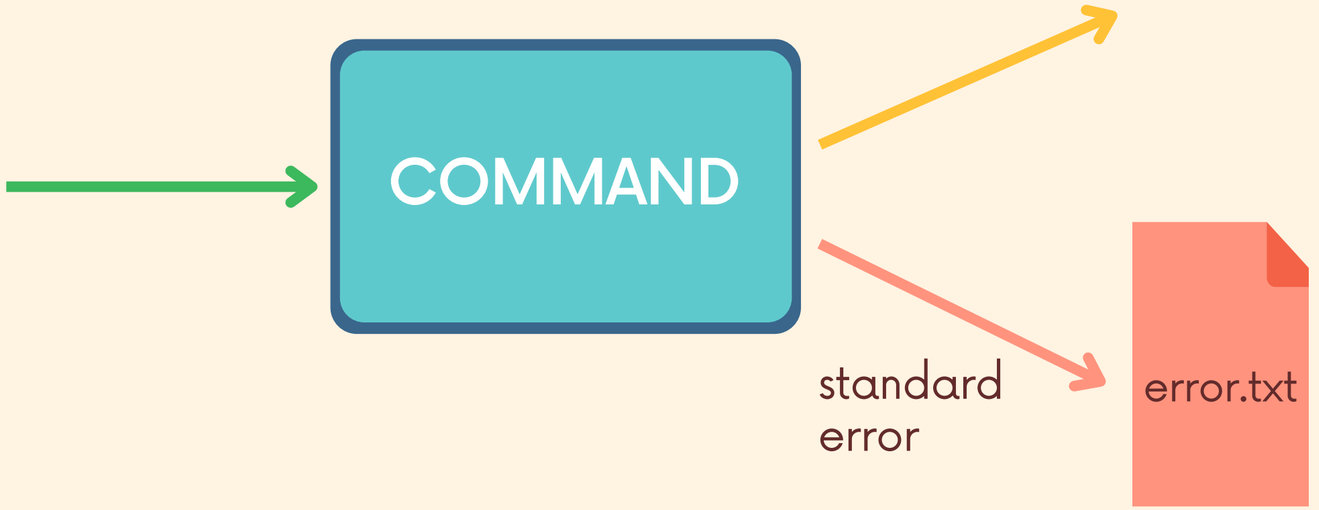


another one

In this example, I'm running **ls** with an invalid option that generates the error message "ls: illegal option -- z"

I'm redirecting standard error to an error.txt file

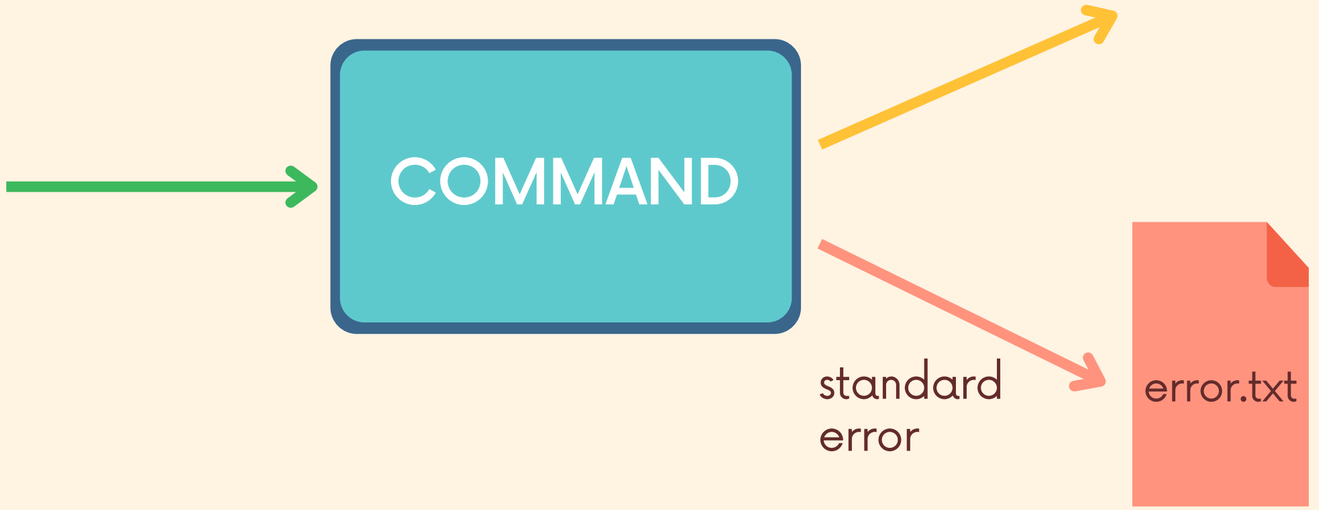
```
❯ ls -zzz 2> error.txt
```



appending

We can use the same >> syntax to append when redirecting standard error.

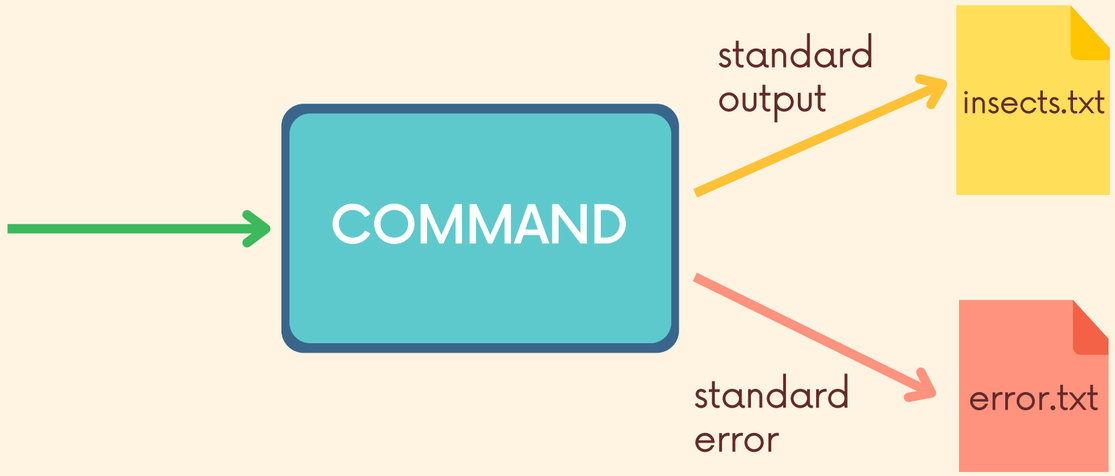
```
❯ ls -zzz 2>> error.txt
```



all together now

We can redirect multiple streams at once! In this example, we are concatenating two files, redirecting standard output to a file called insects.txt, and redirecting standard error to a file called error.txt

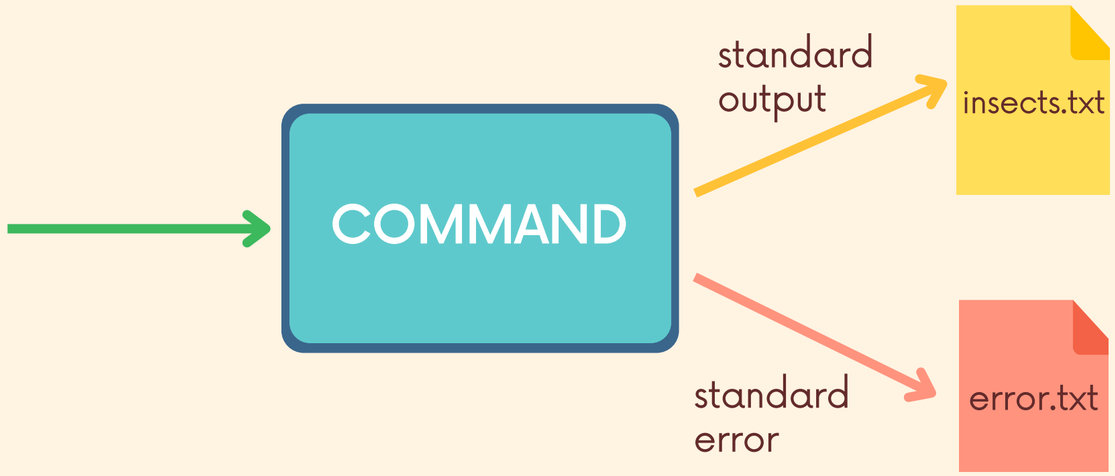
```
❯cat bees.txt ants.txt > insects.txt 2> error.txt
```



Order matters!

When redirecting both standard output and standard error, make sure standard output comes FIRST. Always redirect standard error after standard output.

```
❯cat bees.txt ants.txt > insects.txt 2> error.txt
```



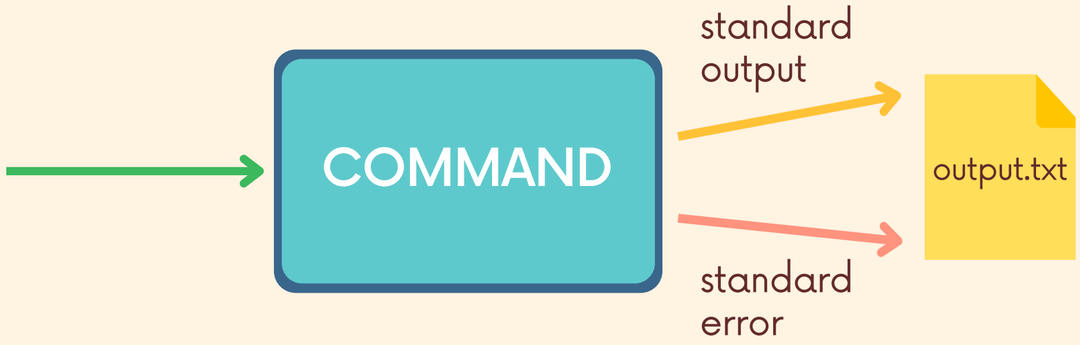
getting fancy

If we wanted to redirect both standard output and standard error to the same file, we could do this...

```
❯ ls docs > output.txt 2> output.txt
```

Or we could instead use `2>&1` which is a fancy syntax for saying "redirect standard error to the same location as standard output (or whatever has the file descriptor #1)

```
❯ ls docs > output.txt 2>&1
```

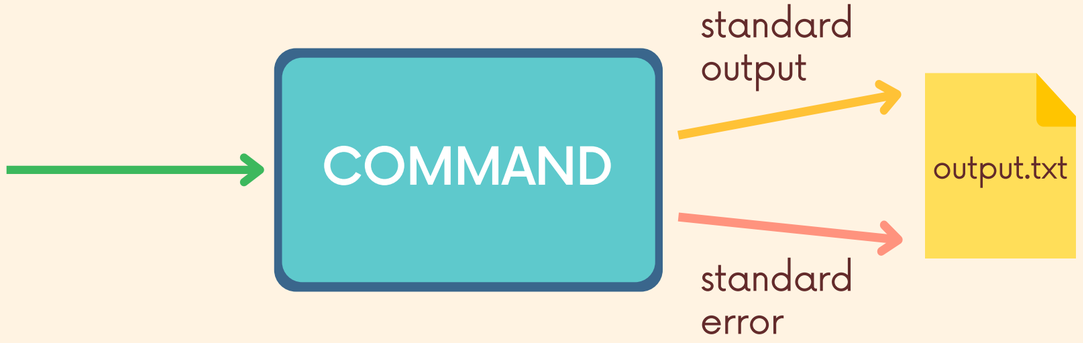


getting fancier

Newer versions of bash also support a fancier syntax for redirecting both standard output and standard error to the same file: the **&>** notation

```
❯ ls docs &> output.txt
```

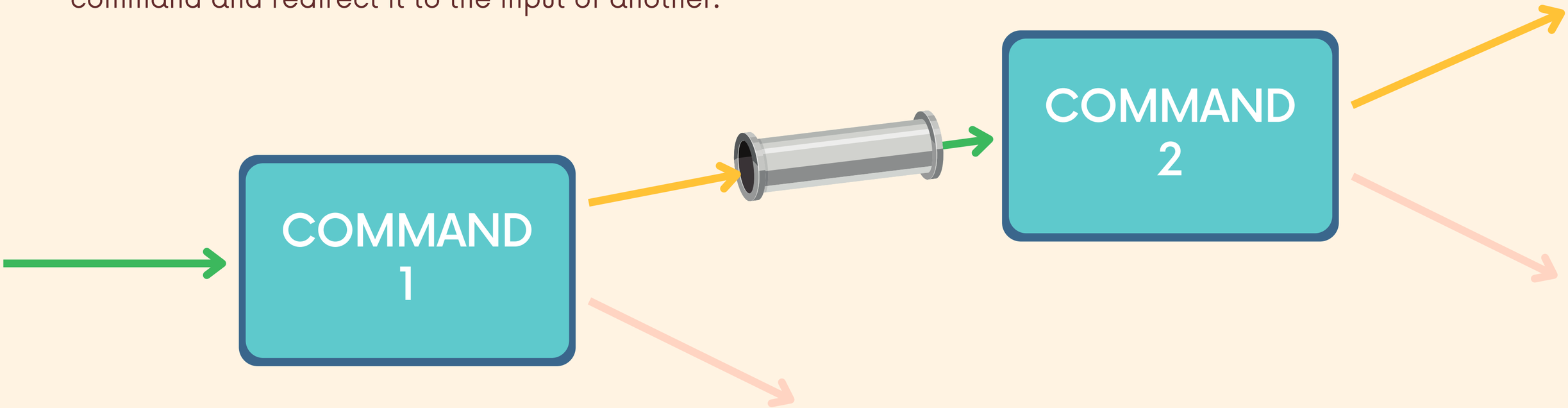
```
❯ls docs &>> output.txt
```



Piping

Pipes

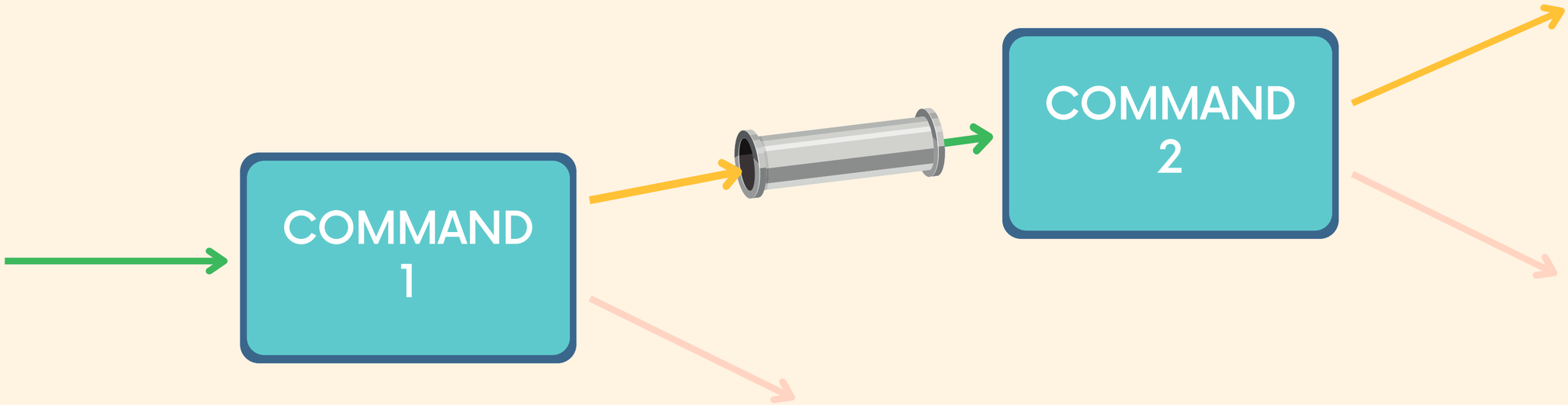
Pipes are used to redirect a stream from one program to another program. We can take the output of one command and redirect it to the input of another.



The Syntax

We use the pipe character (|) to separate two commands. The output of the first command will be passed to the standard input of the second command.

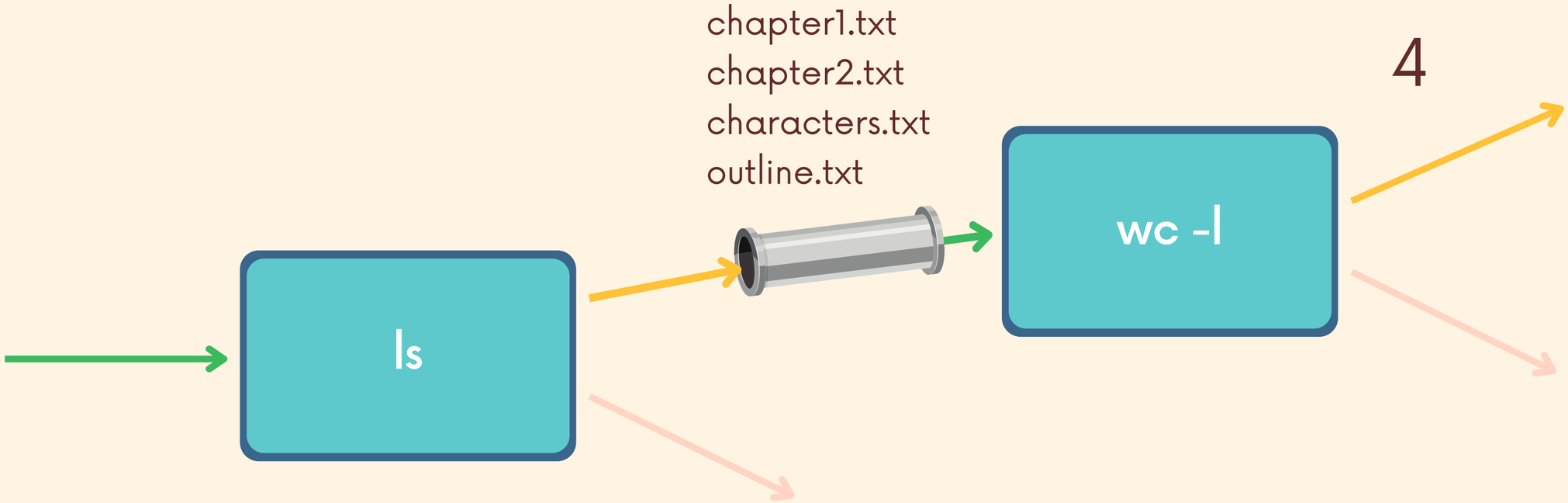
```
❯command1 | command2
```



ls | head

This example list the files (non hidden files) in a directory. We pipe the output of ls to the word count command. The -l option tells wc to count the number of lines.

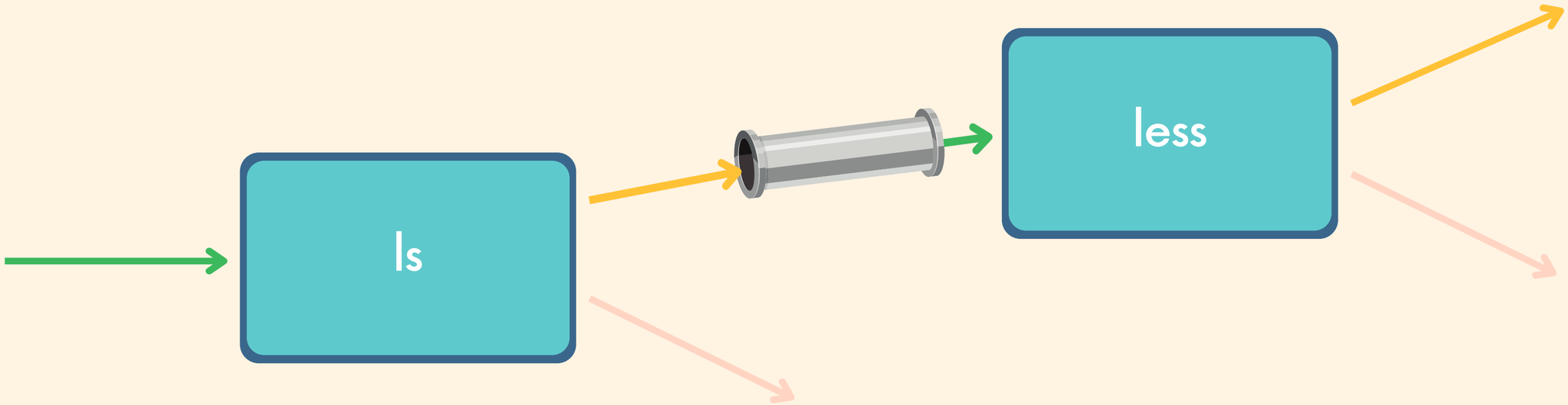
```
❯ ls | head -10
```



ls | less

This example pipes the output of **ls** to **less.** the /usr/bin directory typically contains a bunch of stuff, so it can be nice to use less to read the results in a more manageable way.

```
❯ ls -l /usr/bin | less
```



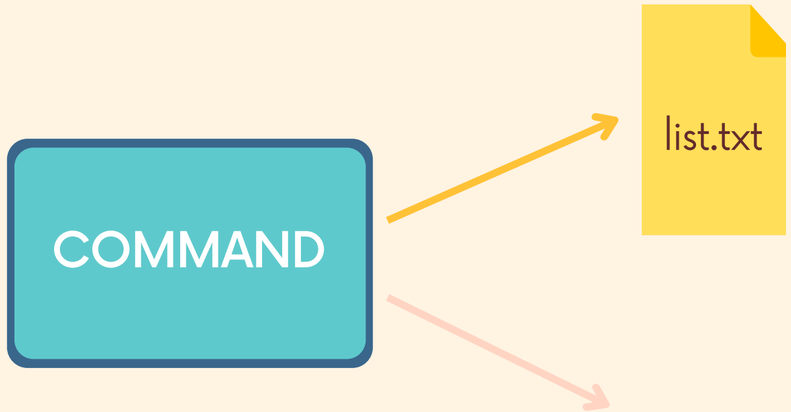
> vs |

Though both the > character and the | character are used to redirect output, they do it in very different ways.

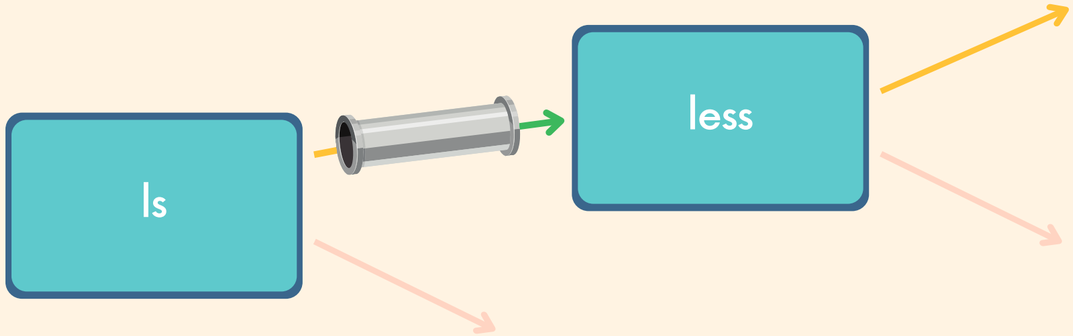
> connects a command to some file.

| connects a command to another command.

```
❯ls -l /usr/bin > list.txt
```



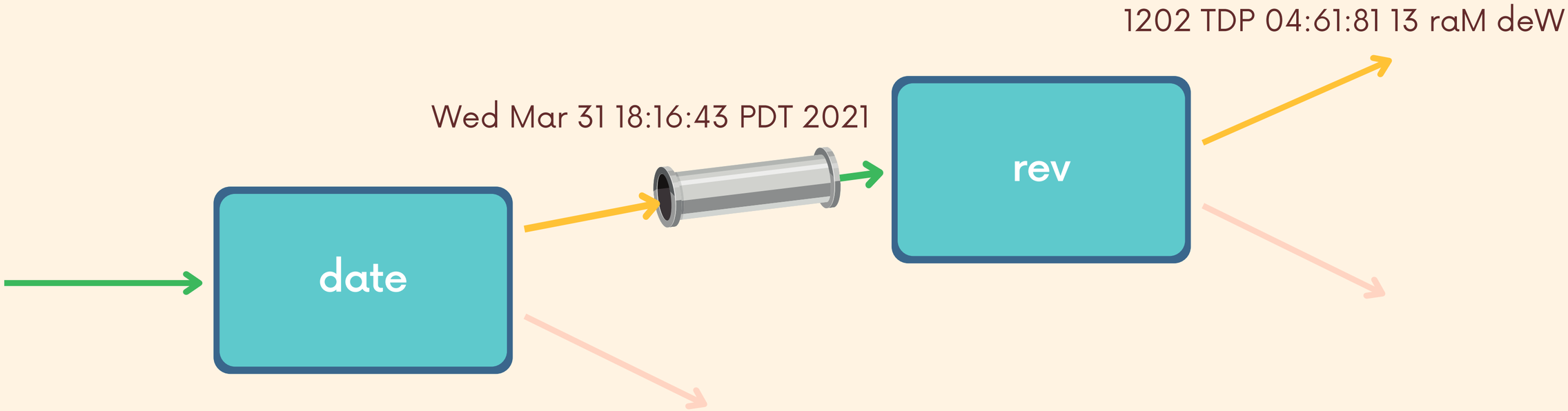

```
❯ls -l /usr/bin | less
```



date | rev

This example shows the output of the date command being piped to the rev command. The end result is the reverse of the current date! Very useful!

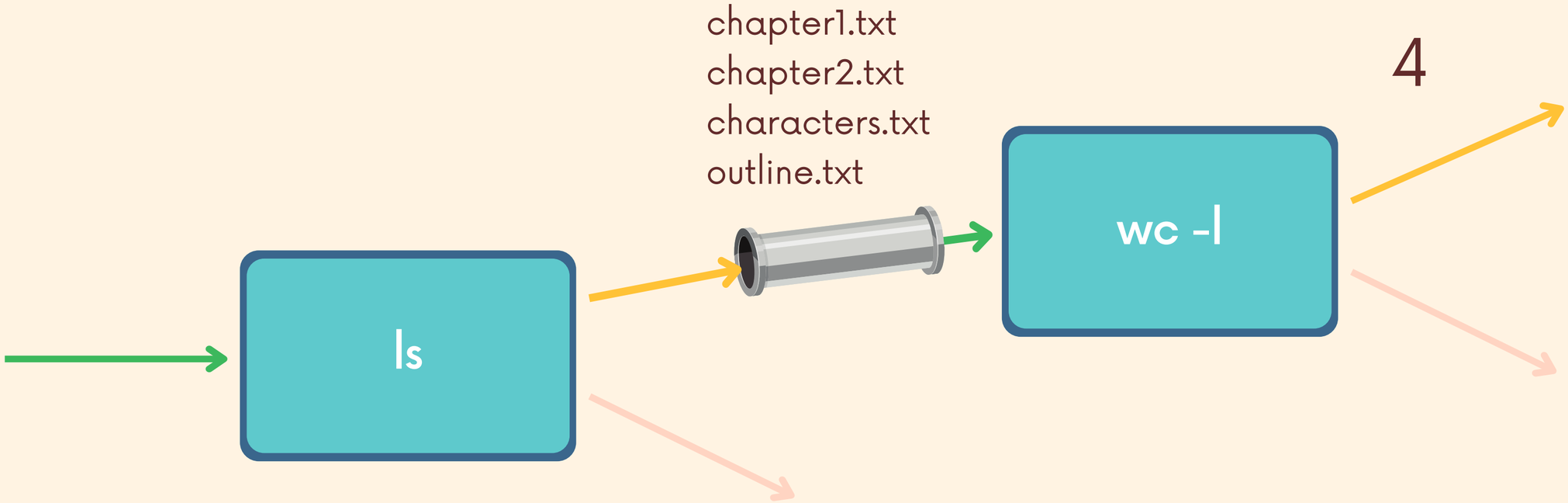
```
❯ date | rev
```



ls | wc

This example counts the number of files (non hidden files) in a directory. We pipe the output of ls to the word count command. The -l option tells wc to count the number of lines.

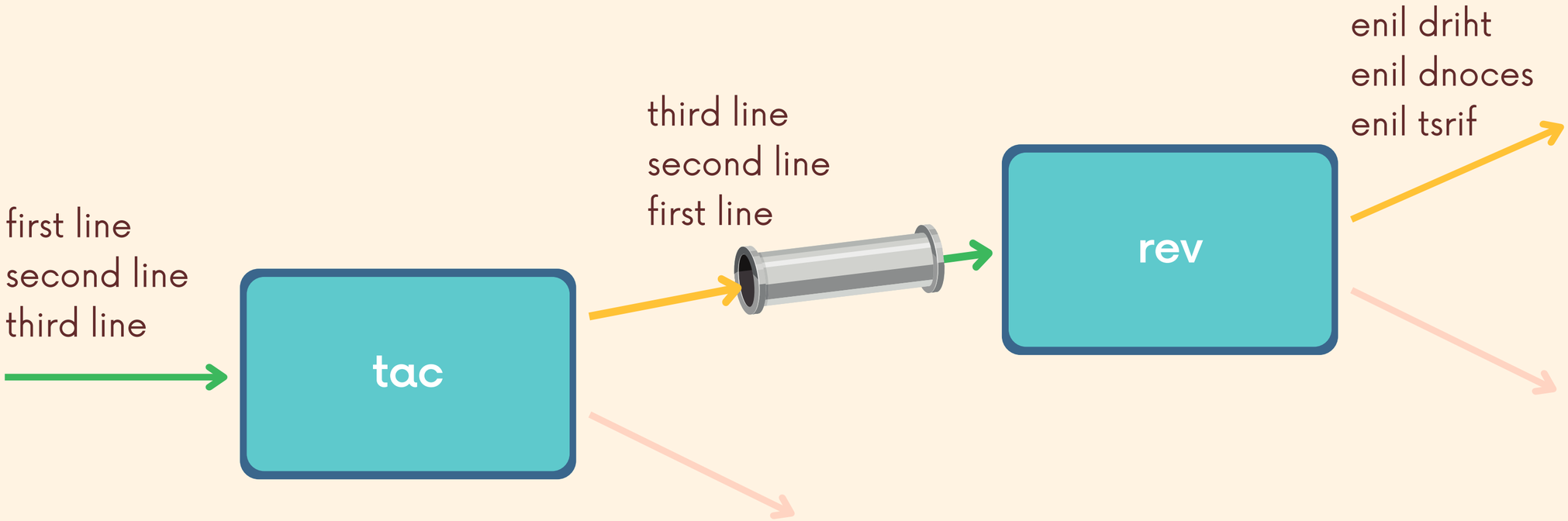
```
❯ ls | wc -l
```



tac | rev

In this example, we are calling tac with a file and then piping the output to rev. The final result is the content of file.txt printed "horizontally" and "vertically" reversed

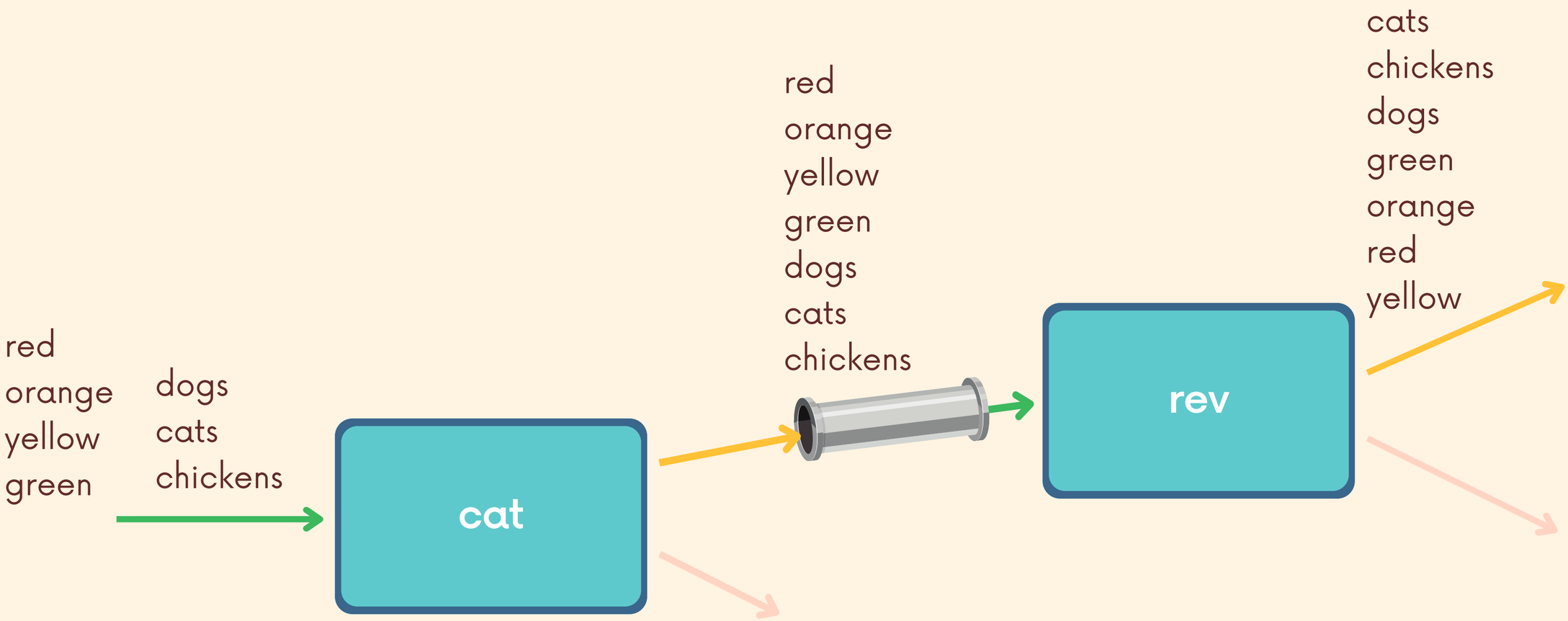
```
❯ tac file.txt | rev
```



cat | sort

This example concatenates two files using cat and then sorts them alphabetically.

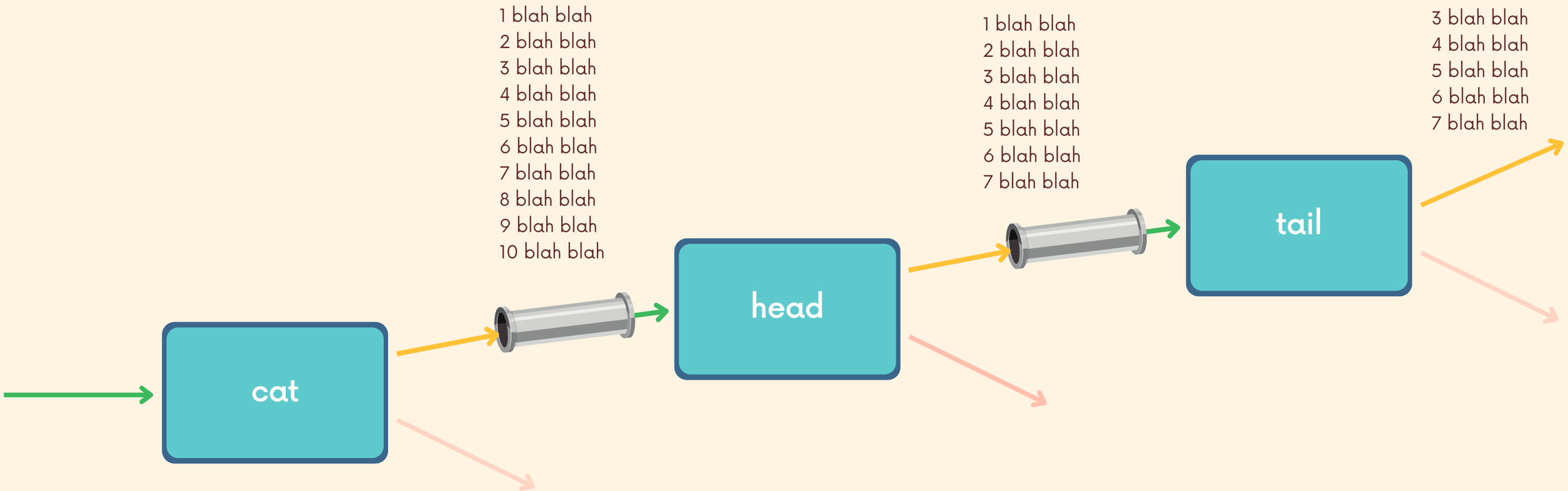
```
❯cat colors.txt pets.txt | sort
```



cat | head | tail

In this example, we are using cat to feed a file to head, which cuts it down to the first 7 lines of the file and passes it to tail, which then outputs the last 5 lines of that "chunk"
The end result is lines 3-7 are output to the screen

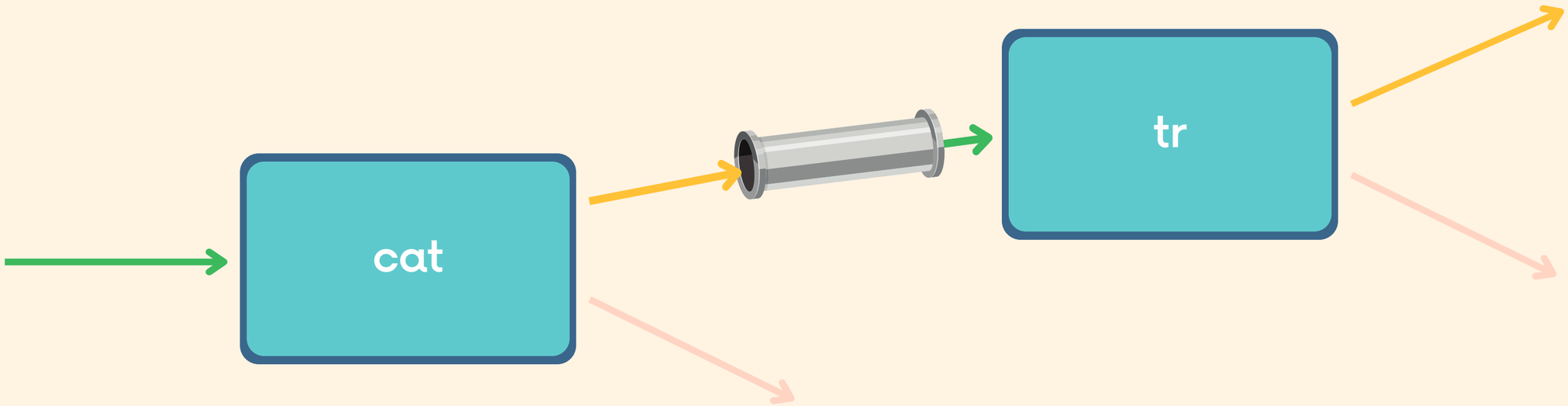
```
❯cat file | head -7 | tail -5
```



The Syntax

We use the pipe character (|) to separate two commands. The output of the first command will be passed to the standard input of the second command.

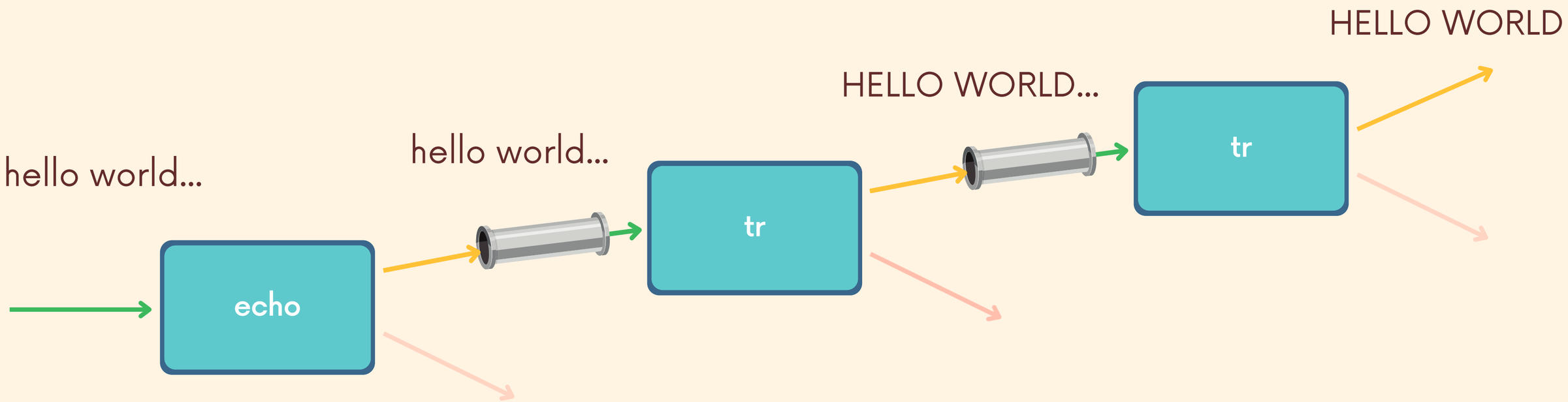
```
❯ cat somefile | tr s $
```



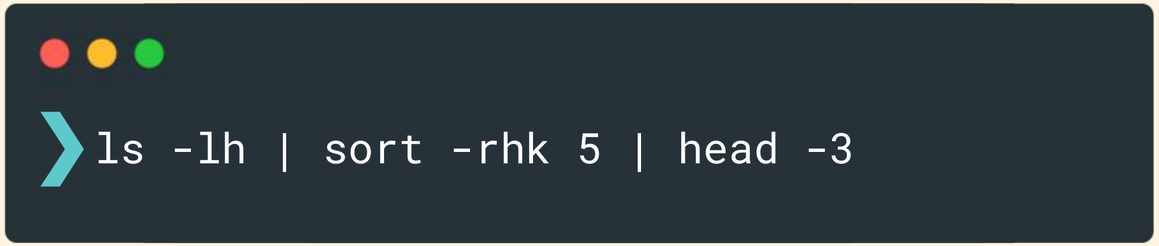
echo | tr | tr

This example uses **tr** to capitalize a string and then again uses **tr** to remove all punctuation from the capitalized string.

```
❯echo "hello world..." |  tr “[:lower:]” “[:upper:]” | tr -d "[:punct:]"
```



ls | sort | head



This example displays the 3 largest files in the current directory, using ls, sort, and head.

First, ls -lh lists out all the files in the current directory. That output is passed to sort, which sorts based on the fifth field (the file size). The -h option is for human readable sort (comparing 100b, 40k, 1g, etc), and the -r reverses the order so that we end up with the largest files first.

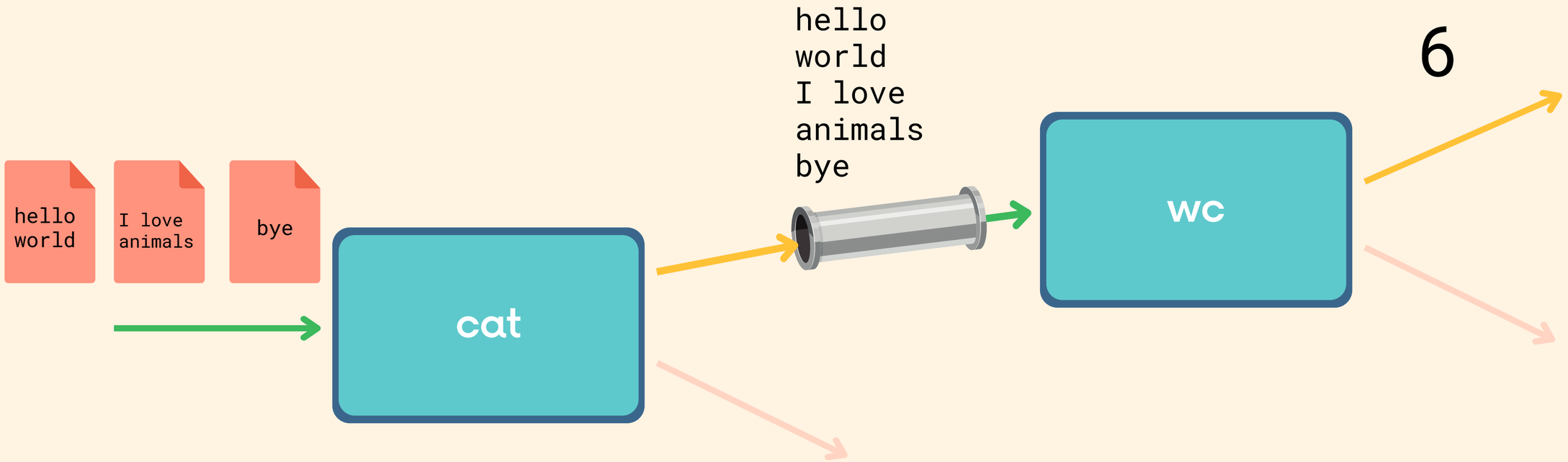
Finally, that output is passed to head, which limits the results to the first 3.

*NOTE: this is not the preferred way to find large files! Use the du command instead

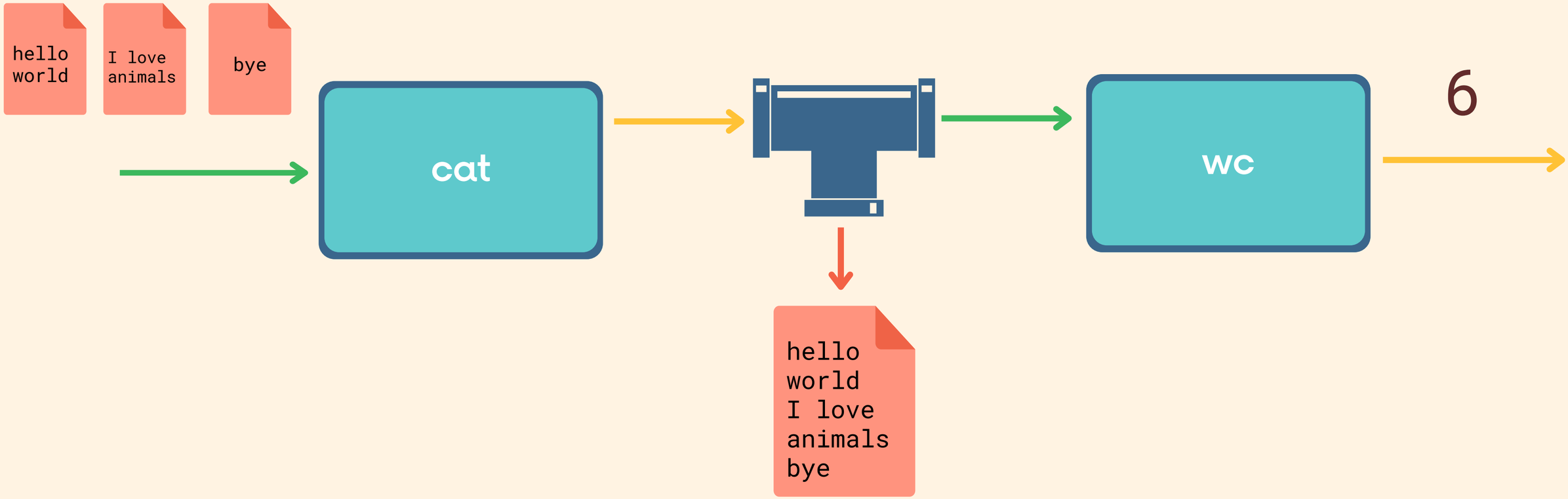
An Example

In this example, I'm using cat to concatenate three files together before piping the output to wc to get a count of the total number of words.

```
❯cat file1 file2 file3 | wc -w
```



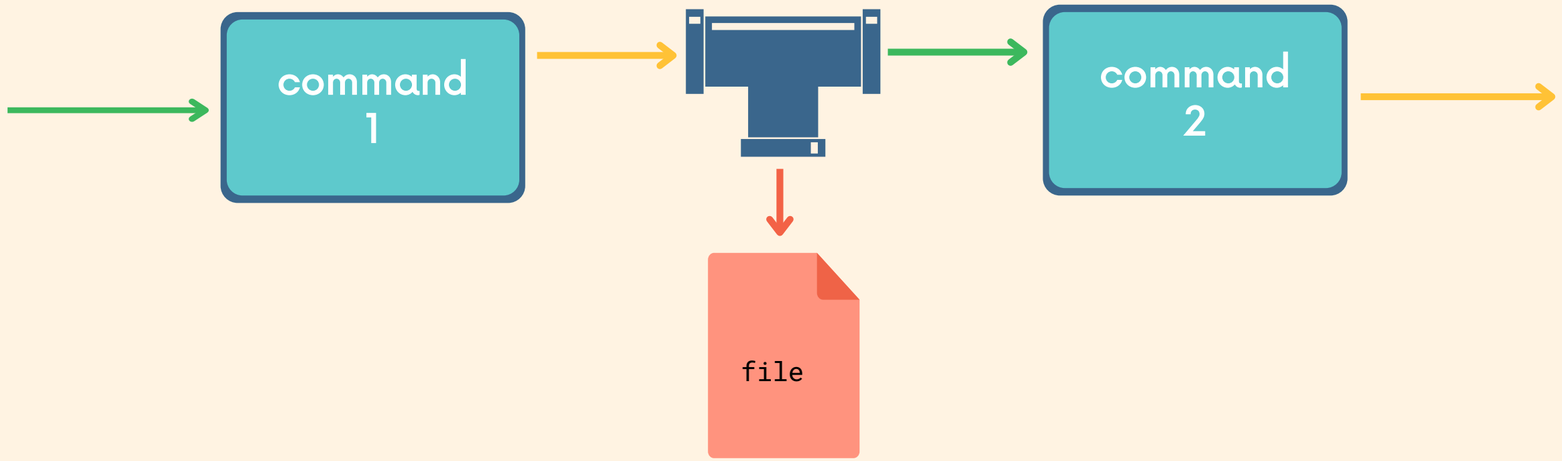
What if I wanted to create a file with the output of cat?



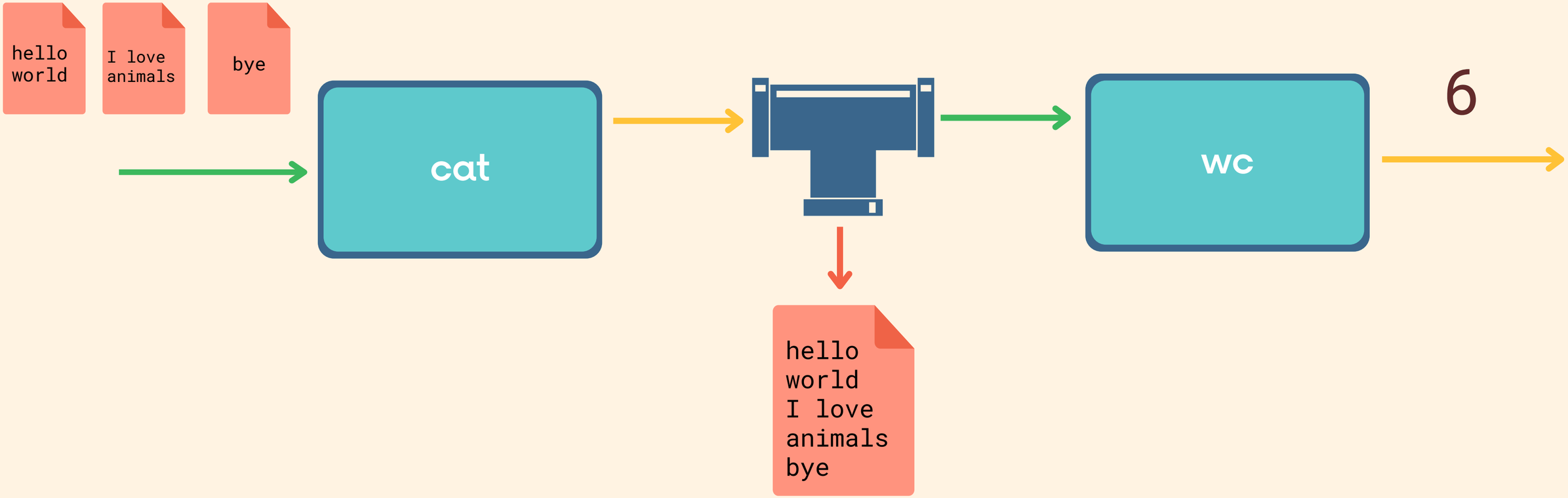
enter the tee command

The tee program reads standard input and copies it both to standard output AND to a file. This allows us to capture information part of the way through a pipeline, without interrupting the flow.

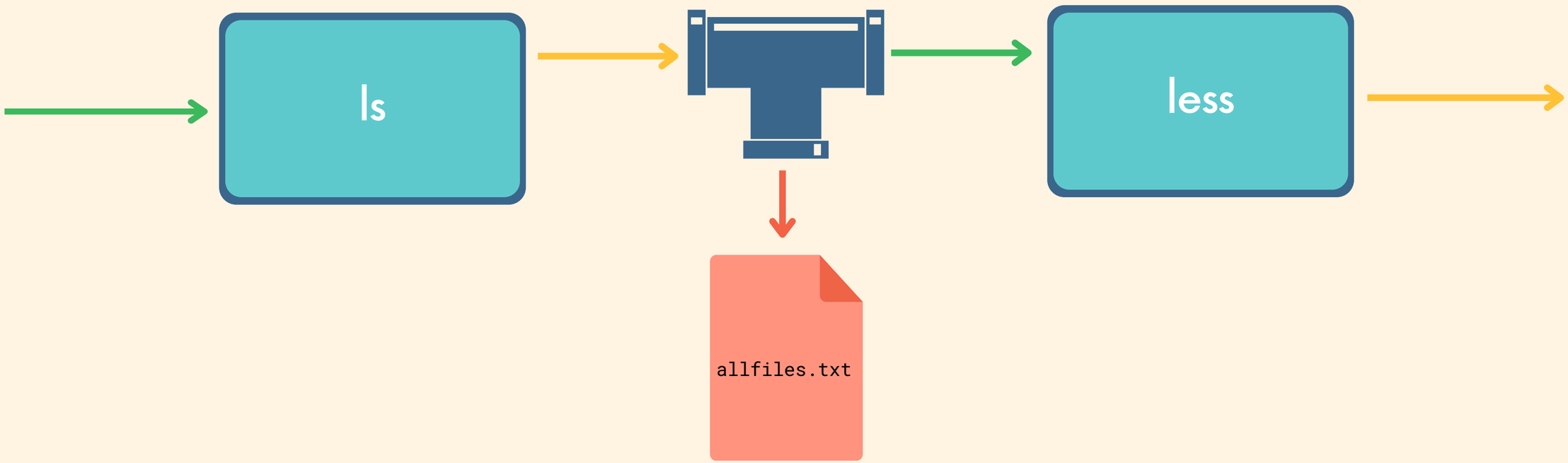
```
❯command1 | tee file.txt | command2
```



```
❯cat file1 file2 file3 | tee combo.txt | wc -w
```



```
❯ ls -l /usr/bin | tee allfiles.txt | less
```

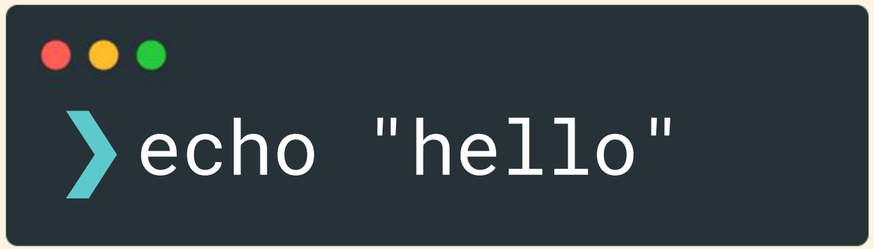


Expansion & Substitution

echo

The echo command is very simple. It simply displays text that we pass to it. We'll be using it to demonstrate some concepts in this section.

It's particularly useful in shell scripts (we cover them later), when we want to output something to the screen from within a file.



Wildcard Characters (aka globbing patterns)

We can use special wildcard characters to build patterns that can match multiple filenames at once.

The asterisk (*) character represents zero or more characters in a filename. For example...

- `ls *.html` will match any files that end with .html like index.html and navbar.html

- `cat blue*` will match any files that start with "blue" like "blue.html" or "bluesteel.js"

The ? Wildcard

The question mark (?) character represents any single character.

- ls app.?? will match any files named "app" that end with two character file extensions like "app.js" or "app.py" but NOT "app.css"

- ls pic?.png will match pic1.png, pic2.png, pic3.png, but also picA.png, or picx.png.

Range Wildcards

Inside of square brackets ([ ]) we can specify a range of characters to match.

- `ls pic[123].png` will only match pic1.png, pic2.png, and pic3.png

- `ls file[0-9]` will match file1, file2, file3, up to file9

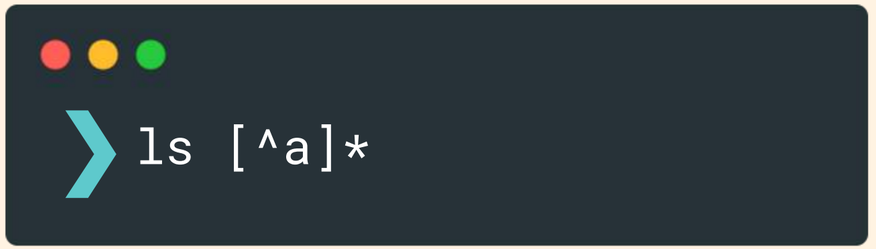
- `ls [A-F]*` will match any files that begin with a capital A, B, C, D, E, or F

Negating Ranges

Inside of square brackets ([ ]) we can specify a range of characters to NOT match, using a caret (^)

- ls [^a]* will match any files that do NOT start with "a"

- ls [^0-9]* will match any files that do NOT start with a numeric digit (0-9)



Character Classes

We can also use predefined named characters classes:

[:alpha:] - alphabetic characters, upper and lower
[:digit:] - digits 0-9
[:lower:] - lower case letters
[:upper:] - upper case letters
[:blank:] - blank characters: space and tab
[:punct:] - punctuation characters
[:alnum:] - alphanumeric characters (alpha + digit)

any file that starts with an uppercase letter

Brace Expansion

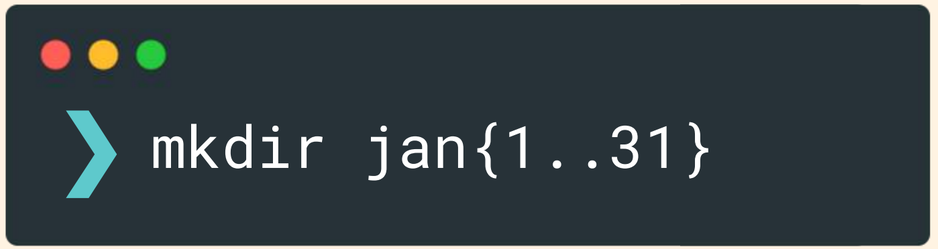
Brace expansion is used to generate arbitrary strings. Basically, it will generate multiple strings for us based on a pattern. We provide a set of strings inside of curly braces ({ }), as well as optional surrounding prefixes and suffixes.

The specified strings are used to generate all possible combinations with the optional prefixes and suffixes.

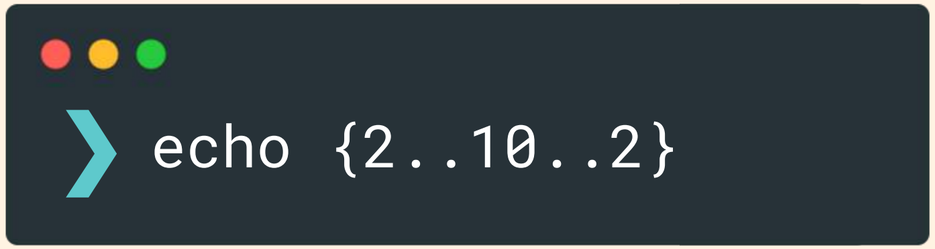
For example, touch page{1,2,3}.txt will generate three new files: page1.txt, page2.txt, and page3.txt

Ranges

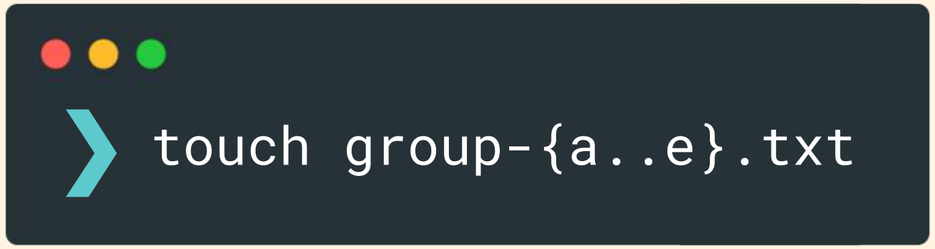
We can provide a numeric range, which will be used to generate a sequence. In this example, jan{1..31} will be expanded to jan1, jan2, jan3, etc. until jan31.



We can provide a third value which defines the interval for the range. In this example, echo {2..10..2} will print out the numbers 2, 4, 6, 8, and 10



We can even specify alphabetical ranges. This example generate the files group-a.txt, group-b.txt, group-c.txt,group-d.txt, and group-e.txt



Brace Expansion

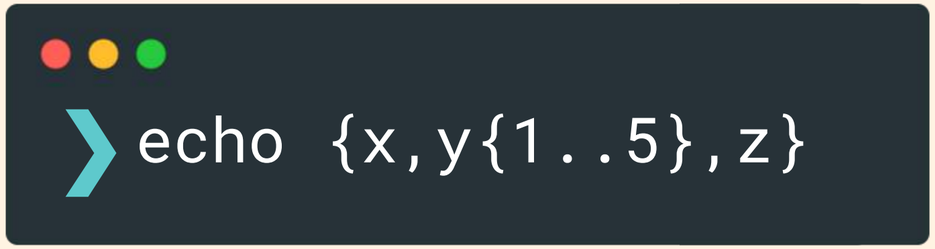
```
❯ echo {a,b,c}{1,2,3}
```

a1 a2 a3 b1 b2
b3 c1 c2 c3

```
❯ echo {b,r}{eef,at,ag}
```

beef bat bag
reef rat rag

Nested Brace Expansion



x y1 y2 y3 y4 y5 z

Arithmetic Expansion

The shell will perform arithmetic via expansion using the S((expression)) syntax. inside the parentheses, we can right artithmetic expressions using:

+ addition
- subtraction
* multiplication
/ division
** exponentiation
% modulo (remainder operator)

```
❯ echo $((10+7))
```

17

```
❯ echo $((3*13))
```

39

```
  echo $((10/3))
```

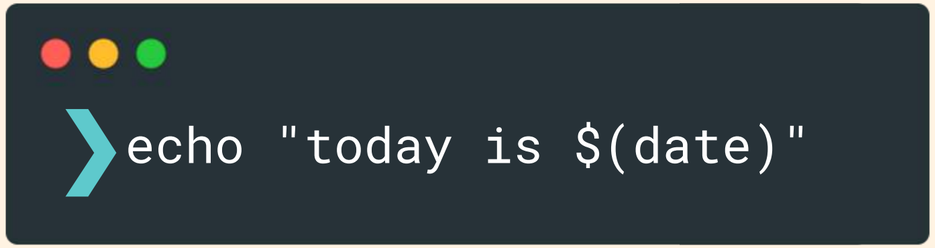
3

The shell only performs integer arithmetic, so the result is always a whole number

Command Substitution

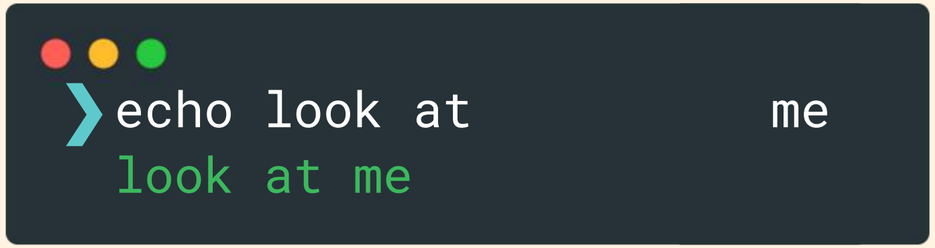
We can use the $(command) syntax to display the output of another command.

For example, echo "today is $(date)" will print "today is Thu 01 May 2021 03:10:31 PM PDT"



Quoting

In this example, our large whitespace is ignored because the shell performs something called word splitting.



In this example, we only see "holy" printed out because the shell thinks we are referencing a variable called hit. It can't find a value for hit, so it substitutes an empty string instead.

Double Quotes

If we wrap text in double quotes, the shell will respect our spacing and will ignore special characters except for dollar sign ($) , backslash (\ ), and backtick (`)

```
❯ echo "look at          me"
look at          me
```

Pathname expansion, brace expansion, and word splitting will be ignored. However, command substitution and arithmetic expansion is still performed because dollar signs still have meaning inside double quotes.

```
❯ echo "{1..9"}
{1..9}
```

Single Quotes

Use single quotes to suppress all forms of substitution.

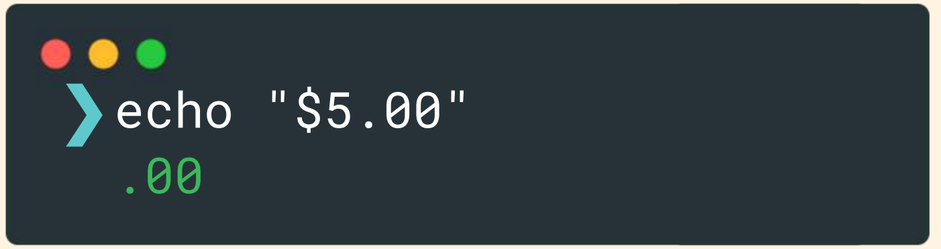
```
❯ echo "$((2+2)) is 4"
  4 is 4
```

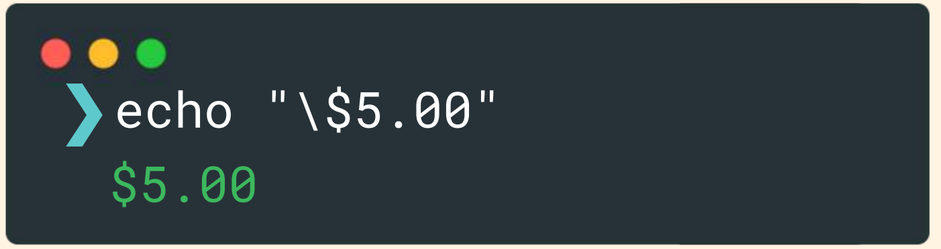
```
❯ echo '$((2+2)) is 4'
  $((2+2)) is 4
```

Escaping

To selectively prevent expansion or substitution for specific characters, we can prefix them with a single backslash.

We can use this to reference special characters that normally have meanings inside of filenames.





Finding Things

locate

The **locate** command performs a search of pathnames across our machine that match a given substring and then prints out any matching names.

It is nice and speedy because it uses a pre-generated database file rather than searching the entire machine.

For example, **locate chick** will perform a search for all files that contain chick in their name.

locate options

The **-e** option will only print entries that actually exist at the time locate is run.

The **-i** option tells locate to ignore casing

The **-l** or **--limit** option will limit the number of entries that locate retrieves.

find

The locate command is nice and easy, but it can only do so much! The **find** command is far more powerful! Unlike locate, find does not use a database file.

By default, **find** on its own will list every single file and directory nested in our current working directory.

We can also provide a specific folder. **find friends/** would print all the files and directories inside the friends directory (including nested folders)

finding by type

We can tell find to only find by file type: only print files, directories, symbolic links, etc using the -type option.

find -type f will limit the search to files

find -type d will limit the search to directories

finding by name

We can provide a specific pattern for **find** to use when matching filenames and directories with the **-name** option. We need to enclose our pattern in quotes.

To find all files on our Desktop that end in the .txt extension, we could run find ~/Desktop -name "*.txt"

Use the -iname option for a case insensitive search

Counting Results

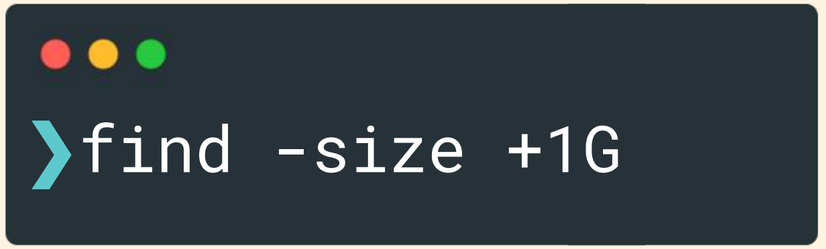
We can pipe the output of find to other commands like word count. Use the -l option to count the number of lines (each result from find is its own line)

finding by size

We can use the -size option to find files of a specific size. For example, to find all files larger than 1 gigabyte we could run **find -size +1G**

To find all files under 50 megabytes, we could run find -size -50M

To find all files that are exactly 20 kilobytes, we could run **find -size 20k**



finding by owner

We can use the -user option to match files and directories that belong to a particular user.

Timestamps

mtime, or modification time, is when a file was last modified AKA when its contents last changed.

ctime, or change time, is when a file was last changed. This occurs anytime mtime changes but also when we rename a file, move it, or alter permissions.

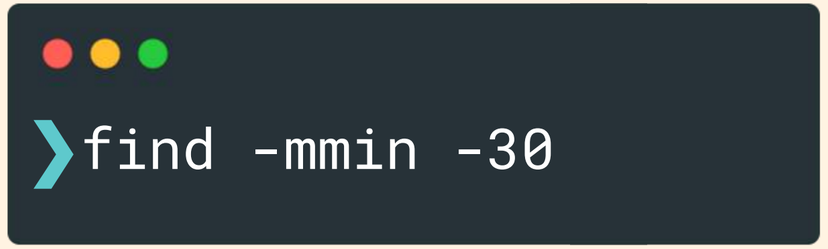
atime, or access time, is updated when a file is read by an application or a command like cat.

Finding By Time

We can use the **-mtime num** option to match files/folders that were last modified num*24 hours ago.

find -mmin -20 matches items that were modified LESS than 20 minutes ago.

find -mmin +60 matches items that were modified more than 60 minutes ago



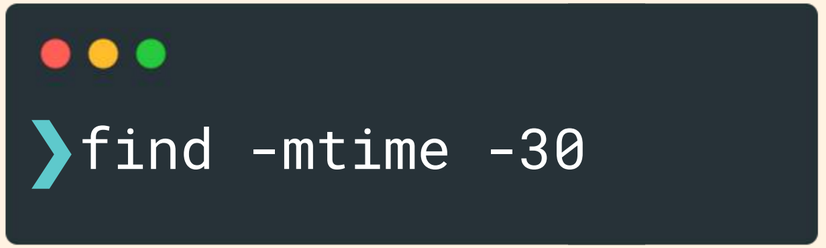
Finding By Time

-amin n will find files that were last accessed n minutes ago. We can specify +n for "greater than n minutes ago" and -n for "less than n minutes ago"

-anewer file will find files that have been accessed more recently that the provided file.

find -cmin -20 matches items that were modified LESS than 20 minutes ago.

find -cmin +60 matches items that were modified more than 60 minutes ago



Logical Operators

We can also use the **-and**, **-or**, and **-not** operators to create more complex queries.

```
❯find -name "*chick*" -or -name "*kitty*"
```

```
❯find -type -f -not -name "*.html"
```

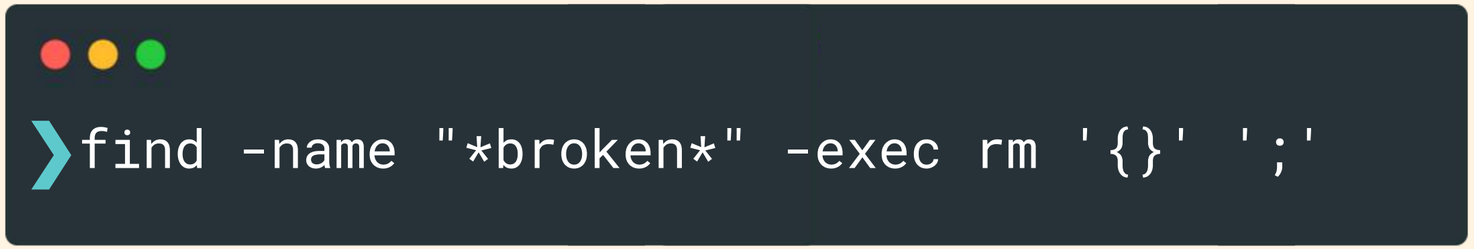
User- Defined Actions

We can provide **find** with our own action to perform using each matching pathname.

The syntax is **find -exec command {} ;**

The {} are a placeholder for the current pathname (each match), and the semicolon is required to indicate the end of the command.

User- Defined Actions



To delete every file that starts with contains "broken" in its file name, we could run:

```
find -name "*broken*" -exec rm '{}' ';'
```

Note that we need to wrap the {} and ; in quotes because those characters have special meanings otherwise

User- Defined Actions

```
❯find -type f -user colt -exec ls -l '{}' ';'
```

The above example finds all files that are owned by the user "colt", and then it lists out the full details for each match using ls -l

`find -type f -user colt -exec ls -l '{}' ';'`

Note that we need to wrap the {} and ; in quotes because those characters have special meanings otherwise

User- Defined Actions

```
❯find -type f -name "*.html" -exec cp '{}' '{}_COPY' ';'
```

The above example finds all files that end with .html. It then creates a copy of each one using the cp command. Each of the copies ends with "_COPY" so we end up with files like "index.html_COPY" and "navbar.html_COPY"

```
find -type f -name "*.html" -exec cp '{}' '{}_COPY' ';'
```

Note that we need to wrap the {} and ; in quotes because those characters have special meanings otherwise

xargs

When we provide a command via **-exec**, that command is executed separately for every single element. We can instead use a special command called **xargs** to build up the input into a bundle that will be provided as an argument list to the next command.

```
❯find -name "*.txt" -exec ls '{}' ';'
```

```
❯find -name "*.txt" | xargs ls
```

xargs

This example finds four individual chapter files (chapter1, chapter2, chapter3, and chapter4) and then passes them to the cat command, which then outputs the combined contents to a file called fullbook.txt.

```
❯find -name "chapter[1-4].txt" | xargs cat > fullbook.txt
```

xargs

xargs reads items from standard input, separated by blanks (spaces or newlines) and then executes a command using those items

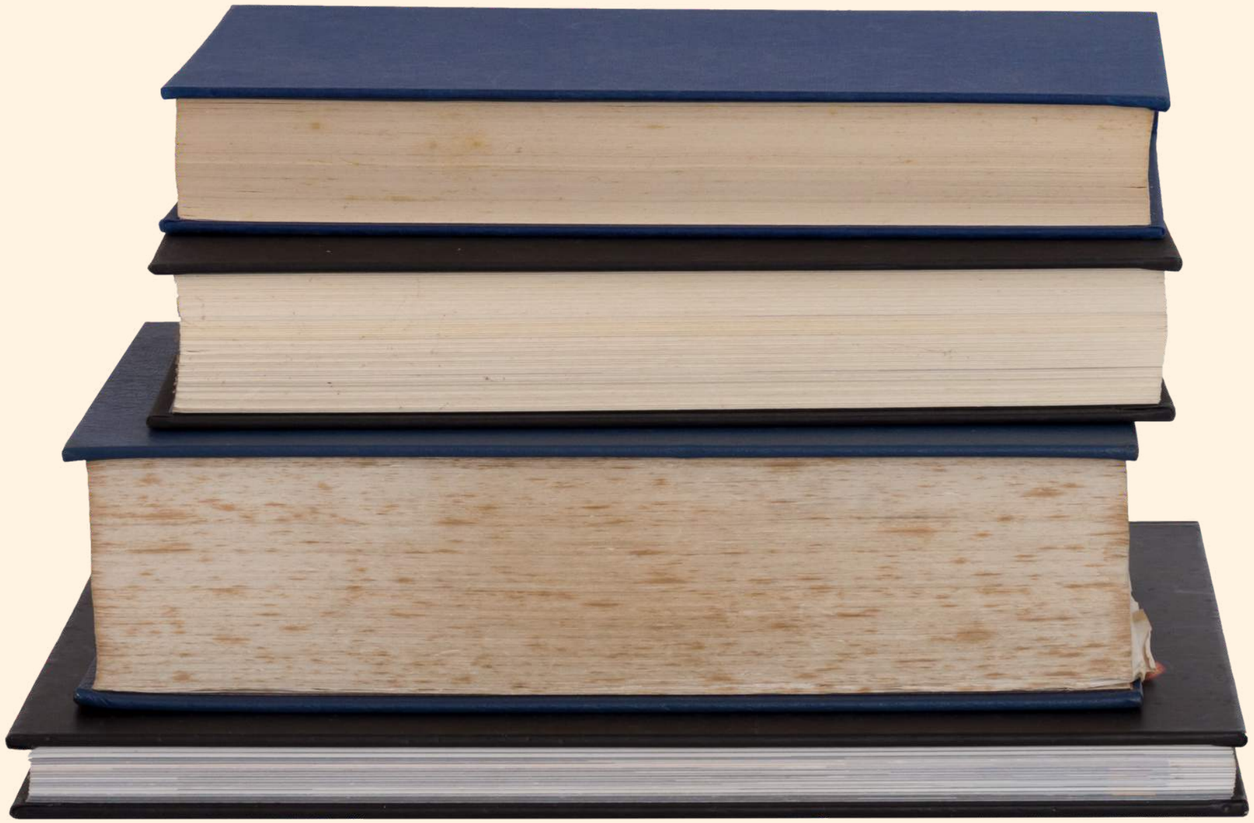
The mkdir command expects us to pass arguments. It doesn't work with standard input, so this example does NOT make any folders for us:

```
❯ echo "hello" "world" | mkdir
mkdir: missing operand
```

We can instead add in the xargs command, which will accept the standard input coming from echo and pass them as arguments to mkdir.

```
❯ echo "hello" "world" | xargs mkdir
❯ ls
hello  world
```

Grep



Grep

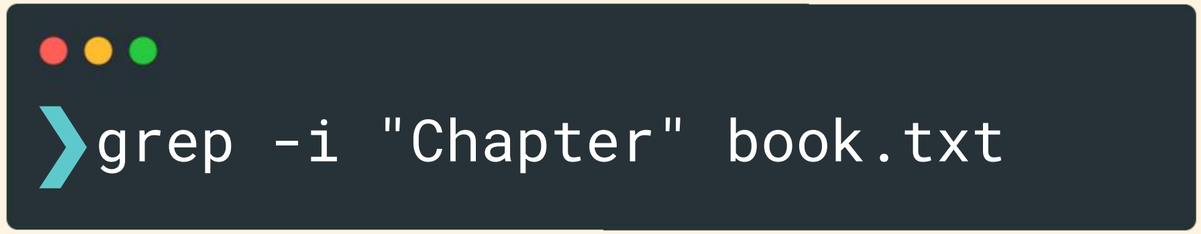
The **grep** command searches for patterns in each file's contents. Grep will print each line that matches a pattern we provide.

For example, grep "chicken" animals.txt will print each line from the animals.txt file that contains the pattern "chicken"

Case Insensitive

Use the **-i** option with grep to make the search case insensitive.

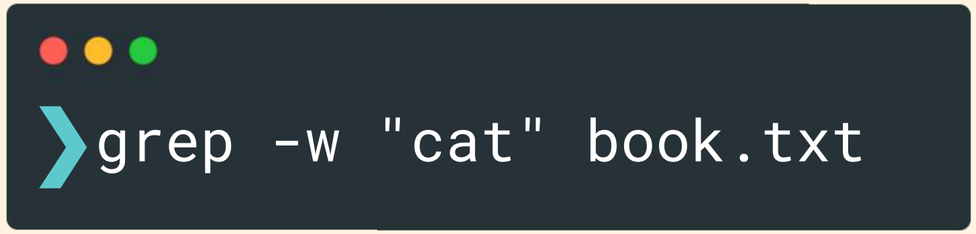
`grep -i "Chapter" book.txt` will print all matching lines from the book.txt file that contain the word "chapter" in any casing.



Word Search

Use the **-w** option to ensure that grep only matches words, rather than fragments located inside of other words. A word is defined by non-word characters on either side (start of line, spaces, end of line, punctuation, etc.)

`grep -w "cat" book.txt` would match cat but not catheter!



Recursive Search

Use the **-r** option to perform a recursive search which will include all files under a directory, subdirectories and their files, and so on.

If we don't specify a starting directory, grep will search the current working directory.

`grep -r "chicken"` will search the current working directory and any nested directories for lines that contain "chicken"

Regex Crash Course

We can provide regular expressions to **grep.** Regular expressions helps us match complex patterns, BUT the syntax does differ from what we've seen so far.

. - matches any single character
^ - matches the start of a line
$ - matches the end of a line
[abc] - matches any character in the set
[^abc] - matches any char NOT in set
[A-Z] - matches characters in a range
* - repeat previous expression 0 or more times
\ - escape meta-characters

Grep

This example matches a string that contains a digit 1-9 (not 0), followed by any 4 characters.

```
❯grep '[1-9]....' prices.txt
$95.99
$30.75
```

Grep -c

The **-c** option tells grep to print the number of matches instead of printing the actual matches

```
$95.99
$30.75
$9.99
$0.50
$2.50
$0.99
$0.75
```

```
❯grep -c "\$[1-9]" prices.txt
4
```

Grep -o

The -o option tells grep to only print out the matches, rather than the entire line containing each match.

```
❯grep -o "\$[1-9]" prices.txt
$9
$3
$9
$2
```

```
$95.99
$30.75
$9.99
$0.50
$2.50
$0.99
$0.75
```

Piping To Grep

A common use case is to use **grep** to whittle down or filter a large chunk of data.

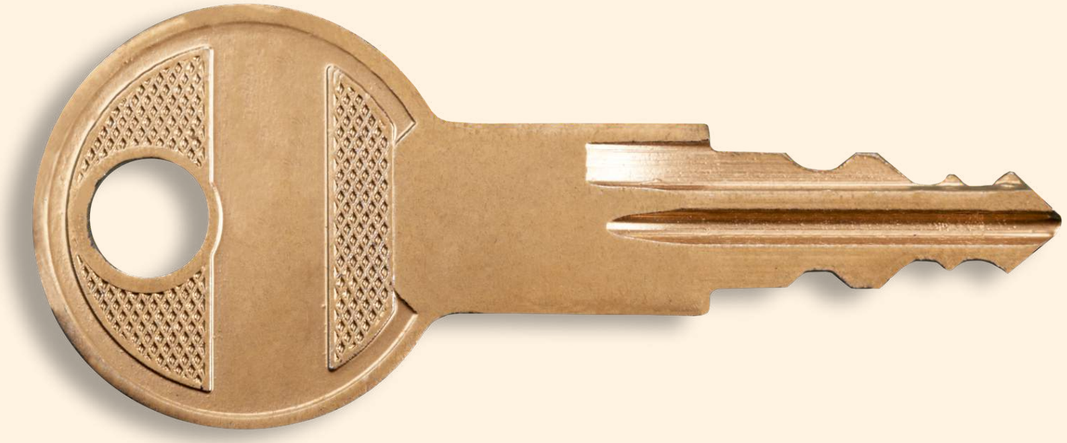
In this example, the **ps -aux** command will output a huge list of all processes running on our machine. We pipe that data to grep, and then filter it down to only the processes that include "hermione"

In effect, this command lets us see what hermione is up to!

Piping To Grep

In this example, we are getting the man page for grep and then piping that to the actual grep command, where we search for the string "count". Basically, it's a weird way of searching the man pages.

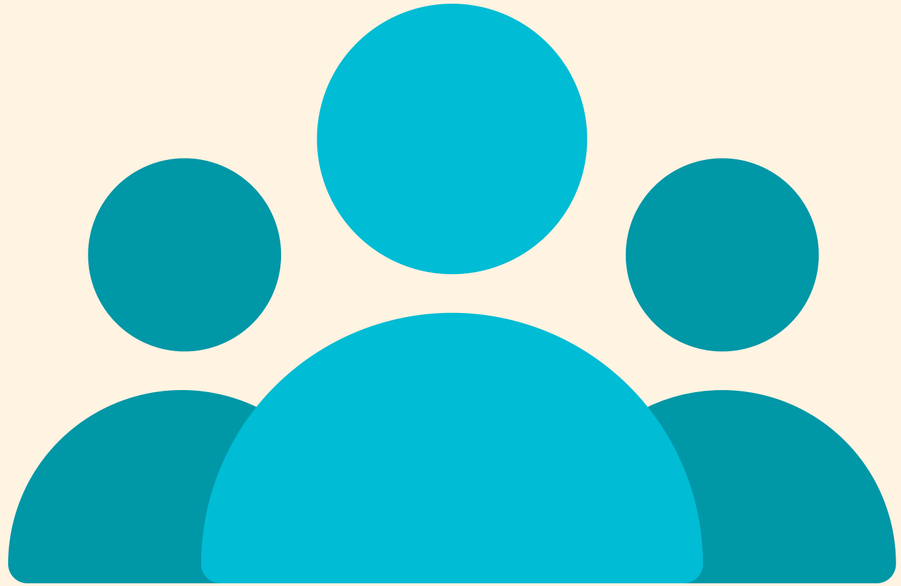
Understanding Permissions



Multiple Users

Unix and unix-like systems are **multiuser** operating systems. More than one person can be using the same computer at the same time (though this is tough logistically with only one display and keyboard!)

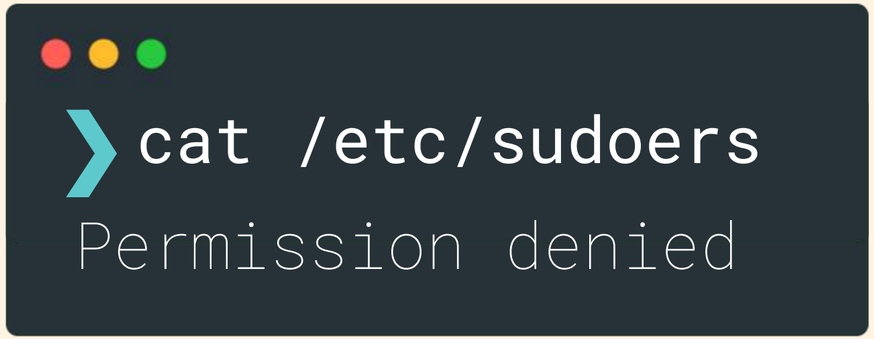
Way back when, computers were crazy expensive, massive machines. A university might only have one computer, but dozens of terminals sprinkled across campus.



Permission Denied?!

As regular users, we do not have permission to write or even read every file on the machine.

For example, if I try to read the file /etc/sudoers using cat /etc/sudoers I get a "permission denied" message.



Groups

On unix systems, a single user may be the owner of files and directories, meaning that they have control over their access.

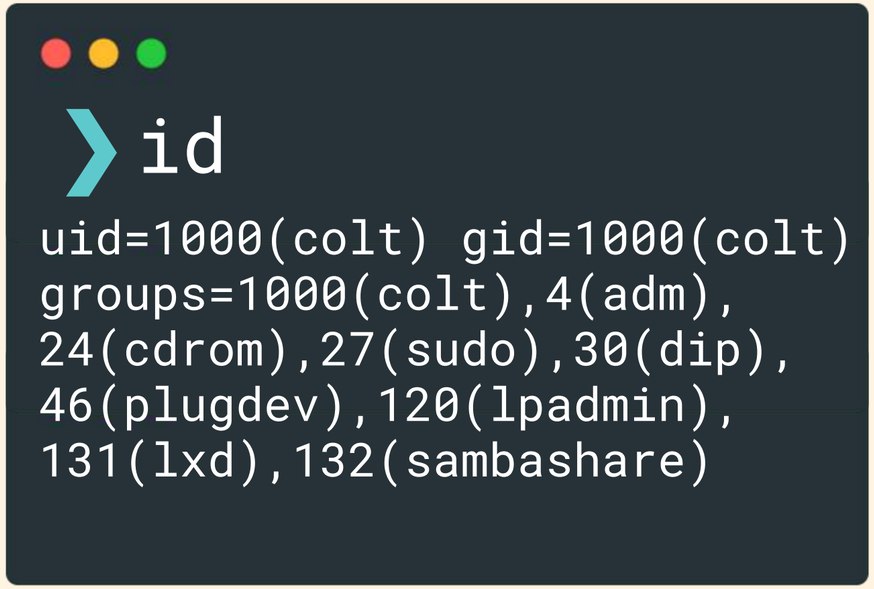
Additionally, users can belong to groups which are given access to particular files and folders by their owners.

User & Group IDs

When a new user account is made, it is assigned a user ID. The user is also assigned a group ID.

We can use the **id** command to view user and group ids.

These user ids are stored in /etc/passwd, and the group ids are in /etc/group




echo "hi" > greet.txt
ls -l greet.txt
-rw-rw-r-- 1 colt colt 6 Oct 7 14:34 greet.txt

File Attributes

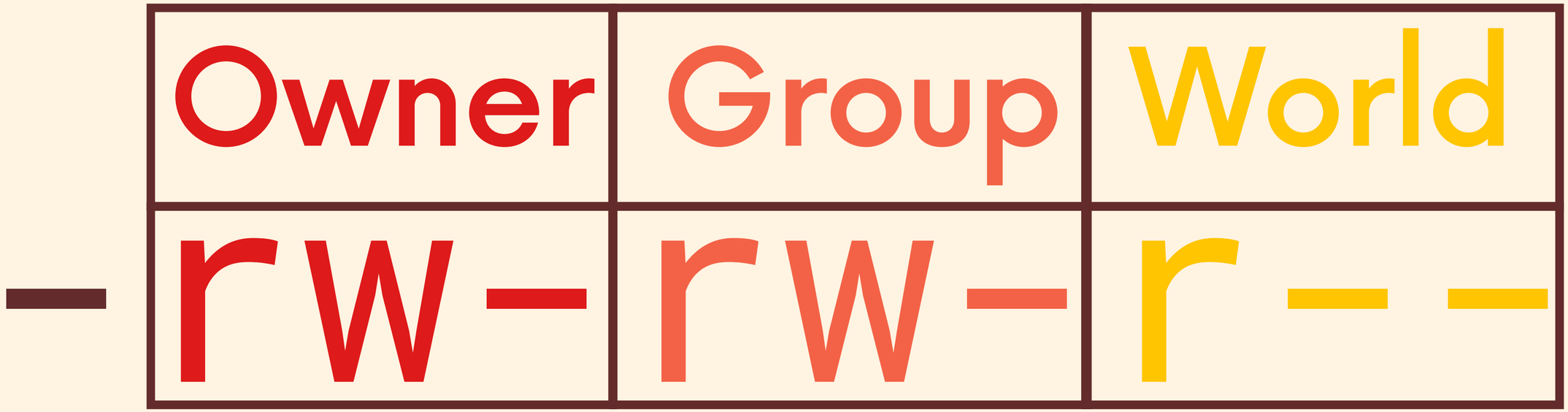
The weird looking 10 characters we see printed out first are the file attributes.

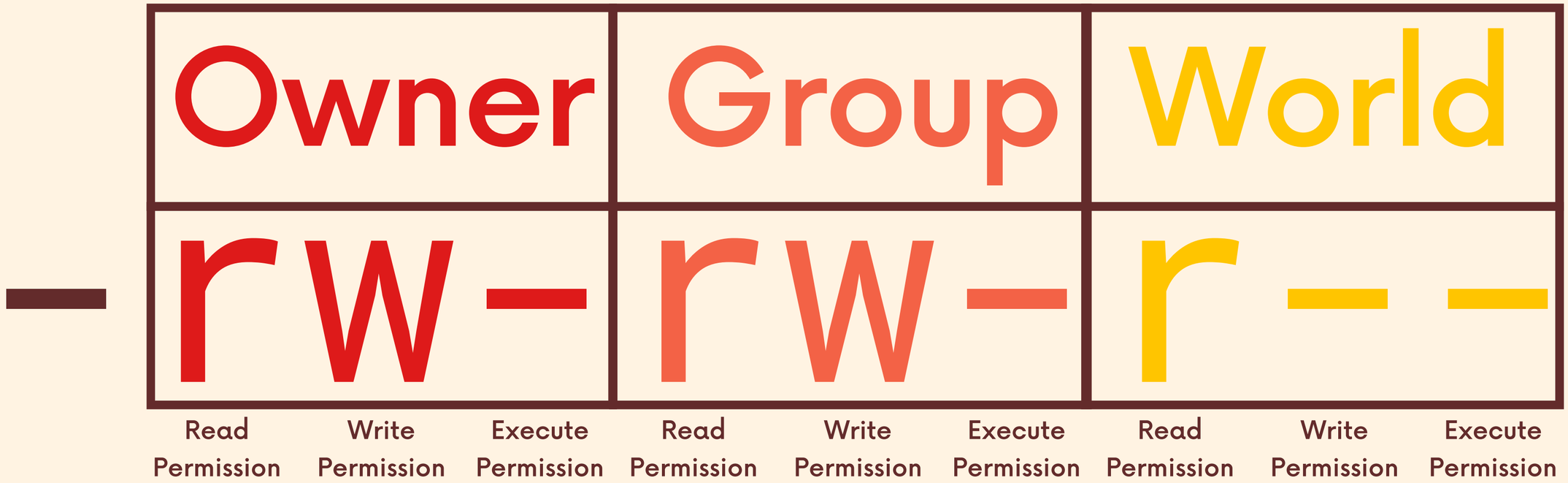
These characters tell us the type of the file, the read, write, and execute permissions for the file's owner, the file's group owner, and everyone else.

File Type

The very first character indicates the type of the file. Some of the more common types and their corresponding attributes are:

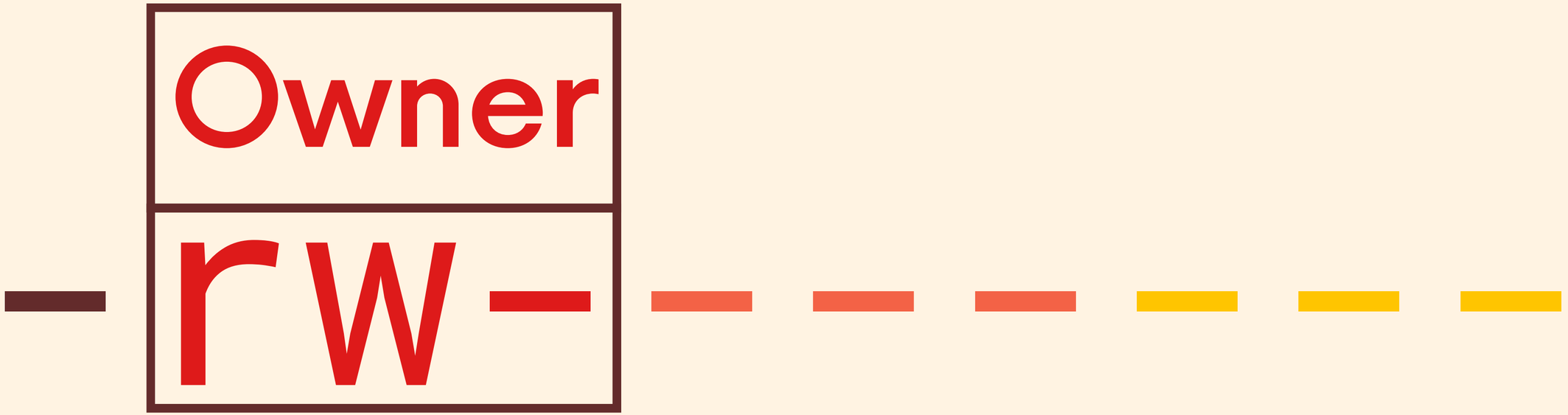
- `-` regular file
- `d` directory
- `c` character special file
- `l` symbolic link


Owner
Group
World
-
rw-
rw-
r--


Owner
Group
World
-rw-rw-r--
Read Permission
Write Permission
Execute Permission
Read Permission
Write Permission
Execute Permission
Read Permission
Write Permission
Execute Permission

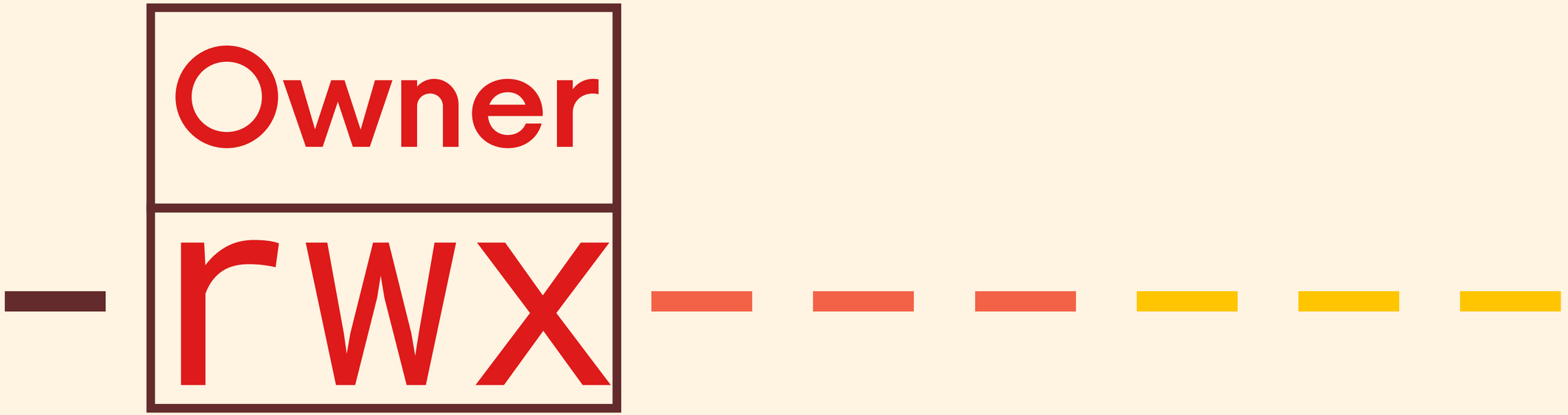
Permissions

Character	Effect On Files	Effect On Directories
r	file can be read	directory's contents can be listed
w	file can be modified	directory's contents can be modified (create new files, rename files/folders) but only if the executable attribute is also set
x	file can be treated as a program to be executed	allows a directory to be entered or "cd"ed into
-	file cannot be read, modified, or executed depending on the location of the - character	directory contents cannot be shown, modified, or cd'ed into depending on the location of the - character



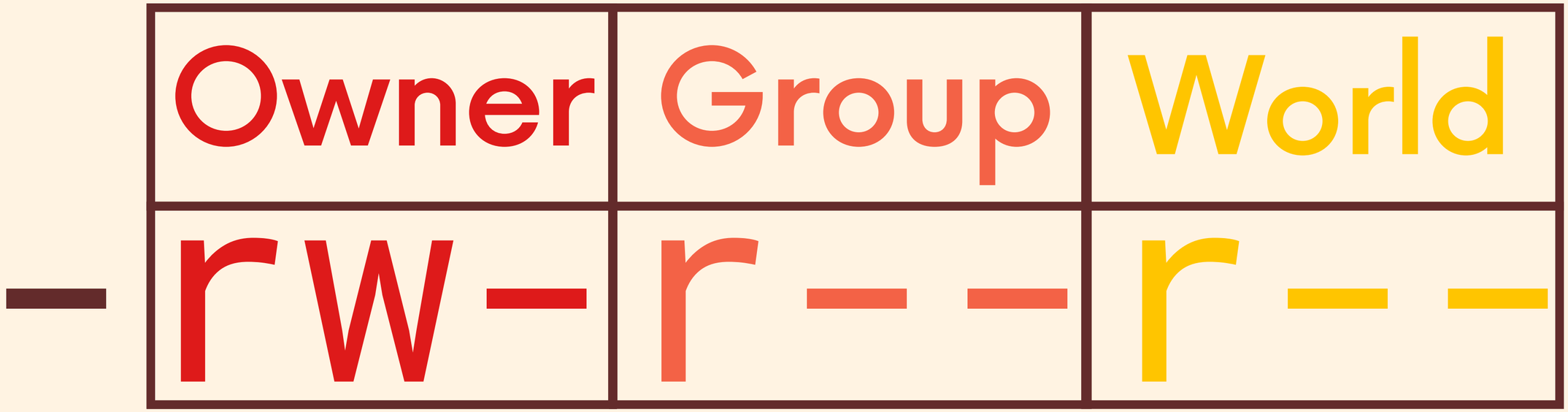
In the above example, we see that the file's owner has read and write permissions but NOT execute permissions

No one else has any access



In the above example, we see that the file's owner has read, write, AND execute permissions.

No one else has any access



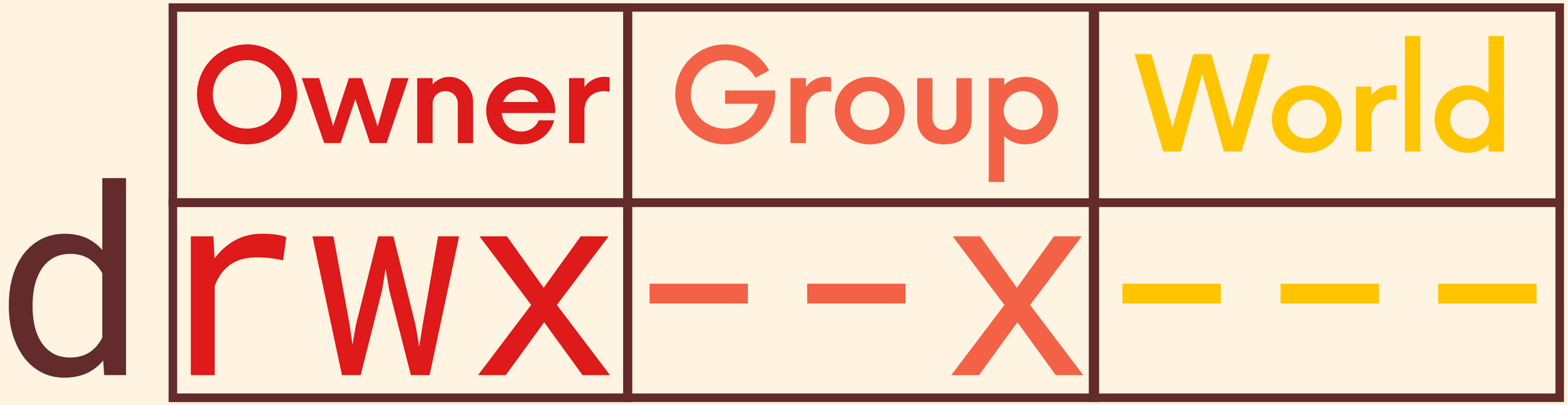
In the above example, we see that the file's owner has read, and write BUT NOT execute permissions.

Members of the file's owner group can only read the file

Everyone else can read the file too.

	Owner	Group	World
d	rwx	rwx	- - -

In the above example, we see that the directory's owner AND member's of the owner group can enter the directory, rename, and remove files from within the directory



In the above example, we see that the directory's owner can enter the directory, rename, and remove files from within the directory.

Members of the owner group can enter the directory but cannot create, delete, or rename files.

Altering Permissions

chmod

To change the permissions of a file or directory, we can use the chmod command (change mode).

To use chmod to alter permissions, we need to tell it:
- **Who** we are changing permissions for
- **What** change are we making? Adding? Removing?
- **Which** permissions are we setting?

chmod

When specifying permissions with chmod, we use a special syntax to write permission statements.

First, we specify the "who" using the following values:

- **u** - user (the owner of the file)
- **g** - group (members of the group the file belongs to
- **o** - others (the "world"
- **a** - all of the above

chmod

Next, we tell chmod "what" we are doing using the following characters:

- - (minus sign) removes the permission
- + (plus sign) grants the permission
- = (equals sign) set a permission and removes others

Finally, the "which" values are:

- r - the read permission
- w - the write permission
- x - the execute permission

All Togther Now

Next, we tell chmod "what" we are doing using the following characters:

- - (minus sign) removes the permission
- + (plus sign) grants the permission
- = (equals sign) set a permission and removes others

Finally, the "which" values are:

- r - the read permission
- w - the write permission
- x - the execute permission

	Owner	Group	World
-	rw-	r--	r--

```
❯ chmod g+w file.txt
```

Add write permissions to the group

	Owner	Group	World
-	rw-	rw-	r--

	Owner	Group	World
-	rw-	rw-	r--

```
❯ chmod a-w file.txt
```

Remove write permissions from all

	Owner	Group	World
-	r--	r--	r--

	Owner	Group	World
-	rw-	rw-	r--

```
❯ chmod u+x file.txt
```

Add executable permissions for owner

	Owner	Group	World
-	rwx	rw-	r--

	Owner	Group	World
-	rwx	rwx	r--

```
❯ chmod a=r file.txt
```

Set permissions to read ONLY for all.

	Owner	Group	World
-	r--	r--	r--

chmod octals

chmod also supports another way of representing permission patterns: octal numbers (base 8). Each digit in an octal number represents 3 binary digits.

Octal	Binary	File Mode
0	000	---
1	001	--x
2	010	-w-
3	011	-wx
4	100	r--
5	101	r-x
6	110	rw-
7	111	rwx

Octal	Binary	File Mode
0	000	---
1	001	--x
2	010	-w-
3	011	-wx
4	100	r--
5	101	r-x
6	110	rw-
7	111	rwx

```
❯ chmod 755 file.txt
```

Owner	Group	World
111	101	101

Owner	Group	World
rwx	r-x	r-x

Changing Our Identity

There may be times we want to start a shell as another user, from within our own shell session.

We can use the su command to do just that.

For example, **su - hermione** would create a new login shell for the user hermione. We would need to enter Hermione's passsword.

To leave the session, type **exit**.

Root User

In Linux systems, there is a super user called root. The root user can run any command and access any file on the machine, regardless of the file's actual owner.

The root user has tons of power and could easily damage or even destroy the system by running the wrong commands!

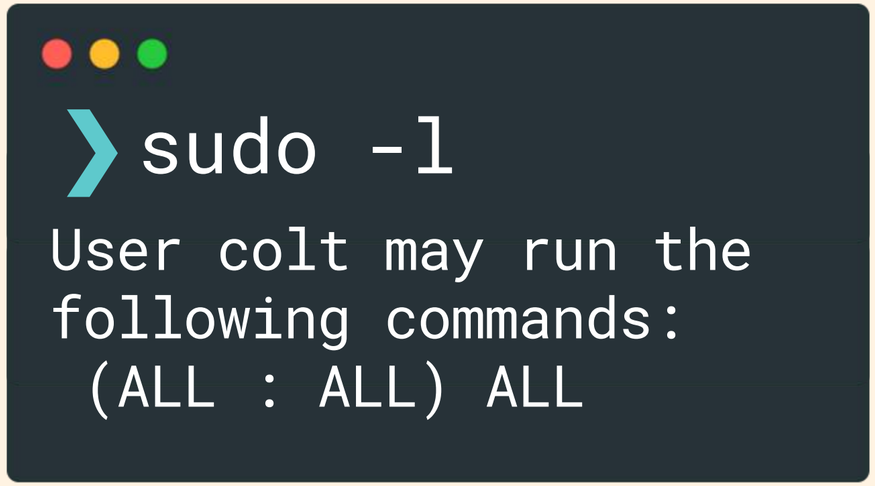
For this reason, Ubuntu locks the root user by default.

Sudo

Even if the root user is locked by default, we can still run specific commands as the root user by using the **sudo** command.

Individual users are granted an "allowed" list of commands they can run as the super user.

Run sudo -l to see the permitted commands for your particular user.

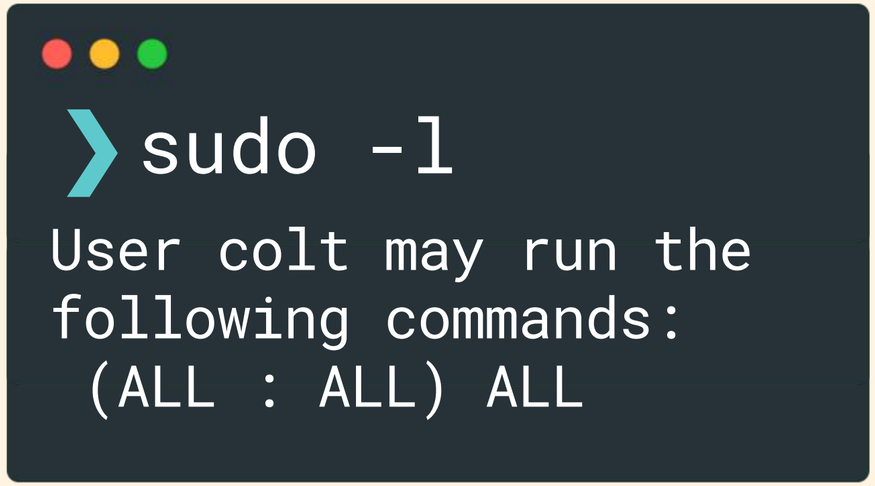


Sudo + Ubuntu

By default, Ubuntu disables logins to the root account. Instead, the initial user is granted full access to all superuser priveleges:

"User colt may run the following commands:
 (ALL : ALL) ALL"

Subsequent users won't have full sudo privileges by default, but the original user can grant them.

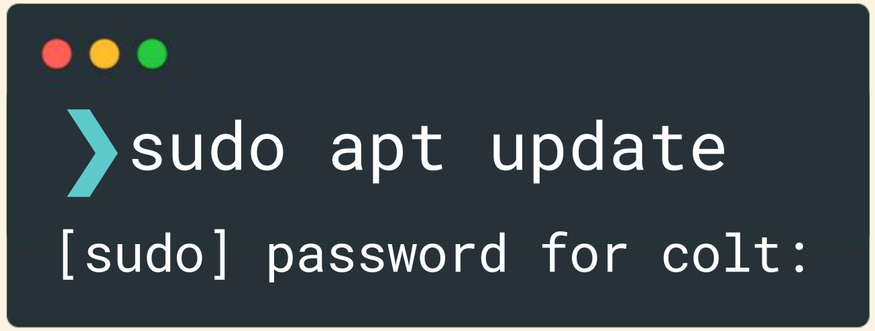


Sudo

To run a command as the root user, prefix it with sudo. You will then need to enter the password for **your account**.

For example to update Ubuntu, I would need to run apt update. However, I can't do this as my "regular" user, as it's something that impacts all users.

Instead, I need to run the command as the root user using sudo apt update



chown

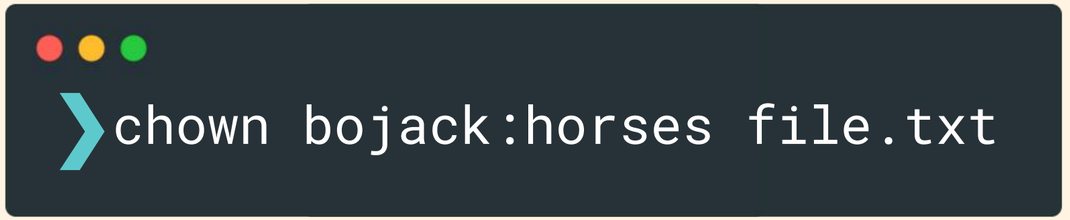
The **chown** command is used to change the owner and/or the group owner of a specific file or directory.

To make bojack the owner of file.txt, we would run **chown bojack file.txt**

chown

To change the owner of a file and the file group owner at once we can provide both using **chown USER:GROUP FILE**.

For example, **chown bojack:horses file.txt** will change the owner of file.txt to bojack AND changes the file group owner to the group named horses.



chown

To only change the group owner of a file, we can run **chown :GROUP FILE**

For example, **chown :horses file.txt** will change the file group owner of file.txt to the group named horses.

groups

To view the groups that a given user belongs to, run **groups USERNAME**

For example, to see which groups hermione is part of, run **groups hermione**

Creating Groups

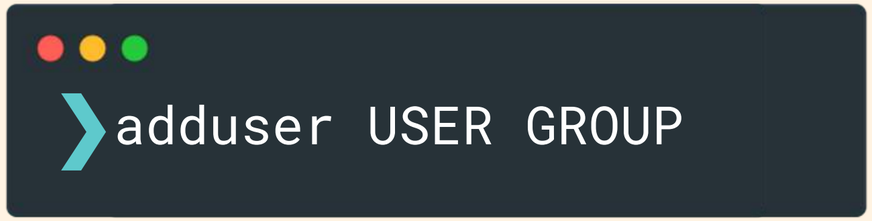
We can create new groups using the addgroup command.

To create a new group called friends, we would run **addgroup friends**

Adding Group Members

To add a user to a group, use the **adduser** command.

To add hermione to friends, we would run **adduser hermione friends**



Don't screw around with groups unless you know what you are doing!
This is just a 30 second intro.

The Environment

The Environment

The shell maintains a set of information during a shell session, known as **the environment.** It's just a series of key-value pairs that define properties like:

- Your home directory
- Your working directory
- The name of your shell
- The name of the logged in user

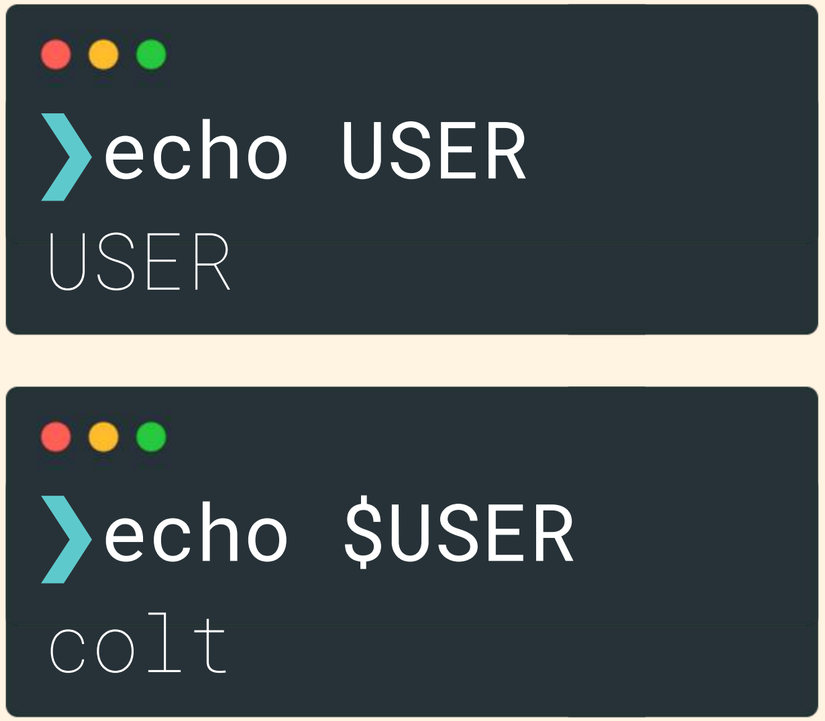
Viewing The Environment

Use the **printenv** command to view the environment variables and their current values. Because there are quite a few values, it can be useful to pipe the output to less.

Parameter Expansion

If we write out the name of an environment variable prefixed with a dollar sign ($), the shell will replace it with the actual value.

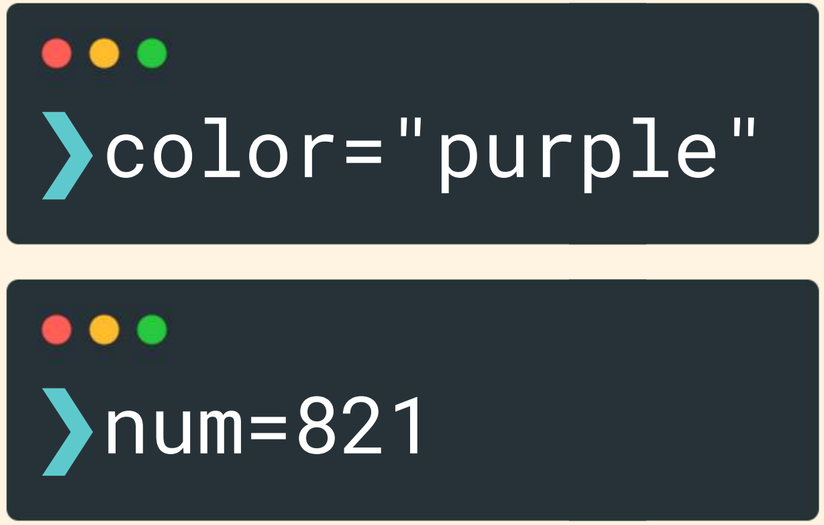
For example, **echo $USER** results in the USER variable's value.



Defining Variables

To define a variable, use the syntax **variable=value**

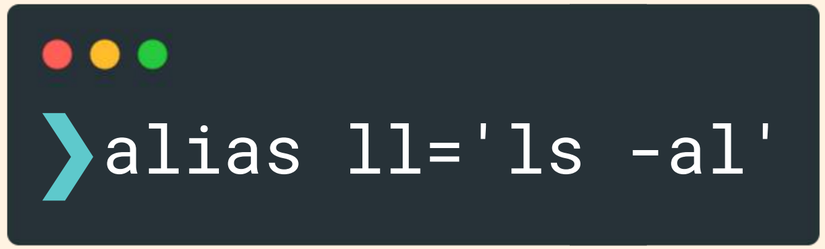
Built-in variables are upper-cased, so it's a common convention to lowercase custom variables to prevent confusion.



Aliases

We can define our own commands using the **alias** keyword.

In the example to the right, we are defining an alias called ll which is equivalent to running ls -al. To execute it, we would simply run ll

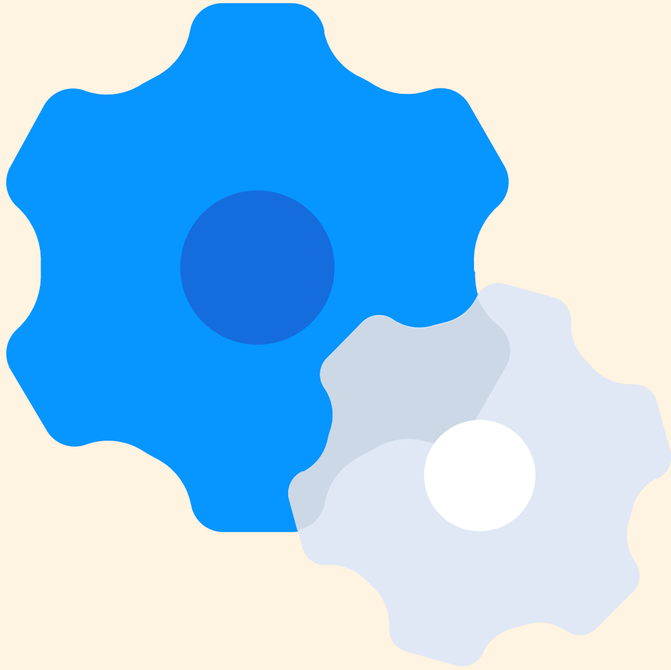


Startup Files

When we log in, the shell reads information from startup files. First, the shell reads from global config files that effect the environment for all users. Then, the shell reads startup files for specific users.

The specific files the shell reads from depends on the type of session: login vs. non-login shell sessions

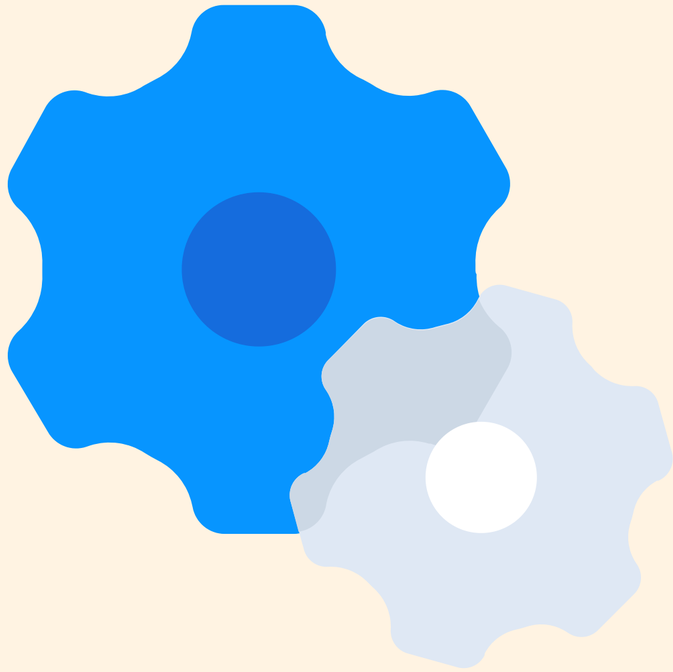
For login sessions:
- /etc/profile - global config for all users
- ~/.bash_profile - user's personal config file
- ~/.bash_login - read if bash_profile isn't found
- ~/.profile - used if previous two aren't found



Startup Files

For non-login sessions (typical session when you launch the terminal via the GUI):

- etc/bash.bashrc - global config for all users
- **~/.bashrc** - specific settings for each user. This is where we can define our own settings and configuration



Bash Scripts

The Basic Steps

1. Write a script in a file and save it
2. Make the script executable using chmod
3. Verify that the shell can find your script

Shebang!

The first line of our script should read **#!/bin/bash**

The #! is called the shebang, and it's used to tell the OS which interpreter it should use when parsing this file.
We want ours to say "use bash to interpret this file!"

After the shebang, we need to include the path to the Bash binary. This is not Bash specific. If we wanted to write a python script, we would include the path to the python binary.

```
#!/bin/bash

#my first script

echo "hello everyone"
```

Comments

Lines that begin with # will not be read by the shell. Write comments to explain particularly tricky bits of code or to leave notes for yourself.

```
#!/bin/bash

#my first script

echo "hello everyone"
```

The Good Stuff

We can write any of the commands that we normally run from the command line. When we execute the script, these commands will run!

```
#!/bin/bash

#my first script

echo "hello everyone"
```

Executing The Script

We can execute the script the "long way" by running **bash pathToFile.**

This works, but it's not as convenient as it could be! What if we could instead just run **hello** from anywhere on our machine, just like we can run **ls** or **grep** anywhere?

Locating Commands

When we run a command like pwd, the shell starts looking for the executable pwd program in the list of directories stored in the **PATH** variable.

It starts looking in the first location and then keeps looking if it doesn't find it.

Locating Commands

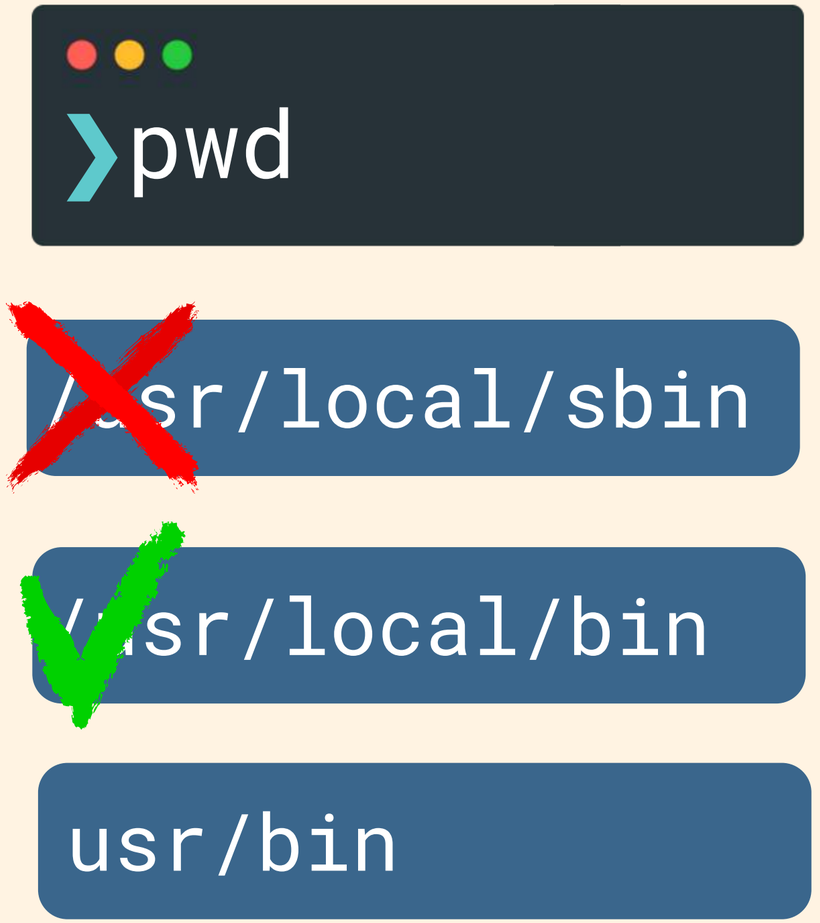
When we run a command like pwd, the shell starts looking for the executable pwd program in the list of directories stored in the PATH variable.

It starts looking in the first location and then keeps looking if it doesn't find it.

Locating Commands

When we run a command like pwd, the shell starts looking for the executable pwd program in the list of directories stored in the PATH variable.

It starts looking in the first location and then keeps looking if it doesn't find it.

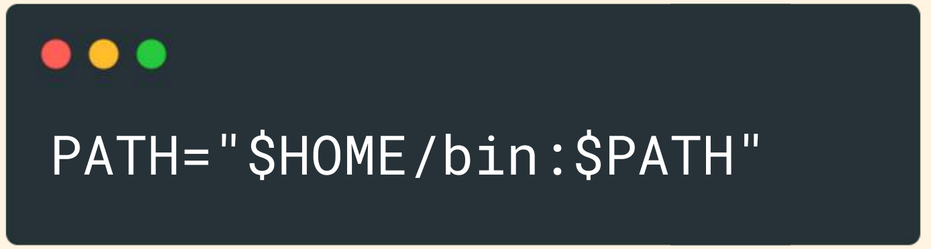


Why It Matters

If we want the shell to find our own programs, we need to make sure we put them in a folder that is in the **PATH** variable.

A common place to put user-defined programs is in a bin folder located in the user's home directory. For me that would be **/home/colt/bin.**

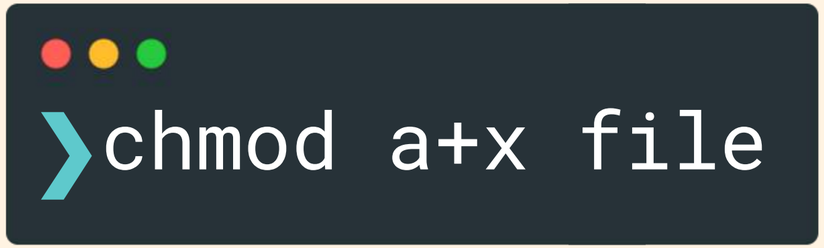
If that directory is not yet part of your path, you can add it by putting **PATH="$HOME/bin:$PATH"** in your .bashrc file



If ~/bin is not yet in your PATH, add the above line to your .bashrc file

Making It Executable

The next step is to make the file containing our script executable. **chmod a+x file** will grant executable permissions to everyone.



Cron

Cron

The **cron** service allows us to schedule commands to run at regular intervals like:

- Every 30 minutes
- Every day at midnight
- Every 1st of the month
- Every December 15th

Editing the crontab

To set up a cron job, we need to edit the crontab configuration file. Rather than edit the files directly, it's best to use the **crontab -e** command.

Cron Syntax

a b c d e command

Minute	Hour	Day	Month	Day(of week)
0-59	0-23	1-31	1-12	0-6

Cron Characters

*	Any value
5,6	List of values (5 and 6)
1-4	Range of values (1 to 4)
*/5	Step values (every 5)

Minute	Hour	Day	Month	Day(of week)
0	4	8-14	*	*
0-59	0-23	1-31	1-12	0-6

Run a job **at minute 30**, every hour
(every time the clock shows x:30)

```
30 * * * * command
```

Minute	Hour	Day	Month	Day(of week)
0-59	0-23	1-31	1-12	0-6

Run a job **every day at midnight**
(when hour is 0 and minute is 0)

```
0  0  *  *  *  command
```

Minute	Hour	Day	Month	Day(of week)
0-59	0-23	1-31	1-12	0-6

Run a job **every day at 6:30AM**
(when hour is 6 and minute is 30)

```
30 6 * * * command
```

Minute	Hour	Day	Month	Day(of week)
0-59	0-23	1-31	1-12	0-6

Run a job **every monday at 6:30AM**

30	6	*	*	1	command
Minute 0-59	Hour 0-23	Day 1-31	Month 1-12	Day(of week) 0-6	

Run a job **every monday in April at 6:30AM**

30	6	*	4	1	command
Minute 0-59	Hour 0-23	Day 1-31	Month 1-12	Day(of week) 0-6	

Run a job **at midnight**
on the first of every month

```
0 0 1 * * command
```

Minute	Hour	Day	Month	Day(of week)
0-59	0-23	1-31	1-12	0-6

Run a job **at midnight**
every weekday (monday-friday)

```
0  0  *  *  1-5  command
```

Minute	Hour	Day	Month	Day(of week)
0-59	0-23	1-31	1-12	0-6

Run a job **every 5 minutes**

```
*/5 * * * * command
```

Minute	Hour	Day	Month	Day(of week)
0-59	0-23	1-31	1-12	0-6