The Secret Rules of the Terminal

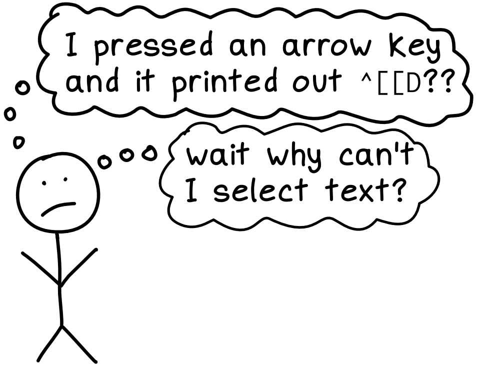
by Julia Evans

about this zine

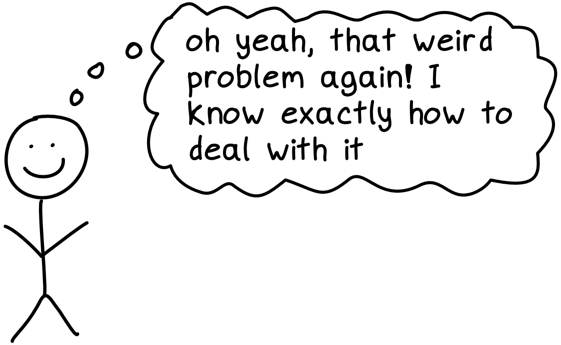
the Unix terminal seems simple at first (just type in commands and run them!).

but the documentation about how the terminal actually works is incredibly scattered and patchy, and a lot of things aren't documented at all because they're just "conventions" that "everyone knows". It makes everything take way longer to learn than it should.

this zine's goal is to help you get from:


I pressed an arrow key and it printed out ^[[D??
wait why can't I select text?

to

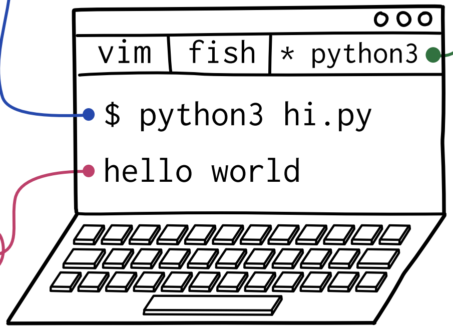

oh yeah, that weird problem again! I know exactly how to deal with it

(this zine comes with a cheat sheet! `https://wizardzines.com/terminal-cheat-sheet.pdf`)

table of contents

SHELL

TERMINAL EMULATOR



```
$ python3 hi.py
hello world
```

BYTES

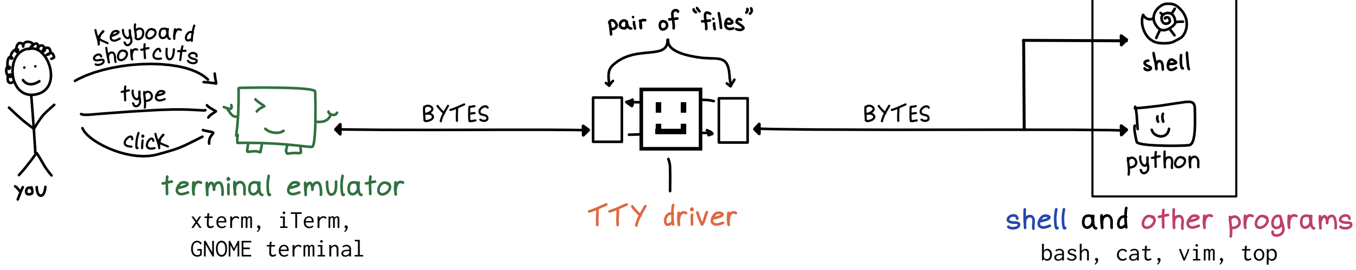
PROGRAMS

BYTES

TTY DRIVER

cast of characters

The "terminal" is actually a bunch of components that work together.
Let's imagine that you're running `python3 blah.py`.



your terminal emulator is a translator:

→ it translates all your typing/clicks into bytes

→ and it takes all the bytes the program sends and displays them on the screen

(more on page 14)

the TTY driver is part of your operating system:

⇄ the terminal emulator & programs communicate by reading/writing to a pair of files

⧉ the TTY driver is in the middle and copies bytes back & forth with some small changes

(more on page 24)

Your shell is a program that you use to start all other programs.

The shell doesn't do much after it starts a program. Programs get a copy of the shell's environment variables & a few other things and then they're on their own.

meet the shell

the shell starts programs

when you run a program in the terminal, you're actually asking your shell to start it for you

it turns out that starting programs is a surprisingly complicated job!

the 3 most popular shells

there are LOTS of shells but 95% of people use

bash or zsh or fish

bash: default on Linux

zsh: default on Mac (in 2025)

fish: aims to be more user friendly

fish: the friendly interactive shell <3

I love how fish has friendly defaults that I can use without configuring it

this is (mostly) not a fish propaganda zine though

bash and zsh are both "POSIX shells"

this means they follow a standard for how Unix shells should behave, but there are still differences

I'll mention when something varies between shells!

where to find your shell's config file

bash:

`~/.bashrc` `~/.bash_profile`

which one is a rabbit hole, huge flow chart at wzrd.page/bashrc

zsh:

`~/.zshrc`

fish: <3

`~/.config/fish/config.fish`

.bashrc vs .bash_profile

here's an trick to figure out whether bash is using .bashrc or .bash_profile (or both!)

Add:

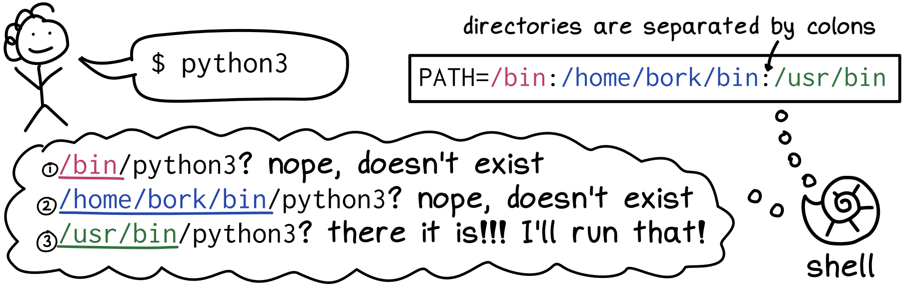
```
echo "this is .bashrc"
echo "this is .bash_profile"
```

to each file, open a new terminal tab, and see what it prints out!

PATH

PATH is how your shell knows where to find programs

It's a list of directories that your shell searches in order.



directories are separated by colons

```
PATH=/bin:/home/bork/bin:/usr/bin
```

$ python3
① /bin/python3? nope, doesn't exist
② /home/bork/bin/python3? nope, doesn't exist
③ /usr/bin/python3? there it is!!! I'll run that!
shell

how to add a program to your PATH

① find the directory the program is in

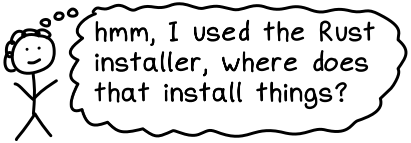
② update PATH in your shell config with that directory

③ restart your shell

for WAY TOO MUCH info about how to do this, see https://wzrd.page/path

... but which directory was the program installed in?

remember how you installed it:



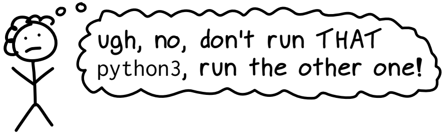
hmm, I used the Rust installer, where does that install things?

... or do a brute force search:

```
find / -name python3 | grep bin
```

(usually I put a 2>/dev/null too)

PATH ordering drama



You can prioritize a directory by adding it to the beginning of your PATH

gotcha: not everything uses your shell's PATH

cron jobs usually have a very basic PATH, maybe just /bin and /usr/bin

in a cron job I'll use the absolute path:

```
/home/bork/bin/someprogram
```

PATH tips

add a directory to your PATH

at the end:

```
export PATH=$PATH:/my/dir
```

at the beginning:

```
export PATH=/my/dir:$PATH
```

in fish the syntax is different, like:

```
set PATH $PATH /my/dir
```

show the first match on your PATH for a program

```
which python3
```

(but in zsh which acts like type)

show ALL matches on your PATH for a program, in order

```
which -a python3
```

look at your PATH

```
echo $PATH
```

show each entry on its own line

```
echo "$PATH" | tr ':' '\n'
```

needs quotes to work in fish

zsh has nice PATH syntax

```
path=(
  $path
  ~/.cargo/bin
  ~/bin
)
```

path is an array that zsh syncs with the PATH environment variable

show what your shell is *actually* going to run with type

```
type python3
```

Your shell doesn't always run a program! Instead of what's in PATH, sometimes it'll run a builtin or alias or cached entry

weird fact: bash and zsh cache PATH entries

this cache gets cleared every time you restart your shell and every time you update PATH so it rarely causes problems

but if you need to you can clear it with:

```
hash -r
```

history

your shell has a history of the commands you ran

some ways to access history:

- ☆ press the up arrow
- ☆ run `history`
- ☆ search it with `Ctrl+R`
- ☆ use `!33` to rerun line 33 from `history` (bash/zsh)

how long does your shell store history for?

☹ in bash, the default is 500 commands (not enough!)

☺ in fish, the default is 256,000 commands

if you're using bash, you might want to set `HISTSIZE` and `HISTFILESIZE` to store more history

in zsh, it's `HISTSIZE` and `SAVEHIST`

when does your shell save history?

→ by default, bash and zsh only save history to a file when you exit the shell

→ fish saves the history continuously

where is history stored?

bash: `~/.bash_history`

zsh: run `echo $HISTFILE`

fish: mine is in `~/.local/share/fish/fish_history`

sometimes I copy over my shell history when setting up a new computer!

history doesn't include everything

usually it includes:

→ the contents of the history file when the shell started

→ the commands you ran in this shell session

if I want to use the history from another terminal tab, I'll open a new tab

a useful history tool: atuin

atuin lets you:

- ♥ save unlimited history
- ♥ search history more easily
- ♥ save commands as soon as you run them
- ♥ sync your history (optionally)

job control

your shell lets you run many programs ("jobs") in the same terminal tab

programs can either be:

- foreground
- background
- stopped (which is more like "paused")

& runs a program in the background

for example I like to convert 100 files in parallel like this:

```
for i in `seq 1 100`
do
   convert $i.png $i.jpg &
done
```

jobs lists backgrounded & stopped jobs

```
$ jobs
[1]  Running   python blah.py &
[2]  Stopped   vim
```

↑ use the numbers to bring them to the foreground or background (like `fg %2`), kill them (`kill %2`), or disown them

when you close a terminal tab all jobs are killed with a SIGHUP signal

you can stop this with `disown` or by starting the program with `nohup`:

disown %1 ← job number goes here

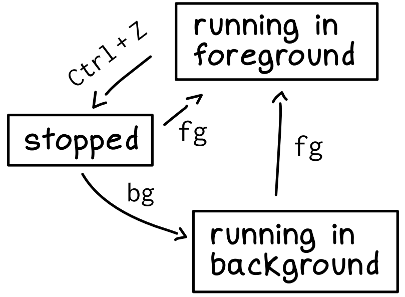
```
nohup my_program &
```

a trick to kill programs if Ctrl+C doesn't work

① press `Ctrl+Z` to stop the program

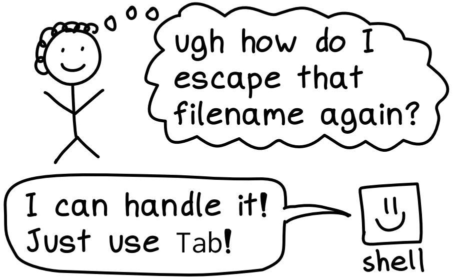
② run `kill %1` to kill it (or `kill -9 %1` if you're feeling extra murderous)

a little flowchart



filename tips

your shell can help you type weird filenames


ugh how do I escape that filename again?
I can handle it! Just use Tab!
shell

tab completion can go wrong

programs can change how tab completion works with plugins called "completions"

this is usually GREAT (`git add <Tab>` only completes modified files!) but sometimes it's buggy

cycle through matching filenames

```
rm f<Tab><Tab><Tab><Tab>
```

(doesn't work in bash unless you configure it)

tab complete from the middle of a filename

```
ls *thing*<Tab>
```

(or in fish `ls thing<Tab>`)

configure bash to cycle through matching filenames

Add this to your `~/.inputrc`:

```
set show-all-if-ambiguous on
set menu-complete-display-prefix on
TAB: menu-complete
```

quote filenames with spaces

```
cat "Julia Evans.txt"
```

(if you don't do this you get weird "file not found" errors for `Julia` and `Evans.txt`)

tab completion works inside quoted strings

```
cat "File N<Tab
```

more filename tips

handle filenames starting with a dash with -- or ./

```
mv -- -file.txt dest
```

```
mv ./-file.txt dest
```

(otherwise `mv` thinks `-file.txt` is an invalid option)

lots of tools support --

for example if you want to grep a file for the text "-x" you can run:

```
grep -- -x file.txt
```

-- means "nothing after this is an option"

match all filenames ending in .png

```
rm *.png
```

(`*.png` is called a "glob" and it's handled by the shell so you can use it with any program!)

match .png files in any subdirectory

```
rm **/*.png
```

(works in zsh/fish, and in bash with `shopt -s globstar`)

you can drag files from your GUI file manager to escape the filename

This only works if your terminal emulator supports it.

match filenames starting with a dot

```
ls .*
```

(dotfiles aren't included in * by default)

gotcha: `.*` in older versions of bash (pre 5.2) includes . and ..

* gotcha: regular expressions

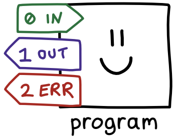
if you want to pass a regexp with a * to grep

```
grep 'def .*' file.txt
```

you need to quote it otherwise it will be treated as a glob

stdin, stdout, stderr

terminal programs have 1 input and 2 outputs



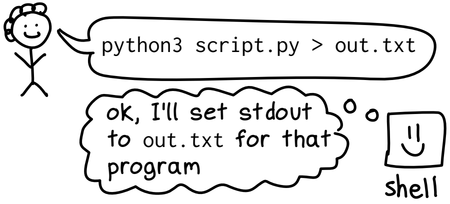
they're numbered: stdin is 0, stdout is 1, stderr is 2

(the numbers are called "file descriptors")

3 things you can set the inputs/outputs to

1. the TTY (the default: display output in your terminal emulator)
2. a file
3. a pipe (send output to another program's input)

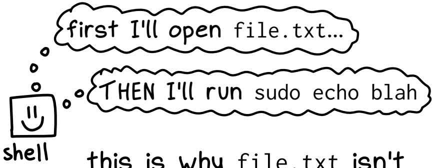
your shell is in charge of setting up stdin/stdout/stderr



```
python3 script.py > out.txt
```

when you redirect, the shell opens the file before the program starts

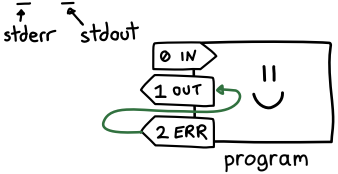
```
sudo echo blah > file.txt
```



this is why file.txt isn't opened as root!

on 2>&1

2>&1 redirects stderr to stdout



you could also do `echo "oops" 1>&2` if you want to write a message to stderr in a script

gotcha: programs often buffer stdout but not stderr

when a program writes text to stdout, it'll often

→ check if stdout is a TTY (using the isatty function)

→ if not, "buffer" the writes until there's 1KB of data to write, for performance reasons

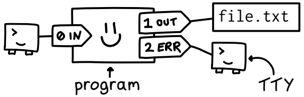
(this is the default in libc)

CHEAT SHEET

redirects

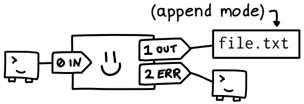
redirect to a file:

```
cmd > file.txt
```



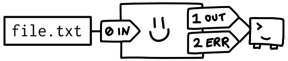
append to a file:

```
cmd >> file.txt
```



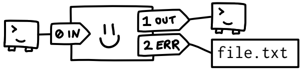
send a file to stdin:

```
cmd < file.txt
```



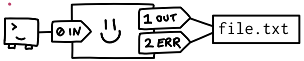
redirect stderr to a file:

```
cmd 2> file.txt
```



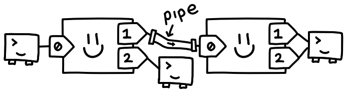
redirect stdout AND stderr:

```
cmd > file.txt 2>&1
```



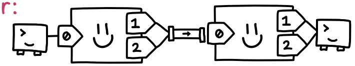
pipe stdout:

```
cmd1 | cmd2
```



pipe stdout AND stderr:

```
cmd1 2>&1 | cmd2
```



three gotchas

① `cmd file.txt > file.txt`

will delete the contents of file.txt!

some people use `set -o noclobber` (in bash/zsh) to avoid this

But I just have "never read from and redirect to the same file" seared into my memory.

② `sudo echo blah > /root/file.txt`

doesn't write to /root/file.txt as root. Instead, do:

```
echo blah | sudo tee /root/file.txt
```

-or-

```
sudo sh -c 'echo blah > /root/file.txt'
```

③ `cmd 2>&1 > file.txt`

doesn't write both stdout and stderr to file.txt. Instead, do:

```
cmd > file.txt 2>&1
```

cat vs <

I almost always prefer to do:

```
cat file.txt | cmd
```

instead of

```
cmd < file.txt
```

it usually works fine & it feels better to me

using `cat` can be slower if it's a GIANT file though

&> and |&

bash and zsh support `&>` and `|&` to redirect/pipe both stdout and stderr

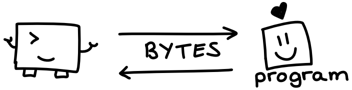
(in fish it's `&|` instead of `|&`)

meet the terminal emulator

your terminal emulator has two main jobs

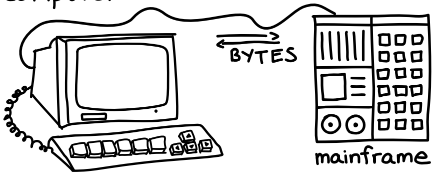
① turn your actions (typing & clicking) into bytes and send them

② receive bytes and display them visually



a little history

it's called an "emulator" because in the 80s a "terminal" was a separate machine from the computer



We still use the same 80s protocol!

what are these "bytes"?

the bytes are either:

→ text (like `cat blah.txt`)

→ escape codes (for example to tell the terminal what colour to display the text in)

→ control characters (for example `Ctrl+C` is the byte `3`)

it's in charge of copy and paste

your terminal emulator lets you select text and copy/paste it (usually with `Ctrl+Shift+C` or `Cmd+C`)

Linux

Mac

(copy & paste tips on page 18!)

it manages colours and fonts!

some terminal emulators come with a big library of colourschemes!

if yours doesn't, this site has colourschemes for many terminal emulators:

`iterm2colorschemes.com`

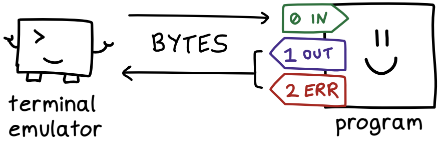
fun fact: how `Ctrl-X` gets translated to bytes

```
Ctrl-A => 1
Ctrl-B => 2
...
Ctrl-Z => 26
```

`Ctrl` and `Shift` are the only modifiers I trust in the terminal, all of the others work differently depending on the situation

escape codes

a program's input and outputs are streams of bytes

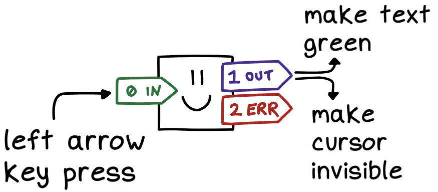


everything you type goes into standard input 0 IN ← except for maybe Ctrl+C/Z/T/S/U/D/Q

all the output you see comes from either standard output 1 OUT or standard error 2 ERR

some inputs/outputs are text and some are special instructions

examples of special instructions:



these special instructions are called "escape codes"

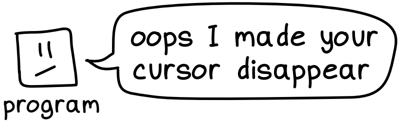
they're called "escape codes" because they all start with the ESC character

```
\033    ^[    ESC    \e    \x1b
```

↖ ↑ ↑ ↗ ↗

five ways people print out ESC

programs can easily "break" your terminal by printing escape codes



It's easy to fix though: run `reset` to print a special escape code that resets everything

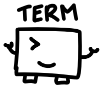
how reset works

`reset` is basically the same as running these 3 commands:

```
tput reset
sleep 1
stty sane
```

`stty sane` ↗ tells the TTY driver to reset (more on page 25)

`tput reset` ← prints the "reset" escape codes



colours

your terminal emulator has 16 configurable colours

	normal	bright
black	0	0
red	1	1
green	2	2
yellow	3	3
blue	4	4
purple	5	5
cyan	6	6
white	7	7

these are called "ANSI colours"

you can configure them in your terminal emulator's settings

-- OR --

run a shell script that prints escape codes to magically set up your colours

https://wzrd.page/scripts

↖(my favourite way!)

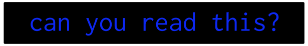
programs can use ANSI colours by printing an escape code

```
echo -e "\033[34m blue text"
```

3 means "normal fg colour"
4 means "blue"

the default ANSI colours often have bad contrast

`ls --color` often displays directories in ANSI "blue" which can look like this:



```
can you read this?
```

ANSI "yellow" on white also often has bad contrast

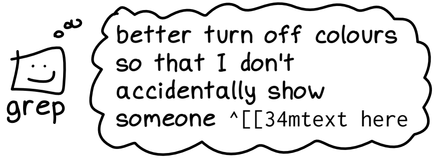
♡ "minimum contrast" ♡

Picking ANSI colours that always have good contrast is impossible.

the only real solution is to use a terminal emulator with a "minimum contrast" feature (like iTerm or kitty) that will fix all contrast issues

usually if a program is writing to a pipe, it'll disable colours

```
$ grep blah file.txt | less
```


grep
better turn off colours so that I don't accidentally show someone ^[[34mtext here

the mouse

when you click in the terminal, it can either be handled by

your terminal emulator or the program

↑ good if you want to copy text

↑ lots of programs have mouse support!

programs can tell the terminal emulator to let them handle the mouse

program: if there's a mouse click, send me escape codes to tell me where it was!

okay! I'll disable all my usual mouse functions like "selecting text"!

this is called "mouse reporting"

some programs that have mouse support

tmux: resize a panel right click for a menu!

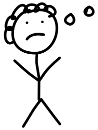
htop: click to sort columns!

micro ↑ text editor with good mouse support

vim ↑ click on the tab bar!

and LOTS more! (lazygit, mc, zellij, btop...)

how to force the terminal emulator to handle the mouse: press Shift or Option*



ugh no I don't want to focus that pane, I want to COPY SOME TEXT!!!

* could be something else too, it depends on your terminal emulator

the scroll wheel

In some programs (like less) the scroll wheel does the same thing as pressing up/down arrow keys really fast

UP UP UP UP UP UP UP UP UP UP UP UP UP UP UP UP

in other programs (like lazygit) it uses "mouse reporting" to report where your mouse was when you scrolled

other mouse features your terminal emulator might have

→ Shift+click (or something) to open a link in a browser

→ Alt+click (or maybe Option) to move the cursor when editing a command in your shell

copy and paste

safe multiline paste

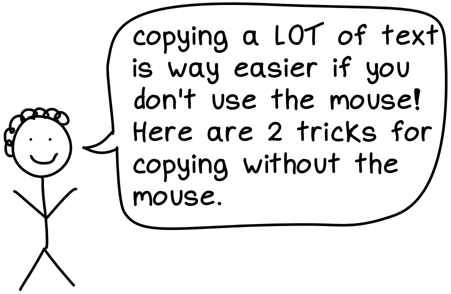
It's SO scary when you paste a bunch of commands by accident and then it runs them all.

fish, zsh, and newer bash versions protect you from this: you have to press Enter before running the thing you pasted. This is called "bracketed paste"

problem: copying with the mouse can go wrong

→ copying 400 lines of text by dragging is nobody's idea of a good time

→ sometimes extra whitespace that you didn't want gets added at the end of lines


copying a LOT of text is way easier if you don't use the mouse! Here are 2 tricks for copying without the mouse.

copy trick 1: pbcopy

macOS comes with two programs that can copy from stdin / paste to stdout, like this:

```
cat main.go | pbcopy
```

They're SO useful and on Linux I like to write my own versions of pbcopy/pbpaste using xsel. There's also wl-copy/wl-paste.

pbcopy over SSH

you can even implement pbcopy over SSH (yes really!) with this bash one-liner.

It uses an escape code called "OSC 52".

```
printf "\033]52;c;%s\007"
       "$(base64 | tr -d '\n')"
```

get it at https://wzrd.page/pbcopy

copy trick 2: syncing the vim clipboard

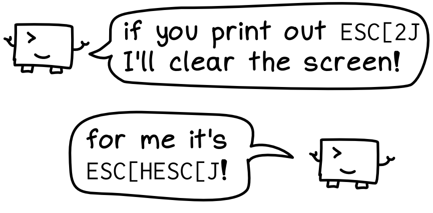
I use vim as a terminal text editor, and I find it's WAY easier if I sync my system clipboard with the vim clipboard like this:

```
set clipboard=unnamed
```

tmux can also copy to your system clipboard.

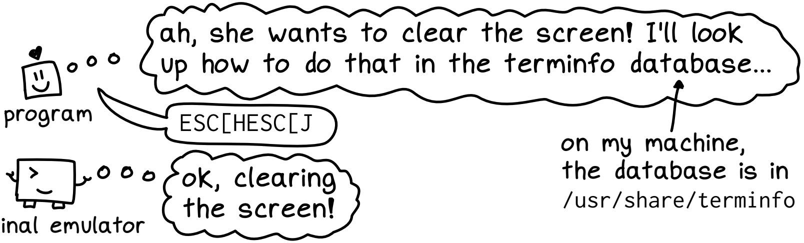
TERM

different terminal emulators use different escape codes



your system has a database called "terminfo" with escape codes in it

here's how it plays out when you press Ctrl+L to clear the screen:


ah, she wants to clear the screen! I'll look up how to do that in the terminfo database...
program
ESC[HESC[J
terminal emulator
ok, clearing the screen!
on my machine, the database is in /usr/share/terminfo

how programs know what terminal you're using:

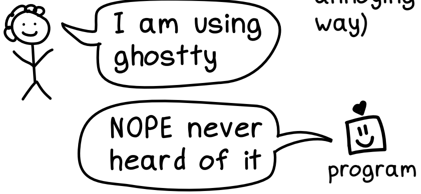
→ TERM ←

your terminal emulator sets the TERM environment variable when it starts

fun fact: terminal emulators often say they're "xterm-256color" even if they're not

this can break when SSHing into an old system with a new terminal emulator

(in a VERY annoying way)

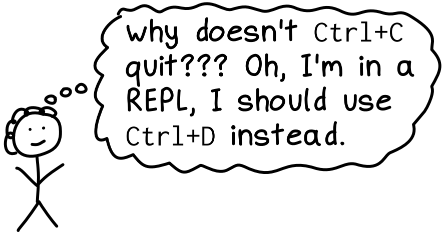


some ways to fix TERM issues

- → install the terminfo file for your terminal emulator on the system
- → use a different terminal emulator
- → just set TERM=xterm-256color, it'll often sort of work

types of programs

knowing what type of program you're in really helps


why doesn't Ctrl+C quit??? Oh, I'm in a REPL, I should use Ctrl+D instead.

REPLs*

(sqlite, ipython, bash)

→ you can probably use basic `readline` shortcuts to edit text

→ `Ctrl+D` usually quits

* REPL stands for Read code, Evaluate it, Print the output, Loop (repeat)

full screen programs

(top, ncdu)

→ `q` might quit

→ `?` might open the help

→ gotcha: if mouse reporting is on, you can't select text without pressing `Shift`

noninteractive programs

(grep, find)

→ `Ctrl+C` usually quits

→ gotcha: you can get "stuck" waiting for input on stdin if you forget to specify an input (like if you run `cat` by itself)

programs that play by their own rules

`vim` doesn't act like any other program.

usually I avoid these unless (like with `vim`) I've made a special effort to learn them

`Ctrl+C` doesn't always quit

REPLs and full-screen programs often use `Ctrl+C` to mean "stop the current operation" instead of "quit the program"

less

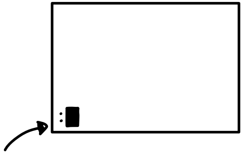
many programs use less without telling you

less lets you scroll through text, so programs will use less by default any time they want to display a lot of text

git
I want to display a huge diff... I'll show it in less!
I need to display a man page... I'll use less!
man

it's called less because it's an improved version of more

how to know you're in less



if it's suddenly full screen and there's this little colon in the bottom left, it might be less

a few less tips

quit:	q
help:	h
scroll:	arrow keys/spacebar/ mouse wheel
search:	/banana ENTER
next/prev match:	n/N
go to start/end:	g/G

also piping to less -R will interpret escape codes like colours

how to tell a program not to use less

you can set the PAGER environment variable to something else to tell programs to use that instead

I've never had any reason to set PAGER though

programs will also drop you into vim sometimes

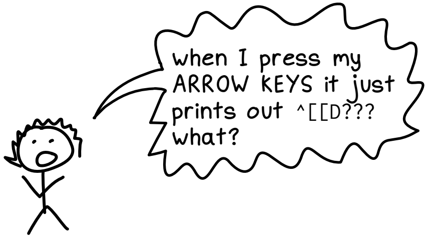
the default text editor is often vim. If you don't like vim you can set the EDITOR environment variable

```
export EDITOR=micro
```

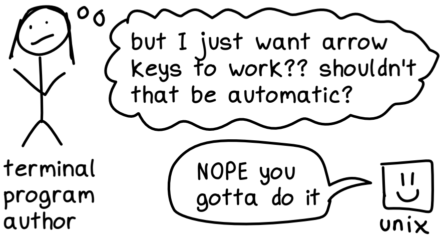
↑ your favourite editor here

editing text in a REPL

editing text in a REPL doesn't always work well



this is because every program has to implement text editing itself


but I just want arrow keys to work?? shouldn't that be automatic?
terminal program author
NOPE you gotta do it
unix

you do get a few things automatically*

- ✭ backspace

  (occasionally backspace won't work and you have to use Ctrl+H instead)
- ✭ Ctrl+W (delete word)
- ✭ Ctrl+U (delete line)

*see page 26 for what "automatically" means

REPLs mostly all have the same keyboard shortcuts

there's a very popular library called "readline", and mostly everyone either uses it or imitates how it works

for example Ctrl+A ("go to beginning of line") comes from readline

rlwrap adds readline keyboard shortcuts

for example on my machine the dash shell doesn't use readline but you can make it better by running:

```
rlwrap dash
```

built-in programs on Mac don't use readline

(for example sqlite3)

this is probably because readline is GPL licensed

They use libedit, which is worse. I like to install a sqlite version with readline support and use that instead.

CHEAT SHEET

keyboard shortcuts

editing text (always works)

(almost)

backspace	
Ctrl + W	delete previous word
Ctrl + U	delete line

(except in text editors)

quitting

Ctrl + C	quit (SIGINT)
Ctrl + Z	stop process (SIGTSTP) (resume with fg or bg or kill with kill)
Ctrl + D	quit (in a REPL) more on page 20
q	quit (in some full screen programs)
Enter ~ .	exit frozen SSH session

or the nuclear option:

```
$ ps aux | grep THING
bork 7213 ... THING
$ kill -9 7213
```

editing text

(these often work in a readline-like situation)

arrow keys	
Ctrl + A or Home	beginning of line
Ctrl + E or End	end of line
Ctrl + arrow keys	left/right a word

(or sometimes Alt + arrow keys
or Option + arrow keys
or Alt+b / Alt+f)

Ctrl + K	delete line forward
Ctrl + Y	paste (from Ctrl+K or Ctrl+U)
Ctrl + H	might work if Backspace doesn't

also many shells have a "vi mode" if that's your jam

other useful stuff

Ctrl + L	clear screen
Ctrl + R	search history
Ctrl + Q	unfreeze screen (that you froze with Ctrl+S)

more on page 25

copy and paste

in your terminal emulator, it's usually:

Ctrl + Shift + C/V
or Cmd + C/V

mouse stuff that might work

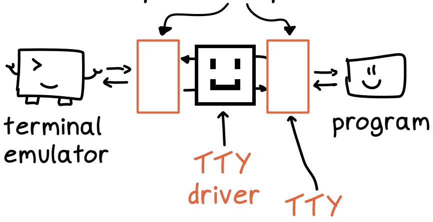
Option + click or Alt + click	place cursor
scroll wheel	scroll

meet the TTY driver

the TTY driver is the most obscure part of the system

You almost never need to think about it, but when I've wanted to do something weird (like put a terminal in a web browser) understanding the TTY driver is SO USEFUL

when you start your terminal emulator, it asks the OS to create a "pseudoterminal pair", which is a pair of special files



a "TTY" is the program's side of the pair

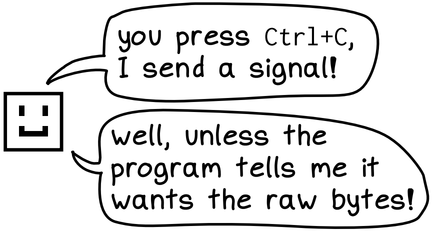
programs use it to:

→ communicate with the terminal emulator by reading/writing bytes

→ configure the TTY driver (more on the next page!)

Run `tty` to see the current TTY!

the TTY driver is why `Ctrl+C` does the same thing relatively consistently


you press `Ctrl+C`, I send a signal!
well, unless the program tells me it wants the raw bytes!

some things the TTY driver is in charge of

(you might think "these are all unrelated" and you'd be right)

80×20 storing the terminal window's size

⚡ sending a `SIGHUP` signal when you close your terminal

✎ a basic mode for entering text called "canonical mode"

STOP pausing the output and confusing you when you press `Ctrl+S`

→ tracking which process is in the "foreground" and sending what you type there

stty

your TTY driver has configuration

you can see how it's configured by running:

```
stty -a
```

for example it'll print out the current window size!

Ctrl+S

by default, pressing `Ctrl+S` will freeze your terminal (and `Ctrl+Q` will unfreeze)

I have never wanted this in my life, you can turn it off with `stty -ixon`

(fish turns it off by default)

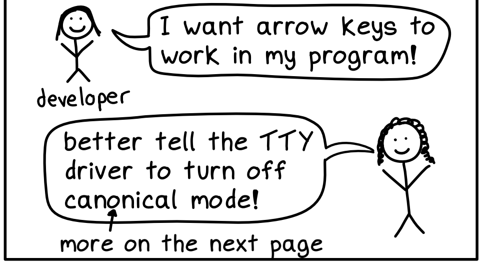
fun fact: changing Ctrl+C

technically you can use `stty` to set a different keyboard shortcut for `Ctrl+C`, like "u"

```
stty intr u
```

this is extremely chaotic and I can't imagine a reason that I would ever do this though

programs have to configure the TTY driver to get friendly features

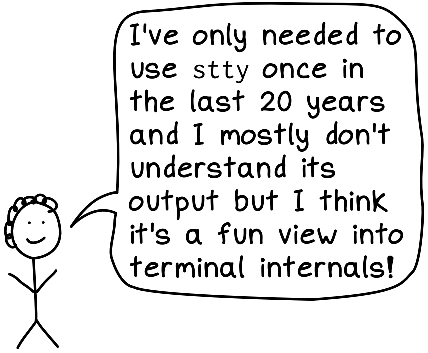

I want arrow keys to work in my program!
developer
better tell the TTY driver to turn off canonical mode!
more on the next page

the TTY driver's settings are called "termios settings"

for all the gnarly details:

```
man termios
```

but if you're writing a terminal program libraries like `readline` or `ncurses` will handle setting up the TTY driver


I've only needed to use stty once in the last 20 years and I mostly don't understand its output but I think it's a fun view into terminal internals!

canonical mode

We said earlier that every program has to implement text editing (on page 21)

This is not 100% true! The TTY driver technically has a very limited text editing system called "canonical mode" that hasn't changed since the 80s

what using canonical mode feels like

? ? ?
I pressed an arrow key and it just printed out ^[[D???
what's an arrow key?
TTY driver

how canonical mode works

① you type in text (helloo<Backspace><Enter>)

② the TTY driver lets you edit the text until you press <Enter>

③ the TTY driver sends the line of text to the program

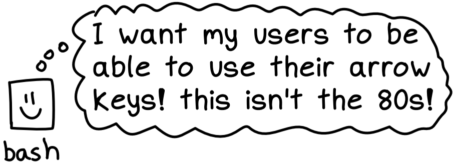
canonical mode is incredibly limited

The only ways it lets you edit text are:

→ backspace
→ `Ctrl+W` (delete word)
→ `Ctrl+U` (delete line)

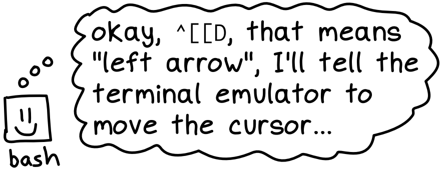
The good thing is those 3 things almost always work.

interactive programs almost never use canonical mode...


I want my users to be able to use their arrow keys! this isn't the 80s!
bash

You can try out canonical mode by running `cat` and typing.

... instead, programs receive bytes as soon as you type them


okay, ^[[D, that means "left arrow", I'll tell the terminal emulator to move the cursor...
bash

(usually by using a library like `readline`)

thanks for reading

The terminal is honestly a bit of a mess (some parts of it are stuck in the 80s with no clear way out!) but lots of people are building tools to make things better.

some things I think are cool:

- there are lots of people rebuilding classic command line tools, like I've been trying `eza` instead of `ls` (more at `https://wzrd.page/tools`)
- some terminal emulators have really amazing features, like I think the way iTerm2 allows you to set a minimum color contrast is incredibly useful
- and as a final plug: the `fish` shell really changed my life in the terminal. It isn't for everyone but I've used it every day for the last 10 years and I love it (more at `https://wzrd.page/ilovefish`)

maybe you'll build the next tool that makes the terminal better!

acknowledgements

Cover illustration: Vladimir Kašiković
Pairing: Marie Claire LeBlanc Flanagan
Technical review: Simon Tatham
Copy editing: Lesley Trites
and thanks to all 95 beta readers

♡ this?
more at
☆ wizardzines.com ☆