VT100 - where it all started

Definitions first

Term	Meaning
command line	a text interface
terminal	program presenting a text interface
console	the terminal is the machine
terminal emulator	software terminal — MobaXterm, URXVT, Kitty
shell	program that encapsulates the OS
command line shell	bash, zsh, fish
graphical shell	desktop environment — XFCE, KDE, Gnome

A more true terminal

If you own a Linux:
CTRL-ALT-F1 Also a virtual terminal
CTRL-ALT-F7 back to the graphical shell

Outline

1 The Operating System

2 A (Bad) Programming Language

3 A Shell

Chapter 1
The Operating System

Outline

1 The Operating System
- Launching programs
- File descriptors
- Exit codes
- Job control (2)

Bash orchestrates the OS

Bash features map to OS primitives

- launching programs
- job control
- file interaction
- exit codes
- job control (2)

Every process bash launches is built from three ingredients:

- **environment** — inherited key/value variables
- **program** — the executable being run
- **arguments** — `argv[]` passed to it

env, program, arguments

```
$ THREADS=8 samtools sort -@ 4 in.bam -o out.bam
  ^^^^^^^^^ ^^^^^^^ ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  env       program argv[]
```

- `VAR=1` — this instance
- `export VAR=1` — all child processes
- `VAR=1 cmd args` — scoped

env command

```
$ env                                  # show complete environment
$ env -u VAR cmd                       # run cmd without env var VAR
$ env -i HOME=$HOME bash -lc 'cmd' # run cmd in clean environment
$ env program                          # resolves the PATH
```

env command

```
$ env                                  # show complete environment
$ env -u VAR cmd                       # run cmd without env var VAR
$ env -i HOME=$HOME bash -lc 'cmd' # run cmd in clean environment
$ env program                          # resolves the PATH
```

env does what you can already accomplish with the shell

env command

```
$ env                                  # show complete environment
$ env -u VAR cmd                       # run cmd without env var VAR
$ env -i HOME=$HOME bash -lc 'cmd' # run cmd in clean environment
$ env program                          # resolves the PATH
```

env does what you can already accomplish with the shell

/usr/bin/env bash often considered a better shebang

```
#!/bin/bash
```

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

Job control: Every process PID

Every process has an ID, also PID

```
$ pgrep -u $USER python    # list matching PIDs
$ ps $PID                  # to show more info
```

xargs

```
pgrep bash | xargs ps o cmd=
find . -name '*.log' | xargs -n1 gzip
```

Parent - children process

processes spawns from another are **child processes**

```
ps --forest   # print child programs
pstree        # idem
exec program  # replace current proces no child program, very
    common for containers
```

File descriptors

stdout / stdin / stderr Three standard streams are opened for every process by default:

Stream	fd	Purpose
stdin	0	input the program reads
stdout	1	normal output
stderr	2	errors / diagnostics (kept separate!)

Any file descriptor is available

Syntax of file descriptors

```
cmd > out.txt          # stdout -> file (fd 1)
cmd 2> err.txt         # stderr -> file (fd 2)
cmd > out.txt 2>&1     # stderr follows stdout
cmd &> both.txt        # bash shortcut for both (also pipe: |&)
cmd < in.txt           # stdin <- file
<in.txt cmd            # reversed also works
cmd 2>&1 >/dev/null    # only errors
cmd >/dev/null 2>&1    # all into null(!)
```

Syntax of file descriptors

```
cmd > out.txt          # stdout -> file (fd 1)
cmd 2> err.txt         # stderr -> file (fd 2)
cmd > out.txt 2>&1     # stderr follows stdout
cmd &> both.txt        # bash shortcut for both (also pipe: |&)
cmd < in.txt           # stdin <- file
<in.txt cmd            # reversed also works
cmd 2>&1 >/dev/null    # only errors
cmd >/dev/null 2>&1    # all into null(!)
```

Order matters: redirections are applied left to right

Example: separating error streams

```
$ bwa mem ref.fa r1.fq r2.fq \
    > aligned.sam 2> bwa.log
```

Using /dev/

Path	Purpose
`/dev/stdout`	alias for this process's stdout
`/dev/urandom`	endless stream of random bytes
`/dev/null`	discards anything written to it
`/dev/fd`	directory of this process's open file descriptors

Example: Using /dev/ instead of a file

```
$ cat file.fa \
    | revseq /dev/stdin /dev/stdout 2>/dev/null \
    | cmd
```

Programs communicate with signals

A **signal** is an asynchronous interrupt

Signal	Number	Meaning
`SIGSTOP`	19	pause process, uncatchable
`SIGTSTP`	20	pause process, catchable (Ctrl-Z)
`SIGCONT`	18	resume a stopped process
`SIGPIPE`	13	reader end of a pipe closed
`SIGCHLD`	17	a child process stopped or terminated

Killing with signals

Signal	Number	Default meaning
`SIGHUP`	1	controlling terminal closed, ignore with `nohup`
`SIGTERM`	15	polite “please stop”
`SIGINT`	2	Ctrl-C interrupt
`SIGQUIT`	3	Ctrl-\ interrupt with core dump
`SIGKILL`	9	immediate death, uncatchable

How to kill

```
kill -9 1234             # SIGKILL to PID 1234
kill -TERM 1234          # polite ask, same as kill 1234
pkill -f "star --run"    # match against full cmdline
killall snakemake        # match by exact process name
xkill                    # kill by clicking
```


killall
-9
unresponsive
program

Signals can be trapped

A process can install a handler to run when a signal arrives — *except* `SIGKILL`/`SIGSTOP`, which the kernel enforces directly.

- `trap 'handler' SIGNAL` registers the handler
- Useful for cleanup: temp files, locks, subprocess teardown
- If untrapped, the shell's default action runs (usually terminate)

Example: cleanup on interrupt

```
tmpdir=$(mktemp -d)
cleanup() { rm -rf "$tmpdir"; echo "cleaned up"; }
trap cleanup EXIT INT TERM

process_bams "$tmpdir"    # if killed mid-run,
                           # tmpdir is still removed
```

`trap ... EXIT` fires on *any* exit path — normal, error, or signal — making it the closest thing bash has to a `finally` block.

Programs communicate back: exit codes

Every process returns a single integer exit code to its parent when it finishes.

- 0 = success
- Any nonzero = failure (the specific number is program-defined)
- Read it immediately after with $?

This one integer is the entire basis for bash's control flow.

Exit codes drive control flow

```
samtools index in.bam
if [ $? -eq 0 ]; then
    echo "indexed OK"
fi

samtools index in.bam && echo "OK" || echo "FAILED"
```

- `cmd1 && cmd2` — run `cmd2` only if `cmd1` succeeded
- `cmd1 || cmd2` — run `cmd2` only if `cmd1` failed

Example: if statements in a pipeline

```
if samtools quickcheck in.bam; then
    echo "valid BAM"
else
    echo "corrupt BAM, re-downloading" >&2
    fetch_bam.sh
fi

# short-circuit style, same effect:
samtools quickcheck in.bam || fetch_bam.sh
```

`if` never inspects a boolean — it runs a command and branches on *its exit code*.

Job control (2): background and foreground

Key/command	Effect
`cmd &`	start in background, keep shell free
`Ctrl-Z`	suspend current foreground job
`bg` / `fg`	resume in background / bring to foreground
`jobs`	list jobs owned by this shell
`nohup cmd &`	survive shell/terminal closing

Ctrl-Z does not stop the job — it suspends it (`SIGTSTP`); it keeps consuming memory until resumed or killed.

```
$ bwa mem ref.fa r1.fq r2.fq \
    > aligned.sam 2> bwa.log &
# PID=$?
$ tail -f bwa.log &          # progress watcher
$ wait $PID && done          # block to wait all background programs
```

```
$ bwa mem ref.fa r1.fq r2.fq \
    > aligned.sam 2> bwa.log &
# PID=$?
$ tail -f bwa.log &          # progress watcher
$ wait $PID && done          # block to wait all background programs
```

`wait` can also wait for all background tasks

e@11 engineering@eleven

Bash

OS interface, programming language, and shell

the operating system | a (bad) programming language | a shell

Welcome back: Bash session 2

Daoud's T-shirt:

```
#!/bin/bash eval "$(base64 -d <<< 'IyE...')"
```

Welcome back: Bash session 2

Daoud's T-shirt:

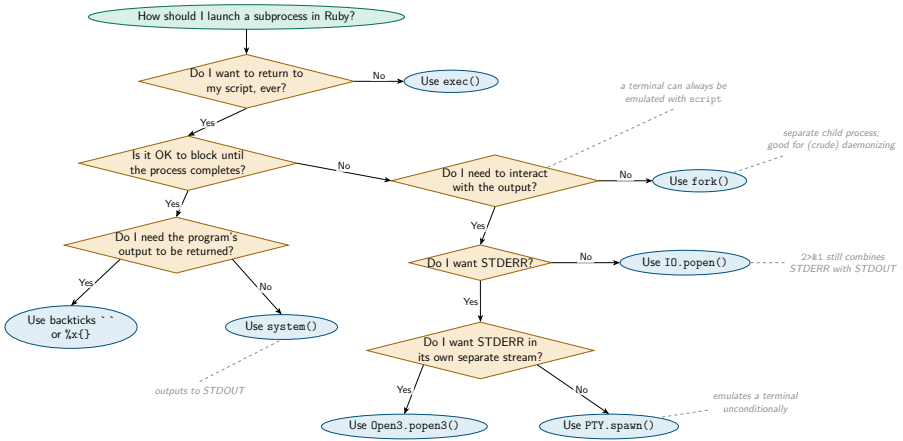
```
#!/bin/bash eval "$(base64 -d <<< 'IyE...')"
```

- Evaluate string as code: eval() is evil
- "$(...)" to have output in string
- <<< here string to present string as stdin

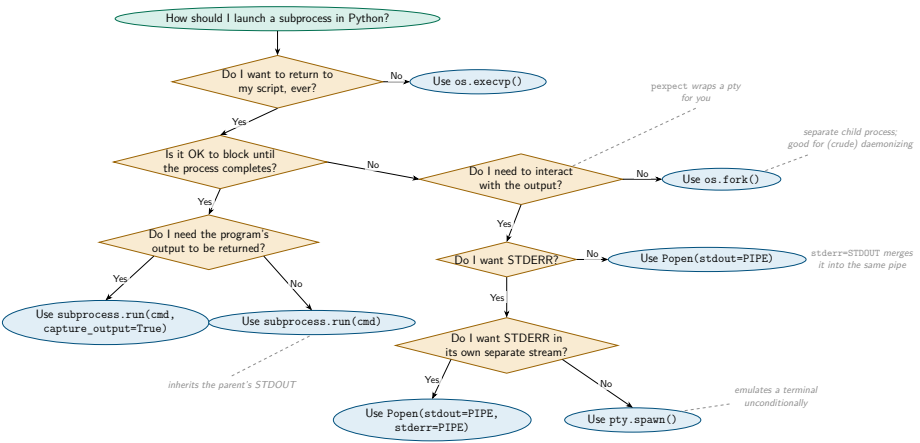
Recap from last time

- You run a command line **shell** in a **terminal**
- Processes consist of **environment**, **program**, **arguments**
- Processes have files (text streams), including **stdin** (0), **stdout** (1), and **stderr** (2)
- Only communication in is **signals** (next to files)
- Only communication out is **exit codes** (next to files)

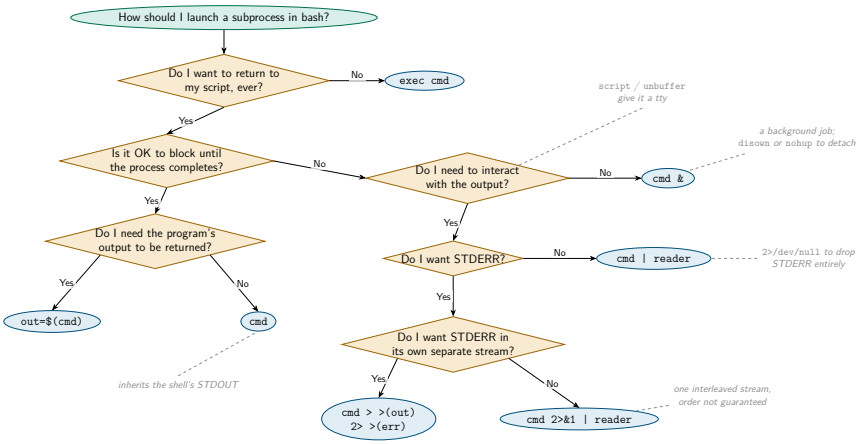
Every language re-invents this decision tree



Python re-invents it too



In bash every leaf is syntax



Bonus 2: Starting a subshell

A subshell is a "forked" process

Syntax	What you get
`( cmdA; cmdB; ) &`	group commands
`$( cmd )`	the output as string
`<( cmd )`	the output as file (to read from)
`>( cmd )`	the output as file (to write from)

Bonus 2: Starting a subshell

A subshell is a "forked" process

Syntax	What you get
`( cmdA; cmdB; ) &`	group commands
`$( cmd )`	the output as string
`<( cmd )`	the output as file (to read from)
`>( cmd )`	the output as file (to write from)

NB You can `2> >( cmd )` to send stderr off to a subshell

Chapter 2
A “Programming Language”

Outline

2 A (Bad) Programming Language
- Stringy typed
- Conditions
- Scoping rules
- Combining from multiple files
- Patterns
- Error handling
- Undefined variables
- Strict mode

Bash as a (bad) programming language

Bash was never designed as a general-purpose language

- No real types — everything is a string
- No real scope by default
- No import system, no error handling
- Lots of quirks

Everything is a string

```
x=5
y=10
echo $((x + y))     # 15  -- arithmetic context needed
echo $x+$y          # "5+10"  -- plain concatenation

n_samples="10"
if [[ $n_samples -gt 5 ]]; then
   echo "big cohort"
fi
```

no integer type: 5 and "5" are same until something *forces* numeric.

Everything is a string

```
x=5
y=10
echo $((x + y))     # 15  -- arithmetic context needed
echo $x+$y          # "5+10"  -- plain concatenation

n_samples="10"
if [[ $n_samples -gt 5 ]]; then
   echo "big cohort"
fi
```

no integer type: 5 and "5" are same until something *forces* numeric.
For True/False values I use 1/0, but “string”/“” is also common.

Everything is a string

```
x=5
y=10
echo $((x + y))     # 15  -- arithmetic context needed
echo $x+$y          # "5+10"  -- plain concatenation

n_samples="10"
if [[ $n_samples -gt 5 ]]; then
   echo "big cohort"
fi
```

no integer type: `5` and `"5"` are same until something *forces* numeric.
For True/False values I use 1/0, but “string”/“” is also common.
`DOCKER = bool(int(os.environ.get("DOCKER", 0)))`

The if statement is weird

```
[ "$a" = "$b" ]        # test command, needs spaces DO NOT USE!
[[ $a == $b* ]]        # bash builtin, supports globs/regex
(( a == b ))           # arithmetic context, C-like syntax
```

- `[` is a *program* (`/usr/bin/[`), not syntax — missing spaces around brackets break it
- `[[ ]]` is bash syntax safer, no word-splitting on unquoted vars

The if statement is weird

```
[ "$a" = "$b" ]        # test command, needs spaces DO NOT USE!
[[ $a == $b* ]]        # bash builtin, supports globs/regex
(( a == b ))           # arithmetic context, C-like syntax
```

- `[` is a *program* (`/usr/bin/[`), not syntax — missing spaces around brackets break it
- `[[ ]]` is bash syntax safer, no word-splitting on unquoted vars

- Three different "if conditions", three different rules

No scope by default

```
count=0
increment() { count=$((count + 1)); }  # leaks into caller!
increment
echo $count   # 1 -- global by default

safe() { local count=99; echo $count; }
```

All variables are global unless declared `local` inside a function — the opposite of what you want

There is no import — `source` is inline

```
# lib.sh
REF_GENOME=/data/ref/GRCh38.fa

# run.sh
source lib.sh            # literally splices lib.sh in here
echo $REF_GENOME         # works: same as copy-paste
```

No namespacing, no module boundary — `source` (or `.`) name collisions are silent.

There is no import — `source` is inline

```
# lib.sh
REF_GENOME=/data/ref/GRCh38.fa

# run.sh
source lib.sh            # literally splices lib.sh in here
echo $REF_GENOME         # works: same as copy-paste
```

No namespacing, no module boundary — `source` (or `.`) name collisions are silent. NB You can of course call a subprocess to isolate functionality

Globs, regex, and parameter expansion

Language	Syntax	Used for
glob	`*.bam`, `[0-9]`	filename expansion by the shell
regex	`[[ $f =~ re ]]`	POSIX pattern matching
param. expansion	`${var%.bam}`	string manipulation on a variable

Example: Globs, regex, and parameter expansion

```
*.bam                          # glob: filename expansion
[[ $f =~ ^chr[0-9]+$ ]]        # =~ : POSIX regex match
${var%.bam}                    # strip shortest suffix match
${var/_R1/_R2}                 # substitute pattern
${var:-default}                # use default if unset
```

Example: Globs, regex, and parameter expansion

```
*.bam                          # glob: filename expansion
[[ $f =~ ^chr[0-9]+$ ]]        # =~ : POSIX regex match
${var%.bam}                    # strip shortest suffix match
${var/_R1/_R2}                 # substitute pattern
${var:-default}                # use default if unset
```

Three *different* pattern languages coexist in bash: shell globs, `[[ =~ ]]` regex, and `$...` parameter expansion — don't mix them up.

There is no error handling

```
cd /data/nonexistent_dir
rm -rf *                    # ...still runs, in $PWD instead!
```

- No exceptions, no `try/catch`
- A failed command by default does not stop the script
- The script simply continues to the next line, exit code or not

Undefined variable — no problem

```
echo "Deleting $OUTDIR/tmp"
rm -rf "$OUTDIR"/tmp      # OUTDIR unset -> rm -rf /tmp !!
```

A typo'd or unset variable silently expands to an empty string — one of the most common causes of catastrophic bash accidents


nice code!
Approved!
Scary!
rm -rf "$STEAMROOT/"*
Issues
Insights

Bash strict mode

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

# -e : exit immediately on any command failure
# -u : error on unset variables
# -o pipefail : a pipeline fails if ANY stage fails,
#               not just the last one
# -x : bonus, prints every command ran
```

Bash strict mode

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

# -e : exit immediately on any command failure
# -u : error on unset variables
# -o pipefail : a pipeline fails if ANY stage fails,
#               not just the last one
# -x : bonus, prints every command ran
```

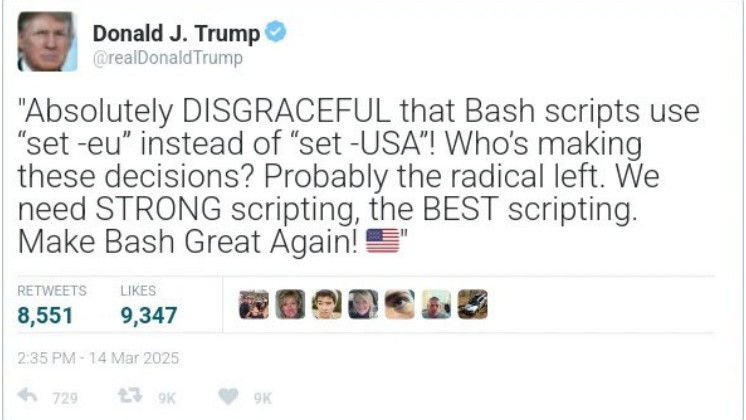
Put this at the top of every script for more safety.

Bash strict mode

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

# -e : exit immediately on any command failure
# -u : error on unset variables
# -o pipefail : a pipeline fails if ANY stage fails,
#               not just the last one
# -x : bonus, prints every command ran
```

Put this at the top of every script for more safety.
Snakemake already does this by default


Donald J. Trump
@realDonaldTrump
"Absolutely DISGRACEFUL that Bash scripts use “set -eu” instead of “set -USA”! Who’s making these decisions? Probably the radical left. We need STRONG scripting, the BEST scripting. Make Bash Great Again! "
RETWEETS
8,551
LIKES
9,347
2:35 PM - 14 Mar 2025
729
9K
9K

Chapter 3
A Shell

Outline

3 A Shell
- History
- Shortcuts
- Prompt customizing
- Other shells

History expansion

Expansion	Meaning
`!!`	previous command
`!$`	last argument of previous command (also `Alt-.`)
`!*`	all arguments of previous command

History manipulation: !!

```
$ apt install samtools
Permission denied
$ sudo !!
sudo apt install samtools
$ samtools sort in.bam -o out.bam
$ samtools index !$      # !$ = last argument of prev cmd
$ !samtools:p            # print, don't run, last "samtools" cmd
$ mv -t . /very/long/path/to/file
$ ls !$:h                # strip filename from last arg (dirname)
```


MAKE ME A SANDWICH
SUDO !!
quickmeme.com

Bash includes `readline` — shortcuts Emacs style

Bash's default line editing is emacs-style (readline)

Key	Effect
`Ctrl-_`	undo last edit to the line
`Ctrl-W`	delete word before cursor
`Ctrl-U`	delete to start of line
`Ctrl-T`	Transpose two characters
`Ctrl-R`	reverse search through history

Bash includes `readline` — shortcuts Emacs style

Bash's default line editing is emacs-style (readline)

Key	Effect
`Ctrl-_`	undo last edit to the line
`Ctrl-W`	delete word before cursor
`Ctrl-U`	delete to start of line
`Ctrl-T`	Transpose two characters
`Ctrl-R`	reverse search through history

Same bindings work in `psql`, `python` REPL, `gdb`

More history

Save more history, expanding your history

```
HISTSIZE=
HISTFILESIZE=
```

PS1: your prompt is a program

```
parse_git_branch() {
  git branch 2>/dev/null | sed -n 's/^\* \(.*\)/(\1)/p'
}
export PS1='\u@\h:\w$(parse_git_branch)\$ '
```

PS1: your prompt is a program

```
parse_git_branch() {
  git branch 2>/dev/null | sed -n 's/^\* \(.*\)/(\1)/p'
}
export PS1='\u@\h:\w$(parse_git_branch)\$ '
```

```
you@laptop:~/data/rnaseq(main)$ samtools sort in.bam
```

PS1: your prompt is a program

```
parse_git_branch() {
  git branch 2>/dev/null | sed -n 's/^\* \(.*\)/(\1)/p'
}
export PS1='\u@\h:\w$(parse_git_branch)\$ '
```

```
you@laptop:~/data/rnaseq(main)$ samtools sort in.bam
```

PS1 is re-evaluated *every prompt* — it is dynamic, not static

git-prompt comes pre-installed: `__git_ps1`

```
source /etc/bash_completion.d/git-prompt
export GIT_PS1_SHOWDIRTYSTATE=1
export PS1='\[\033[01;34m\]\w\[\033[00m\]$(__git_ps1 "(%s)")\$ '
```

git-prompt comes pre-installed: `__git_ps1`

```
source /etc/bash_completion.d/git-prompt
export GIT_PS1_SHOWDIRTYSTATE=1
export PS1='\[\033[01;34m\]\w\[\033[00m\]$(__git_ps1 "(%s)")\$ '
```

```
~/data/rnaseq(main *)$ git commit -am fix
```

git-prompt comes pre-installed: `__git_ps1`

```
source /etc/bash_completion.d/git-prompt
export GIT_PS1_SHOWDIRTYSTATE=1
export PS1='\[\033[01;34m\]\w\[\033[00m\]$(__git_ps1 "(%s)")\$ '
```

```
~/data/rnaseq(main *)$ git commit -am fix
```

`*` = unstaged changes, `+` = staged (from `GIT_PS1_SHOWDIRTYSTATE`)

What happens when a command is interpreted

	Step	Example
1	is it a keyword?	`if`, `for`, `[[` — syntax, not a program
2	is it an alias?	`ll` → `ls -l`
3	brace expansion	`s{1,2}.bam` → `s1.bam s2.bam`
4	tilde expansion	`~/data` → `/home/you/data`
5	parameter expansion	`$f`, `${f%.bam}`
6	command substitution	`$(samtools --version)`
7	arithmetic expansion	`$((n * 2))`
8	word splitting	split on `$IFS` — unquoted only
9	pathname expansion	`*.bam` → the matching files
10	command lookup	first word: function → builtin → `PATH`

What happens when a command is interpreted

	Step	Example
1	is it a keyword?	`if`, `for`, `[[` — syntax, not a program
2	is it an alias?	`ll` → `ls -l`
3	brace expansion	`s{1,2}.bam` → `s1.bam s2.bam`
4	tilde expansion	`~/data` → `/home/you/data`
5	parameter expansion	`$f`, `${f%.bam}`
6	command substitution	`$(samtools --version)`
7	arithmetic expansion	`$((n * 2))`
8	word splitting	split on `$IFS` — unquoted only
9	pathname expansion	`*.bam` → the matching files
10	command lookup	first word: function → builtin → `PATH`

5–7 are one pass, left to right; 8 and 9 come *after* — hence `"$var"`

Special variables

Variable	Meaning
`$0`	name of the script itself
`$1`, `$2`, ...	positional arguments
`$#`	number of arguments
`"$@"`	all arguments, each one still its own word
`"$*"`	all arguments glued into *one* word
`$?`	exit code of the last command
`$$`	PID of the current shell
`$!`	PID of the last background job

Special variables

Variable	Meaning
`$0`	name of the script itself
`$1`, `$2`, …	positional arguments
`$#`	number of arguments
`"$@"`	all arguments, each one still its own word
`"$*"`	all arguments glued into *one* word
`$?`	exit code of the last command
`$$`	PID of the current shell
`$!`	PID of the last background job

Always `"$@"` — `$*` destroys arguments containing spaces

Finding the folder of the script

```
# script's own directory, correct even when sourced
SCRIPT_DIR=$(cd -- "$(dirname -- "${BASH_SOURCE[0]}")" && pwd)

# bash's __name__ == "__main__"
if [[ "${BASH_SOURCE[0]}" == "$0" ]]; then
    main "$@"
fi
```

- $0 is what the *caller* typed — wrong when sourced
- ${BASH_SOURCE[0]} — this file
- ${BASH_SOURCE[-1]} — the main script that started it all

Better shells exist

Bash wins on ubiquity (it's everywhere, POSIX-compatible, scriptable)
But for convenience, alternatives exist:

- **zsh** + oh-my-zsh — bash-compatible, huge plugin ecosystem
- **fish** — not POSIX-compatible, but batteries included

Better shells exist

Bash wins on ubiquity (it's everywhere, POSIX-compatible, scriptable)
But for convenience, alternatives exist:

- **zsh** + oh-my-zsh — bash-compatible, huge plugin ecosystem
- **fish** — not POSIX-compatible, but batteries included

Rule of thumb: script in bash (portability), live in whatever shell you like.

zsh with oh-my-zsh

- Near-total bash compatibility — existing scripts (mostly) still run
- Autocompletion, spelling correction, extended glob qualifiers
- `oh-my-zsh`: community config framework — themes, git-aware prompt, plugins out of the box
- Default shell on macOS since Catalina (license issue)

zsh with oh-my-zsh

- Near-total bash compatibility — existing scripts (mostly) still run
- Autocompletion, spelling correction, extended glob qualifiers
- `oh-my-zsh`: community config framework — themes, git-aware prompt, plugins out of the box
- Default shell on macOS since Catalina (license issue)

Take my rice for a spin on service-hgn with `zsh`

fish: batteries included

- Autosuggestions and syntax highlighting by default, no plugins needed
- Sane scripting syntax (no `[[ ]]` vs `[ ]` confusion) — but not POSIX/bash compatible
- Great interactively; but don't write portable pipeline scripts

Takeaways

1. Bash syntax mirrors the **OS**: processes, fds, signals, exit codes
2. As a **language** it's stringly-typed and forgiving by default
3. As a **shell**, history and readline tricks
4. Move away from bash if you need more then simple process orchestration
5. Script in bash for portability; live in zsh/fish for comfort

References

- *GNU Bash Reference Manual*, Free Software Foundation.
- J. Collins, *Beginner's guide to the bash terminal*, `https://www.youtube.com/watch?v=oxuRxtrO2Ag`
- ShellCheck — static analysis for shell scripts, `shellcheck.net`.
- *oh-my-zsh* project, `ohmyz.sh`.
- *fish shell* documentation, `fishshell.com`.

Thank You

Thank You

Follow up presentions:

- Bash scripting part 3
- SystemD
- The deep command line belt
- Debugging programs and systems