Command Line Interface Guidelines

An open-source guide to help you write better command-line programs, taking traditional UNIX principles and updating them for the modern day.

Aanand Prasad
Engineer at Squarespace, co-creator of Docker Compose.
@aanandprasad

Ben Firshman
Co-creator Replicate, co-creator of Docker Compose.
@bfirsh

Carl Tashian
Developer Advocate at Smallstep, first engineer at Zipcar, co-founder Trove.
tashian.com @tashian

Eva Parish
Technical Writer at Squarespace, O’Reilly contributor.
evaparish.com @evpari

Design by Mark Hurrell. Thanks to Andreas Jansson for early contributions, and Andrew Reitz, Ashley Williams, Brendan Falk, Chester Ramey, Dj Walker-Morgan, Jacob Maine, James Coglan, Michael Dwan, and Steve Klabnik for reviewing drafts.

https://ghbtns.com/github-btn.html?user=cli-guidelines&repo=cli-guidelines&type=star&count=true&size=large

Join us on Discord if you want to discuss the guide or CLI design.

CLIG: Foreword

In the 1980s, if you wanted a personal computer to do something for you, you needed to know what to type when confronted with `C:\>` or `~$`. Help came in the form of thick, spiral-bound manuals. Error messages were opaque. There was no Stack Overflow to save you. But if you were lucky enough to have internet access, you could get help from Usenet—an early internet community filled with other people who were just as frustrated as you were. They could either help you solve your problem, or at least provide some moral support and camaraderie.

Forty years later, computers have become so much more accessible to everyone, often at the expense of low-level end user control. On many devices, there is no command-line access at all, in part because it goes against the corporate interests of walled gardens and app stores.

Most people today don't know what the command line is, much less why they would want to bother with it. As computing pioneer Alan Kay said in a 2017 interview, "Because people don't understand what computing is about, they think they have it in the iPhone, and that illusion is as bad as the illusion that 'Guitar Hero' is the same as a real guitar."

Kay's "real guitar" isn't the CLI—not exactly. He was talking about ways of programming computers that offer the power of the CLI and that transcend writing software in text files. There is a belief among Kay's disciples that we need to break out of a text-based local maxima that we've been living in for decades.

It's exciting to imagine a future where we program computers very differently. Even today, spreadsheets are by far the most popular programming language, and the no-code movement is taking off quickly as it attempts to replace some of the intense demand for talented programmers.

Contents

Yet with its creaky, decades-old constraints and inexplicable quirks, the command line is still the most *versatile* corner of the computer. It lets you pull back the curtain, see what's really going on, and creatively interact with the machine at a level of sophistication and depth that GUIs cannot afford. It's available on almost any laptop, for anyone who wants to learn it. It can be used interactively, or it can be automated. And, it doesn't change as fast as other parts of the system. There is creative value in its stability.

So, while we still have it, we should try to maximize its utility and accessibility.

A lot has changed about how we program computers since those early days. The command line of the past was *machine-first*: little more than a REPL on top of a scripting platform. But as general-purpose interpreted languages have flourished, the role of the shell script has shrunk. Today's command line is *human-first*: a text-based UI that affords access to all kinds of tools, systems and platforms. In the past, the editor was inside the terminal—today, the terminal is just as often a feature of the editor. And there's been a proliferation of `git`-like multi-tool commands. Commands within commands, and high-level commands that perform entire workflows rather than atomic functions.

Inspired by traditional UNIX philosophy, driven by an interest in encouraging a more delightful and accessible CLI environment, and guided by our experiences as programmers, we decided it was time to revisit the best practices and design principles for building command-line programs.

Long live the command line!

CLIG: Introduction

This document covers both high-level design philosophy, and concrete guidelines. It's heavier on the guidelines because our philosophy as practitioners is not to philosophize too much. We believe in learning by example, so we've provided plenty of those.

This guide doesn't cover full-screen terminal programs like emacs and vim. Full-screen programs are niche projects—very few of us will ever be in the position to design one.

This guide is also agnostic about programming languages and tooling in general.

Who is this guide for?

- If you are creating a CLI program and you are looking for principles and concrete best practices for its UI design, this guide is for you.
- If you are a professional "CLI UI designer," that's amazing—we'd love to learn from you.
- If you'd like to avoid obvious missteps of the variety that go against 40 years of CLI design conventions, this guide is for you.
- If you want to delight people with your program's good design and helpful help, this guide is definitely for you.
- If you are creating a GUI program, this guide is not for you—though you may learn some GUI anti-patterns if you

decide to read it anyway. (Do GUI programmers even read, or do they just look at things?)

Contents

If you are designing an immersive, full-screen CLI port of Minecraft, this guide isn't for you. (But we can't wait to see it!)

CLIG: Philosophy

These are what we consider to be the fundamental principles of good CLI design.

CLIG: Human-first design

Traditionally, UNIX commands were written under the assumption they were going to be used primarily by other programs. They had more in common with functions in a programming language than with graphical applications.

Today, even though many CLI programs are used primarily (or even exclusively) by humans, a lot of their interaction design still carries the baggage of the past. It's time to shed some of this baggage: if a command is going to be used primarily by humans, it should be designed for humans first.

Contents

CLIG: Simple parts that work together

A core tenet of the original UNIX philosophy is the idea that small, simple programs with clean interfaces can be combined to build larger systems. Rather than stuff more and more features into those programs, you make programs that are modular enough to be recombined as needed.

In the old days, pipes and shell scripts played a crucial role in the process of composing programs together. Their role might have diminished with the rise of general-purpose interpreted languages, but they certainly haven't gone away. What's more, large-scale automation—in the form of CI/CD, orchestration and configuration management—has flourished. Making programs composable is just as important as ever.

Fortunately, the long-established conventions of the UNIX environment, designed for this exact purpose, still help us today. Standard in/out/err, signals, exit codes and other mechanisms ensure that different programs click together nicely. Plain, line-based text is easy to pipe between commands. JSON, a much more recent invention, affords us more structure when we need it, and lets us more easily integrate command-line tools with the web.

Whatever software you're building, you can be absolutely certain that people will use it in ways you didn't anticipate. Your software *will* become a part in a larger system—your only choice is over whether it will be a well-behaved part.

Most importantly, designing for composability does not need to be at odds with designing for humans first. Much of the advice in this document is about how to achieve both.

Contents

CLIG: Consistency across programs

The terminal's conventions are hardwired into our fingers. We had to pay an upfront cost by learning about command line syntax, flags, environment variables and so on, but it pays off in long-term efficiency… as long as programs are consistent.

Where possible, a CLI should follow patterns that already exist. That's what makes CLIs intuitive and guessable; that's what makes users efficient.

That being said, sometimes consistency conflicts with ease of use. For example, many long-established UNIX commands don't output much information by default, which can cause confusion or worry for people less familiar with the command line.

When following convention would compromise a program's usability, it might be time to break with it—but such a decision should be made with care.

Contents

CLIG: Saying (just) enough

The terminal is a world of pure information. You could make an argument that information is the interface—and that, just like with any interface, there’s often too much or too little of it.

A command is saying too little when it hangs for several minutes and the user starts to wonder if it’s broken. A command is saying too much when it dumps pages and pages of debugging output, drowning what’s truly important in an ocean of loose detritus. The end result is the same: a lack of clarity, leaving the user confused and irritated.

It can be very difficult to get this balance right, but it’s absolutely crucial if software is to empower and serve its users.

CLIG: Ease of discovery

When it comes to making functionality discoverable, GUIs have the upper hand. Everything you can do is laid out in front of you on the screen, so you can find what you need without having to learn anything, and perhaps even discover things you didn't know were possible.

It is assumed that command-line interfaces are the opposite of this—that you have to remember how to do everything. The original Macintosh Human Interface Guidelines, published in 1992, recommend "See-and-point (instead of remember-and-type)," as if you could only choose one or the other.

These things needn't be mutually exclusive. The efficiency of using the command-line comes from remembering commands, but there's no reason the commands can't help you learn and remember.

Discoverable CLIs have comprehensive help texts, provide lots of examples, suggest what command to run next, suggest what to do when there is an error. There are lots of ideas that can be stolen from GUIs to make CLIs easier to learn and use, even for power users.

Citation: The Design of Everyday Things (Don Norman), Macintosh Human Interface Guidelines

GUI design, particularly in its early days, made heavy use of *metaphor*: desktops, files, folders, recycle bins. It made a lot of sense, because computers were still trying to bootstrap themselves into legitimacy. The ease of implementation of metaphors was one of the huge advantages GUIs wielded over CLIs. Ironically, though, the CLI has embodied an accidental metaphor all along: it's a conversation.

Beyond the most utterly simple commands, running a program usually involves more than one invocation. Usually, this is because it's hard to get it right the first time: the user types a command, gets an error, changes the command, gets a different error, and so on, until it works. This mode of learning through repeated failure is like a conversation the user is having with the program.

Trial-and-error isn't the only type of conversational interaction, though. There are others:

- Running one command to set up a tool and then learning what commands to run to actually start using it.
- Running several commands to set up an operation, and then a final command to run it (e.g. multiple `git add`s, followed by a `git commit`).
- Exploring a system—for example, doing a lot of `cd` and `ls` to get a sense of a directory structure, or `git log` and `git show` to explore the history of a file.
- Doing a dry-run of a complex operation before running it for real.

Acknowledging the conversational nature of command-line interaction means you can bring relevant techniques to bear on its design. You can suggest possible corrections when user input is invalid, you can make the intermediate state clear when the user is going through a multi-step process, you can confirm for them that everything looks good before they do something scary.

The user is conversing with your software, whether you intended it or not. At worst, it's a hostile conversation which makes them feel stupid and resentful. At best, it's a pleasant exchange that speeds them on their way with newfound knowledge and a feeling of achievement.

Contents

Further reading: The Anti-Mac User Interface (Don Gentner and Jakob Nielsen)

CLIG: Robustness

Robustness is both an objective and a subjective property. Software should *be* robust, of course: unexpected input should be handled gracefully, operations should be idempotent where possible, and so on. But it should also *feel* robust.

You want your software to feel like it isn't going to fall apart. You want it to feel immediate and responsive, as if it were a big mechanical machine, not a flimsy plastic "soft switch."

Subjective robustness requires attention to detail and thinking hard about what can go wrong. It's lots of little things: keeping the user informed about what's happening, explaining what common errors mean, not printing scary-looking stack traces.

As a general rule, robustness can also come from keeping it simple. Lots of special cases and complex code tend to make a program fragile.

CLIG: Empathy

Command-line tools are a programmer's creative toolkit, so they should be enjoyable to use. This doesn't mean turning them into a video game, or using lots of emoji (though there's nothing inherently wrong with emoji 😉). It means giving the user the feeling that you are on their side, that you want them to succeed, that you have thought carefully about their problems and how to solve them.

There's no list of actions you can take that will ensure they feel this way, although we hope that following our advice will take you some of the way there. Delighting the user means *exceeding their expectations* at every turn, and that starts with empathy.

Contents

CLIG: Chaos

The world of the terminal is a mess. Inconsistencies are everywhere, slowing us down and making us second-guess ourselves.

Yet it’s undeniable that this chaos has been a source of power. The terminal, like the UNIX-descended computing environment in general, places very few constraints on what you can build. In that space, all manner of invention has bloomed.

It’s ironic that this document implores you to follow existing patterns, right alongside advice that contradicts decades of command-line tradition. We’re just as guilty of breaking the rules as anyone.

The time might come when you, too, have to break the rules. Do so with intention and clarity of purpose.

“Abandon a standard when it is demonstrably harmful to productivity or user satisfaction.” — Jef Raskin, The Humane Interface

Contents

CLIG: Guidelines

This is a collection of specific things you can do to make your command-line program better.

The first section contains the essential things you need to follow. Get these wrong, and your program will be either hard to use or a bad CLI citizen.

The rest are nice-to-haves. If you have the time and energy to add these things, your program will be a lot better than the average program.

The idea is that, if you don't want to think too hard about the design of your program, you don't have to: just follow these rules and your program will probably be good. On the other hand, if you've thought about it and determined that a rule is wrong for your program, that's fine. (There's no central authority that will reject your program for not following arbitrary rules.)

Also—these rules aren't written in stone. If you disagree with a general rule for good reason, we hope you'll propose a change.

CLIG: The Basics

There are a few basic rules you need to follow. Get these wrong, and your program will be either very hard to use, or flat-out broken.

Contents

Use a command-line argument parsing library where you can. Either your language's built-in one, or a good third-party one. They will normally handle arguments, flag parsing, help text, and even spelling suggestions in a sensible way.

Here are some that we like:

- Go: Cobra, cli
- Node: oclif
- Python: Click, Typer
- Ruby: TTY

Return zero exit code on success, non-zero on failure. Exit codes are how scripts determine whether a program succeeded or failed, so you should report this correctly. Map the non-zero exit codes to the most important failure modes.

Send output to `stdout`. The primary output for your command should go to `stdout`. Anything that is machine readable should also go to `stdout`—this is where piping sends things by default.

Send messaging to `stderr`. Log messages, errors, and so on should all be sent to `stderr`. This means that when commands are piped together, these messages are displayed to the user and not fed into the next command.

Contents

CLIG: Help

Display help text when passed no options, the `-h` flag, or the `--help` flag.

Display a concise help text by default. If you can, display help by default when `myapp` or `myapp subcommand` is run. Unless your program is very simple and does something obvious by default (e.g. `ls`), or your program reads input interactively (e.g. `cat`).

The concise help text should only include:

- A description of what your program does.
- One or two example invocations.
- Descriptions of flags, unless there are lots of them.
- An instruction to pass the `--help` flag for more information.

`jq` does this well. When you type `jq`, it displays an introductory description and an example, then prompts you to pass `jq --help` for the full listing of flags:

```
$ jq
jq - commandline JSON processor [version 1.6]

Usage:    jq [options] <jq filter> [file...]
    jq [options] --args <jq filter> [strings...]
    jq [options] --jsonargs <jq filter> [JSON_TEXTS...]

jq is a tool for processing JSON inputs, applying the given filter to
```

```
its JSON text inputs and producing the filter's results as JSON on
standard output.

The simplest filter is ., which copies jq's input to its output
unmodified (except for formatting, but note that IEEE754 is used
for number representation internally, with all that that implies).

For more advanced filters see the jq(1) manpage ("man jq")
and/or https://stedolan.github.io/jq

Example:

    $ echo '{"foo": 0}' | jq .
    {
        "foo": 0
    }

For a listing of options, use jq --help.
```

Contents

Show full help when `-h` and `--help` is passed. All of these should show help:

```
$ myapp
$ myapp --help
$ myapp -h
```

Ignore any other flags and arguments that are passed—you should be able to add `-h` to the end of anything and it should show help. Don't overload `-h`.

If your program is `git`-like, the following should also offer help:

Contents

```
$ myapp help
$ myapp help subcommand
$ myapp subcommand --help
$ myapp subcommand -h
```

Provide a support path for feedback and issues. A website or GitHub link in the top-level help text is common.

In help text, link to the web version of the documentation. If you have a specific page or anchor for a subcommand, link directly to that. This is particularly useful if there is more detailed documentation on the web, or further reading that might explain the behavior of something.

Lead with examples. Users tend to use examples over other forms of documentation, so show them first in the help page, particularly the common complex uses. If it helps explain what it's doing and it isn't too long, show the actual output too.

You can tell a story with a series of examples, building your way toward complex uses.

If you've got loads of examples, put them somewhere else, in a cheat sheet command or a web page. It's useful to have exhaustive, advanced examples, but you don't want to make your help text really long.

For more complex use cases, e.g. when integrating with another tool, it might be appropriate to write a fully-fledged tutorial.

Don't bother with man pages. We believe that if you're following these guidelines for help and documentation, you won't need man pages. Not enough people use man pages, and they don't work on Windows. If your CLI framework and package manager make it easy to output man pages, go for it, but otherwise your time is best spent improving web docs and built-in help text.

Citation: 12 Factor CLI Apps.

If your help text is long, pipe it through a pager. This is one useful thing that `man` does for you. See the advice in the "Output" section below.

Display the most common flags and commands at the start of the help text. It's fine to have lots of flags, but if you've got some really common ones, display them first. For example, the Git command displays the commands for getting started and the most commonly used subcommands first:

```
$ git
usage: git [--version] [--help] [-C <path>] [-c <name>=<value>]
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p | --paginate | -P | --no-pager] [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           <command> [<args>]

These are common Git commands used in various situations:

start a working area (see also: git help tutorial)
   clone      Clone a repository into a new directory
```

```
   clone      Clone a repository into a new directory
   init       Create an empty Git repository or reinitialize an existing one

work on the current change (see also: git help everyday)
   add        Add file contents to the index
   mv         Move or rename a file, a directory, or a symlink
   reset      Reset current HEAD to the specified state
   rm         Remove files from the working tree and from the index

examine the history and state (see also: git help revisions)
   bisect     Use binary search to find the commit that introduced a bug
   grep       Print lines matching a pattern
   log        Show commit logs
   show       Show various types of objects
   status     Show the working tree status
…
```

Contents

Use formatting in your help text. Bold headings make it much easier to scan. But, try to do it in a terminal-independent way so that your users aren't staring down a wall of escape characters.

```
$ heroku apps --help
list your apps

USAGE
  $ heroku apps

OPTIONS

  -A, --all          include apps in all teams
  -p, --personal     list apps in personal account when a default team is set
```

```
  -p, --personal     list apps in personal account when a default team is set
  -s, --space=space  filter by space
  -t, --team=team    team to use
  --json             output in json format

EXAMPLES
  $ heroku apps
  === My Apps
  example
  example2

  === Collaborated Apps
  theirapp   other@owner.name

COMMANDS
  apps:create     creates a new app
  apps:destroy    permanently destroy an app
  apps:errors     view app errors
  apps:favorites  list favorited apps
  apps:info       show detailed app information
  apps:join       add yourself to a team app
  apps:leave      remove yourself from a team app
  apps:lock       prevent team members from joining an app
  apps:open       open the app in a web browser
  apps:rename     rename an app
  apps:stacks     show the list of available stacks
  apps:transfer   transfer applications to another user or team
  apps:unlock     unlock an app so any team member can join
```


Note: When `heroku apps --help` is piped through a pager, the command emits no escape characters.

If the user did something wrong and you can guess what they meant, suggest it. For example, `brew update jq` tells you that you should run `brew upgrade jq`.

You can ask if they want to run the suggested command, but don't force it on them. For example:

Contents

```
$ heroku pss
 ›   Warning: pss is not a heroku command.
Did you mean ps? [y/n]:
```

Rather than suggesting the corrected syntax, you might be tempted to just run it for them, as if they'd typed it right in the first place. Sometimes this is the right thing to do, but not always.

Firstly, invalid input doesn't necessarily imply a simple typo—it can often mean the user has made a logical mistake, or misused a shell variable. Assuming what they meant can be dangerous, especially if the resulting action modifies state.

Secondly, be aware that if you change what the user typed, they won't learn the correct syntax. In effect, you're ruling that the way they typed it is valid and correct, and you're committing to supporting that indefinitely. Be intentional in making that decision, and document both syntaxes.

Further reading: "Do What I Mean"

If your command is expecting to have something piped to it and `stdin` is an interactive terminal, display help immediately and quit. This means it doesn't just hang, like `cat`. Alternatively, you could print a log message to `stderr`.

Contents

CLIG: Output

Human-readable output is paramount. Humans come first, machines second. The most simple and straightforward heuristic for whether a particular output stream (`stdout` or `stderr`) is being read by a human is *whether or not it's a TTY*. Whatever language you're using, it will have a utility or library for doing this (e.g. Python, Node, Go).

Further reading on what a TTY is.

Have machine-readable output where it does not impact usability. Streams of text is the universal interface in UNIX. Programs typically output lines of text, and programs typically expect lines of text as input, therefore you can compose multiple programs together. This is normally done to make it possible to write scripts, but it can also help the usability for humans using programs. For example, a user should be able to pipe output to `grep` and it should do what they expect.

"Expect the output of every program to become the input to another, as yet unknown, program." — Doug McIlroy

If human-readable output breaks machine-readable output, use `--plain` to display output in plain, tabular text format for integration with tools like `grep` or `awk`. In some cases, you might need to output information in a different way to make it human-readable.

For example, if you are displaying a line-based table, you might choose to split a cell into multiple lines, fitting in more information while keeping it within the width of the screen. This breaks the expected behavior of there being one piece of data per line, so you should provide a `--plain` flag for scripts, which disables all such manipulation and outputs one record per line.

Display output as formatted JSON if `--json` is passed. JSON allows for more structure than plain text, so it makes it much easier to output and handle complex data structures. `jq` is a common tool for working with JSON on the command-line, and there is now a whole ecosystem of tools that output and manipulate JSON.

It is also widely used on the web, so by using JSON as the input and output of programs, you can pipe directly to and from web services using `curl`.

Display output on success, but keep it brief. Traditionally, when nothing is wrong, UNIX commands display no output to the user. This makes sense when they're being used in scripts, but can make commands appear to be hanging or broken when used by humans. For example, `cp` will not print anything, even if it takes a long time.

It's rare that printing nothing at all is the best default behavior, but it's usually best to err on the side of less.

For instances where you do want no output (for example, when used in shell scripts), to avoid clumsy redirection of `stderr` to `/dev/null`, you can provide a `-q` option to suppress all non-essential output.

If you change state, tell the user. When a command changes the state of a system, it's especially valuable to explain what has just happened, so the user can model the state of the system in their head—particularly if the result doesn't directly map to what the user requested.

For example, `git push` tells you exactly what it is doing, and what the new state of the remote branch is:


```
$ git push
Enumerating objects: 18, done.
Counting objects: 100% (18/18), done.
Delta compression using up to 8 threads
Compressing objects: 100% (10/10), done.
Writing objects: 100% (10/10), 2.09 KiB | 2.09 MiB/s, done.
Total 10 (delta 8), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (8/8), completed with 8 local objects.
To github.com:replicate/replicate.git
 + 6c22c90...a2a5217 bfirsh/fix-delete -> bfirsh/fix-delete
```

Make it easy to see the current state of the system. If your program does a lot of complex state changes and it is not immediately visible in the filesystem, make sure you make this easy to view.

For example, `git status` tells you as much information as possible about the current state of your Git repository, and some hints at how to modify the state:

```
$ git status
On branch bfirsh/fix-delete
Your branch is up to date with 'origin/bfirsh/fix-delete'.

Changes not staged for commit:
```

```
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   cli/pkg/cli/rm.go

no changes added to commit (use "git add" and/or "git commit -a")
```

Contents

Suggest commands the user should run. When several commands form a workflow, suggesting to the user commands they can run next helps them learn how to use your program and discover new functionality. For example, in the git status output above, it suggests commands you can run to modify the state you are viewing.

Actions crossing the boundary of the program's internal world should usually be explicit. This includes things like:

- Reading or writing files that the user didn't explicitly pass as arguments (unless those files are storing internal program state, such as a cache).
- Talking to a remote server, e.g. to download a file.

Increase information density—with ASCII art! For example, ls shows permissions in a scannable way. When you first see it, you can ignore most of the information. Then, as you learn how it works, you pick out more patterns over time.

```
-rw-r--r-- 1 root root     68 Aug 22 23:20 resolv.conf
lrwxrwxrwx 1 root root     13 Mar 14 20:24 rmt -> /usr/sbin/rmt
drwxr-xr-x 4 root root   4.0K Jul 20 14:51 security
drwxr-xr-x 2 root root   4.0K Jul 20 14:53 selinux
-rw-r----- 1 root shadow  501 Jul 20 14:44 shadow
-rw-r--r-- 1 root root    116 Jul 20 14:43 shells
```

```
drwxr-xr-x 2 root root   4.0K Jul 20 14:57 skel
-rw-r--r-- 1 root root      0 Jul 20 14:43 subgid
-rw-r--r-- 1 root root      0 Jul 20 14:43 subuid
```

Contents

Use color with intention. For example, you might want to highlight some text so the user notices it, or use red to indicate an error. Don't overuse it—if everything is a different color, then the color means nothing and only makes it harder to read.

Disable color if your program is not in a terminal or the user requested it. These things should disable colors:

- `stdout` or `stderr` is not an interactive terminal (a TTY). It's best to individually check—if you're piping `stdout` to another program, it's still useful to get colors on `stderr`.
- The `NO_COLOR` environment variable is set.
- The `TERM` environment variable has the value `dumb`.
- The user passes the option `--no-color`.
- You may also want to add a `MYAPP_NO_COLOR` environment variable in case users want to disable color specifically for your program.

Further reading: no-color.org, 12 Factor CLI Apps

If `stdout` is not an interactive terminal, don't display any animations. This will stop progress bars turning into Christmas trees in CI log output.

Use symbols and emoji where it makes things clearer. Pictures can be better than words if you need to make several things distinct, catch the user's attention, or just add a bit of character. Be careful, though—it can be easy to overdo it and make your program look cluttered or feel like a toy.

For example, yubikey-agent uses emoji to add structure to the output so it isn't just a wall of text, and a ❌ to draw your attention to an important piece of information:


```
$ yubikey-agent -setup
🔐 The PIN is up to 8 numbers, letters, or symbols. Not just numbers!
❌ The key will be lost if the PIN and PUK are locked after 3 incorrect tries.

Choose a new PIN/PUK:
Repeat the PIN/PUK:

🧪 Retriculating splines …

✔ Done! This YubiKey is secured and ready to go.
🤏 When the YubiKey blinks, touch it to authorize the login.

🔑 Here's your new shiny SSH public key:
ecdsa-sha2-nistp256 AAAAE2VjZHNhLXNoYTItbmlzdHAyNTYAAAAIbmlzdHAyNTYAAABBBCEJ/
UwlHnUFXgENO3ifPZd8zoSKMxESxxot4tMgvfXjmRp5G3BGrAnonncE7Aj11pn3SSYgEcrrn2sMyL(

💭 Remember: everything breaks, have a backup plan for when this YubiKey does.
```

By default, don't output information that's only understandable by the creators of the software. If a piece of output serves only to help you (the developer) understand what your software is doing, it almost certainly shouldn't be displayed to normal users by default—only in verbose mode.

Invite usability feedback from outsiders and people who are new to your project. They'll help you see important issues that you are too close to the code to notice.

Don't treat `stderr` like a log file, at least not by default. Don't print log level labels (`ERR`, `WARN`, etc.) or extraneous contextual information, unless in verbose mode.

Contents

Use a pager (e.g. `less`) if you are outputting a lot of text. For example, `git diff` does this by default. Using a pager can be error-prone, so be careful with your implementation such that you don't make the experience worse for the user. You shouldn't use a pager if `stdin` or `stdout` is not an interactive terminal.

A good sensible set of options to use for `less` is `less -FIRX`. This does not page if the content fills one screen, ignores case when you search, enables color and formatting, and leaves the contents on the screen when `less` quits.

There might be libraries in your language that are more robust than piping to `less`. For example, pypager in Python.

CLIG: Errors

One of the most common reasons to consult documentation is to fix errors. If you can make errors into documentation, then this will save the user loads of time.

Catch errors and rewrite them for humans. If you're expecting an error to happen, catch it and rewrite the error message to be useful. Think of it like a conversation, where the user has done something wrong and the program is guiding them in the right direction. Example: “Can’t write to file.txt. You might need to make it writable by running ‘chmod +w file.txt’.”

Contents

Signal-to-noise ratio is crucial. The more irrelevant output you produce, the longer it’s going to take the user to figure out what they did wrong. If your program produces multiple errors of the same type, consider grouping them under a single explanatory header instead of printing many similar-looking lines.

Consider where the user will look first. Put the most important information at the end of the output. The eye will be drawn to red text, so use it intentionally and sparingly.

If there is an unexpected or unexplainable error, provide debug and traceback information, and instructions on how to submit a bug. That said, don’t forget about the signal-to-noise ratio: you don’t want to overwhelm the user with information they don’t understand. Consider writing the debug log to a file instead of printing it to the terminal.

Make it effortless to submit bug reports. One nice thing you can do is provide a URL and have it pre-populate as much information as possible.

A note on terminology:

> *Arguments*, or *args*, are positional parameters to a command. For example, the file paths you provide to `cp` are args. The order of args is often important: `cp foo bar` means something different from `cp bar foo`.
> *Flags* are named parameters, denoted with either a hyphen and a single-letter name (`-r`) or a double hyphen and a multiple-letter name (`--recursive`). They may or may not also include a user-specified value (`--file foo.txt`, or `--file=foo.txt`). The order of flags, generally speaking, does not affect program semantics.

Prefer flags to args. It's a bit more typing, but it makes it much clearer what is going on. It also makes it easier to make changes to how you accept input in the future. Sometimes when using args, it's impossible to add new input without breaking existing behavior or creating ambiguity.

Citation: 12 Factor CLI Apps.

Have full-length versions of all flags. For example, have both `-h` and `--help`. Having the full version is useful in scripts where you want to be verbose and descriptive, and you don't have to look up the meaning of flags everywhere.

Citation: GNU Coding Standards.

Only use one-letter flags for commonly used flags, particularly at the top-level when using subcommands. That way you don't "pollute" your namespace of short flags, forcing you to use convoluted letters and cases for flags you add in the future.

Multiple arguments are fine for simple actions against multiple files. For example, `rm file1.txt file2.txt` file3.txt . This also makes it work with globbing: rm *.txt .

Contents

If you've got two or more arguments for different things, you're probably doing something wrong. The exception is a common, primary action, where the brevity is worth memorizing. For example, cp <source> <destination> .

Citation: 12 Factor CLI Apps.

Use standard names for flags, if there is a standard. If another commonly used command uses a flag name, it's best to follow that existing pattern. That way, a user doesn't have to remember two different options (and which command it applies to), and users can even guess an option without having to look at the help text.

Here's a list of commonly used options:

- -a , --all : All. For example, ps , fetchmail .
- -d , --debug : Show debugging output.
- -f , --force : Force. For example, rm -f will force the removal of files, even if it thinks it does not have permission to do it. This is also useful for commands which are doing something destructive that usually require user confirmation, but you want to force it to do that destructive action in a script.
- --json : Display JSON output. See the output section.
- -h , --help : Help. This should only mean help. See the help section.
- --no-input : See the interactivity section.
- -o , --output : Output file. For example, sort , gcc .
- -p , --port : Port. For example, psql , ssh .
- -q , --quiet : Quiet. Display less output. This is particularly useful when displaying output for humans that you might want to hide when running in a script.
- -u , --user : User. For example, ps , ssh .
- --version : Version.
- -v : This can often mean either verbose or version. You might want to use -d for verbose and this for version, or

for nothing to avoid confusion.

Contents

Make the default the right thing for most users. Making things configurable is good, but most users are not going to find the right flag and remember to use it all the time (or alias it). If it’s not the default, you’re making the experience worse for most of your users.

For example, `ls` has terse default output to optimize for scripts and other historical reasons, but if it were designed today, it would probably default to `ls -lhFGT`.

Prompt for user input. If a user doesn’t pass an argument or flag, prompt for it. (See also: Interactivity)

Never *require* a prompt. Always provide a way of passing input with flags or arguments. If `stdin` is not an interactive terminal, skip prompting and just require those flags/args.

Confirm before doing anything dangerous. A common convention is to prompt for the user to type `y` or `yes` if running interactively, or requiring them to pass `-f` or `--force` otherwise.

“Dangerous” is a subjective term, and there are differing levels of danger:

- **Mild:** A small, local change such as deleting a file. You might want to prompt for confirmation, you might not. For example, if the user is explicitly running a command called something like “delete,” you probably don’t need to ask.
- **Moderate:** A bigger local change like deleting a directory, a remote change like deleting a resource of some kind, or a complex bulk modification that can’t be easily undone. You usually want to prompt for confirmation here. Consider giving the user a way to “dry run” the operation so they can see what’ll happen before they commit to it.

Contents

Severe: Deleting something complex, like an entire remote application or server. You don't just want to prompt for confirmation here—you want to make it hard to confirm by accident. Consider asking them to type something non-trivial such as the name of the thing they're deleting. Let them alternatively pass a flag such as `--confirm="name-of-thing"`, so it's still scriptable.

Consider whether there are non-obvious ways to accidentally destroy things. For example, imagine a situation where changing a number in a configuration file from 10 to 1 means that 9 things will be implicitly deleted—this should be considered a severe risk, and should be difficult to do by accident.

If input or output is a file, support `-` to read from `stdin` or write to `stdout`. This lets the output of another command be the input of your command and vice versa, without using a temporary file. For example, `tar` can extract files from `stdin`:

```
$ curl https://example.com/something.tar.gz | tar xvf -
```

If a flag can accept an optional value, allow a special word like "none." For example, `ssh -F` takes an optional filename of an alternative `ssh_config` file, and `ssh -F none` runs SSH with no config file. Don't just use a blank value—this can make it ambiguous whether arguments are flag values or arguments.

If possible, make arguments, flags and subcommands order-independent. A lot of CLIs, especially those with subcommands, have unspoken rules on where you can put various arguments. For example a command might have a `--foo` flag that only works if you put it before the subcommand:

Contents

```
mycmd --foo=1 subcmd
works

$ mycmd subcmd --foo=1
unknown flag: --foo
```

This can be very confusing for the user—especially given that one of the most common things users do when trying to get a command to work is to hit the up arrow to get the last invocation, stick another option on the end, and run it again. If possible, try to make both forms equivalent, although you might run up against the limitations of your argument parser.

Allow sensitive argument values to be passed in via files. Let's say your command takes a secret via a `--password` argument. A raw `--password` argument will leak the secret into `ps` output and potentially shell history. It's easy to misuse. Consider allowing secrets only via files, e.g. with a `--password-file` argument. A `--password-file` argument allows a secret to be passed in discreetly, in a wide variety of contexts.

(One could read a password file in Bash by using `--password $(< password.txt)`. Unfortunately, not every context in which a command is run will have access to magical shell substitutions. For example, `systemd` service definitions, `exec` system calls, and some `Dockerfile` command forms do not support the substitutions available in most shells.)

Contents

CLIG: Interactivity

Only use prompts or interactive elements if `stdin` is an interactive terminal (a TTY). This is a pretty reliable way to tell whether you're piping data into a command or whether it's being run in a script, in which case a prompt won't work and you should throw an error telling the user what flag to pass.

If `--no-input` is passed, don't prompt or do anything interactive. This allows users an explicit way to disable all prompts in commands. If the command requires input, fail and tell the user how to pass the information as a flag.

If you're prompting for a password, don't print it as the user types. This is done by turning off echo in the terminal. Your language should have helpers for this.

Let the user escape. Make it clear how to get out. (Don't do what vim does.) If your program hangs on network I/O etc, always make Ctrl-C still work. If it's a wrapper around program execution where Ctrl-C can't quit (SSH, tmux, telnet, etc), make it clear how to do that. For example, SSH allows escape sequences with the `~` escape character.

Contents

CLIG: Subcommands

If you've got a tool that's sufficiently complex, you can reduce its complexity by making a set of subcommands. If you have several tools that are very closely related, you can make them easier to use and discover by combining them into a single command (for example, RCS vs. Git).

They're useful for sharing stuff—global flags, help text, configuration, storage mechanisms.

Be consistent across subcommands. Use the same flag names for the same things, have similar output formatting, etc.

Use consistent names for multiple levels of subcommand. If a complex piece of software has lots of objects and operations that can be performed on those objects, it is a common pattern to use two levels of subcommand for this, where one is a noun and one is a verb. For example, `docker container create`. Be consistent with the verbs you use across different types of objects.

Either `noun verb` or `verb noun` ordering works, but `noun verb` seems to be more common.

Further reading: User experience, CLIs, and breaking the world, by John Starich.

Don't have ambiguous or similarly-named commands. For example, having two subcommands called "update" and "upgrade" is quite confusing. You might want to use different words, or disambiguate with extra words.

CLIG: Robustness

Validate user input. Everywhere your program accepts data from the user, it will eventually be given bad data. Check early and bail out before anything bad happens, and make the errors understandable.

Responsive is more important than fast. Print something to the user in <100ms. If you're making a network request, print something before you do it so it doesn't hang and look broken.

Show progress if something takes a long time. If your program displays no output for a while, it will look broken. A good spinner or progress indicator can make a program appear to be faster than it is.

Ubuntu 20.04 has a nice progress bar that sticks to the bottom of the terminal.

If the progress bar gets stuck in one place for a long time, the user won't know if stuff is still happening or if the program's crashed. It's good to show estimated time remaining, or even just have an animated component, to reassure them that you're still working on it.

There are many good libraries for generating progress bars. For example, tqdm for Python, schollz/progressbar for Go, and node-progress for Node.js.

Contents

Do stuff in parallel where you can, but be thoughtful about it. It's already difficult to report progress in the shell; doing it for parallel processes is ten times harder. Make sure it's robust, and that the output isn't confusingly interleaved. If you can use a library, do so—this is code you don't want to write yourself. Libraries like tqdm for Python and schollz/progressbar for Go support multiple progress bars natively.

The upside is that it can be a huge usability gain. For example, docker pull's multiple progress bars offer crucial insight into what's going on.

One thing to be aware of: hiding logs behind progress bars when things go *well* makes it much easier for the user to understand what's going on, but if there is an error, make sure you print out the logs. Otherwise, it will be very hard to debug.

Make things time out. Allow network timeouts to be configured, and have a reasonable default so it doesn't hang forever.

Make it idempotent. If the program fails for some transient reason (e.g. the internet connection went down), you should be able to hit <up> and <enter> and it should pick up from where it left off.

Make it crash-only. This is the next step up from idempotence. If you can avoid needing to do any cleanup after operations, or you can defer that cleanup to the next run, your program can exit immediately on failure or interruption. This makes it both more robust and more responsive.

Citation: Crash-only software: More than meets the eye.

People are going to misuse your program. Be prepared for that. They will wrap it in scripts, use it on bad internet connections, run many instances of it at once, and use it in environments you haven't tested in, with quirks you didn't anticipate. (Did you know macOS filesystems are case-insensitive but also case-preserving?)

Contents

CLIG: Future-proofing

In software of any kind, it's crucial that interfaces don't change without a lengthy and well-documented deprecation process. Subcommands, arguments, flags, configuration files, environment variables: these are all interfaces, and you're committing to keeping them working. (Semantic versioning can only excuse so much change; if you're putting out a major version bump every month, it's meaningless.)

Keep changes additive where you can. Rather than modify the behavior of a flag in a backwards-incompatible way, maybe you can add a new flag—as long as it doesn't bloat the interface too much. (See also: Prefer flags to args.)

Warn before you make a non-additive change. Eventually, you'll find that you can't avoid breaking an interface. Before you do, forewarn your users in the program itself: when they pass the flag you're looking to deprecate, tell them it's going to change soon. Make sure there's a way they can modify their usage today to make it future-proof, and tell them how to do it.

Contents

If possible, you should detect when they've changed their usage and not show the warning any more: now they won't notice a thing when you finally roll out the change.

Changing output for humans is usually OK. The only way to make an interface easy to use is to iterate on it, and if the output is considered an interface, then you can't iterate on it. Encourage your users to use `--plain` or `--json` in scripts to keep output stable (see Output).

Don't have a catch-all subcommand. If you have a subcommand that's likely to be the most-used one, you might be tempted to let people omit it entirely for brevity's sake. For example, say you have a `run` command that wraps an arbitrary shell command:

```
$ mycmd run echo "hello world"
```

You could make it so that if the first argument to `mycmd` isn't the name of an existing subcommand, you assume the user means `run`, so they can just type this:

```
$ mycmd echo "hello world"
```

This has a serious drawback, though: now you can never add a subcommand named `echo`—or *anything at all*—without risking breaking existing usages. If there's a script out there that uses `mycmd echo`, it will do something entirely different after that user upgrades to the new version of your tool.

Contents

Don't allow arbitrary abbreviations of subcommands. For example, say your command has an `install` subcommand. When you added it, you wanted to save users some typing, so you allowed them to type any non-ambiguous prefix, like `mycmd ins`, or even just `mycmd i`, and have it be an alias for `mycmd install`. Now you're stuck: you can't add any more commands beginning with `i`, because there are scripts out there that assume `i` means `install`.

There's nothing wrong with aliases—saving on typing is good—but they should be explicit and remain stable.

Don't create a "time bomb." Imagine it's 20 years from now. Will your command still run the same as it does today, or will it stop working because some external dependency on the internet has changed or is no longer maintained? The server most likely to not exist in 20 years is the one that you are maintaining right now. (But don't build in a blocking call to Google Analytics either.)

CLIG: Signals and control characters

Contents

If a user hits Ctrl-C (the INT signal), exit as soon as possible. Say something immediately, before you start clean-up. Add a timeout to any clean-up code so it can't hang forever.

If a user hits Ctrl-C during clean-up operations that might take a long time, skip them. Tell the user what will happen when they hit Ctrl-C again, in case it is a destructive action.

For example, when quitting Docker Compose, you can hit Ctrl-C a second time to force your containers to stop immediately instead of shutting them down gracefully.

```
$  docker-compose up
…
^CGracefully stopping... (press Ctrl+C again to force)
```

Your program should expect to be started in a situation where clean-up has not been run. (See Crash-only software: More than meets the eye.)

Command-line tools have lots of different types of configuration, and lots of different ways to supply it (flags, environment variables, project-level config files). The best way to supply each piece of configuration depends on a few factors, chief among them *specificity*, *stability* and *complexity*.

Configuration generally falls into a few categories:

1. Likely to vary from one invocation of the command to the next.

   Examples:

   > Setting the level of debugging output
   > Enabling a safe mode or dry run of a program

   Recommendation: **Use flags.** Environment variables may or may not be useful as well.

2. Generally stable from one invocation to the next, but not always. Might vary between projects. Definitely varies between different users working on the same project.

   This type of configuration is often specific to an individual computer.

   Examples:

   > Providing a non-default path to items needed for a program to start
   > Specifying how or whether color should appear in output
   > Specifying an HTTP proxy server to route all requests through

Recommendation: **Use flags and probably environment variables too.** Users may want to set the variables in their shell profile so they apply globally, or in `.env` for a particular project.

If this configuration is sufficiently complex, it may warrant a configuration file of its own, but environment variables are usually good enough.

3. Stable within a project, for all users.

   This is the type of configuration that belongs in version control. Files like `Makefile`, `package.json` and `docker-compose.yml` are all examples of this.

   Recommendation: **Use a command-specific, version-controlled file.**

Follow the XDG-spec. In 2010 the X Desktop Group, now freedesktop.org, developed a specification for the location of base directories where config files may be located. One goal was to limit the proliferation of dotfiles in a user's home directory by supporting a general-purpose `~/.config` folder. The XDG Base Directory Specification (full spec, summary) is supported by yarn, fish, wireshark, emacs, neovim, tmux, and many other projects you know and love.

If you automatically modify configuration that is not your program's, ask the user for consent and tell them exactly what you're doing. Prefer creating a new config file (e.g. `/etc/cron.d/myapp`) rather than appending to an existing config file (e.g. `/etc/crontab`). If you have to append or modify to a system-wide config file, use a dated comment in that file to delineate your additions.

Apply configuration parameters in order of precedence. Here is the precedence for config parameters, from highest to lowest:

Flags
The running shell's environment variables
Project-level configuration (eg. `.env`)
User-level configuration
System wide configuration

Contents

CLIG: Environment variables

Environment variables are for behavior that *varies with the context* in which a command is run. The "environment" of an environment variable is the terminal session—the context in which the command is running. So, an env var might change each time a command runs, or between terminal sessions on one machine, or between instantiations of one project across several machines.

Environment variables may duplicate the functionality of flags or configuration parameters, or they may be distinct from those things. See Configuration for a breakdown of common types of configuration and recommendations on when environment variables are most appropriate.

For maximum portability, environment variable names must only contain letters, numbers, and underscores (and mustn't start with a number). Which means `O_O` and `OWO` are the only emoticons that are also valid environment variable names.

Aim for single-line environment variable values. While multi-line values are possible, they create usability issues with the `env` command.

Contents

Avoid commandeering widely used names. Here's a list of POSIX standard env vars.

Check general-purpose environment variables for configuration values when possible:

- `NO_COLOR`, to disable color (see Output).
- `DEBUG`, to enable more verbose output.
- `EDITOR`, if you need to prompt the user to edit a file or input more than a single line.
- `HTTP_PROXY`, `HTTPS_PROXY`, `ALL_PROXY` and `NO_PROXY`, if you're going to perform network operations. (The HTTP library you're using might already check for these.)
- `SHELL`, if you need to open up an interactive session of the user's preferred shell. (If you need to execute a shell script, use a specific interpreter like `/bin/sh`)
- `TERM`, `TERMINFO` and `TERMCAP`, if you're going to use terminal-specific escape sequences.
- `TMPDIR`, if you're going to create temporary files.
- `HOME`, for locating configuration files.
- `PAGER`, if you want to automatically page output.
- `LINES` and `COLUMNS`, for output that's dependent on screen size (e.g. tables).

Read environment variables from `.env` where appropriate. If a command defines environment variables that are unlikely to change as long as the user is working in a particular directory, then it should also read them from a local `.env` file so users can configure it differently for different projects without having to specify them every time. Many languages have libraries for reading `.env` files (Rust, Node, Ruby).

Don't use `.env` as a substitute for a proper configuration file. `.env` files have a lot of limitations:

- A `.env` file is not commonly stored in source control
- (Therefore, any configuration stored in it has no history)
- It has only one data type: string
- It lends itself to being poorly organized
- It makes encoding issues easy to introduce
- It often contains sensitive credentials & key material that would be better stored more securely

Contents

If it seems like these limitations will hamper usability or security, then a dedicated config file might be more appropriate.

CLIG: Naming

The name of your program is particularly important on the CLI: your users will be typing it all the time, and it needs to be easy to remember and type.

Make it a simple, memorable word. But not too generic, or you’ll step on the toes of other commands and confuse users. For example, both ImageMagick and Windows used the command `convert`.

Use only lowercase letters, and dashes if you really need to. `curl` is a good name, `DownloadURL` is not.

Keep it short. Users will be typing it all the time. Don't make it *too* short: the very shortest commands are best reserved for the common utilities used all the time, such as `cd`, `ls`, `ps`.

Make it easy to type. Some words flow across the QWERTY keyboard much more easily than others, and it's not just about brevity. `plum` may be short but it's an awkward, angular dance. `apple` trips you up with the double letter. `orange` is longer than both, but flows much better.

Further reading: The Poetics of CLI Command Names

CLIG: Distribution

If possible, distribute as a single binary. If your language doesn't compile to binary executables as standard, see if it has something like PyInstaller. If you really can't distribute as a single binary, use the platform's native package installer so you aren't scattering things on disk that can't easily be removed. Tread lightly on the user's computer.

If you're making a language-specific tool, such as a code linter, then this rule doesn't apply—it's safe to assume the user has an interpreter for that language installed on their computer.

Make it easy to uninstall. If it needs instructions, put them at the bottom of the install instructions—one of the most common times people want to uninstall software is right after installing it.

CLIG: Analytics

Usage metrics can be helpful to understand how users are using your program, how to make it better, and where to focus effort. But, unlike websites, users of the command-line expect to be in control of their environment, and it is surprising when programs do things in the background without telling them.

Do not phone home usage or crash data without consent. Users will find out, and they will be angry. Be very explicit about what you collect, why you collect it, how anonymous it is and how you go about anonymizing it, and how long you retain it for.

Ideally, ask users whether they want to contribute data ("opt-in"). If you choose to do it by default ("opt-out"), then clearly tell users about it on your website or first run, and make it easy to disable.

Examples of projects that collect usage statistics:

Contents

Angular.js collects detailed analytics using Google Analytics, in the name of feature prioritization. You have to explicitly opt in. You can change the tracking ID to point to your own Google Analytics property if you want to track Angular usage inside your organization.

Homebrew sends metrics to Google Analytics and has a nice FAQ detailing their practices.

Next.js collects anonymized usage statistics and is enabled by default.

Consider alternatives to collecting analytics.

Instrument your web docs. If you want to know how people are using your CLI tool, make a set of docs around the use cases you'd like to understand best, and see how they perform over time. Look at what people search for within your docs.

Instrument your downloads. This can be a rough metric to understand usage and what operating systems your users are running.

Talk to your users. Reach out and ask people how they're using your tool. Encourage feedback and feature requests in your docs and repos, and try to draw out more context from those who submit feedback.

Further reading: Open Source Metrics

CLIG: Further reading

The Unix Programming Environment, Brian W. Kernighan and Rob Pike
POSIX Utility Conventions
Program Behavior for All Programs, GNU Coding Standards
12 Factor CLI Apps, Jeff Dickey
CLI Style Guide, Heroku