

way of the hax0r
n00b to console punk

Chapter 2

Anatomy of a Command

In the shell, you type a series of commands. n00bs look at commands as magic incantations to recite. However, commands have a common mode of operation. Learning how commands operate will allow you to quickly understand and use existing commands and build new commands of your own.

Remember our console notation: a bold line beginning with \$ is a command to enter. Do not type the \$ marker, even if your actual prompt looks completely different. Unmarked lines show output.

The concepts in this chapter are:

- **command names, arguments, and options;**
- **quoting and escaping**, which keep spaces and special characters inside an argument;
- what the shell changes before a program runs: **environment variables, command substitution, wildcards, and quote removal;**
- the kinds of thing a name can select: **built-ins, external programs, aliases, functions, and cmdlets;**
- the **environment, PATH**, and how the shell finds a program;
- **package managers** and installation;
- **help** and **man** and how to read a syntax line.

A Name and Some Arguments

To run a command, type the command name at the command prompt in the shell followed by **arguments**. We use the term **options** or **flags** for **arguments** that are keywords to indicate a certain behavior. We already saw the echo command that prints the message, given as an **argument** to the command; echo hello will print hello to the terminal.

On Linux and OSX, the head command will show the first few lines of a file, so head README.md will print the first few lines of the file README.md. If we just want the first two lines of the file we can use the -n option to specify the number of lines. head -n 2 README.md runs the command head with three arguments, one of which is an option.

Most shells — all of the ones we are talking about — use spaces to separate arguments. What if we want to pass an argument that has spaces in it? Like

```
$ echo "Two spaces after the period. Please."
Two spaces after the period. Please.
```

We will see the rules around using quotes a little later. Hopefully, you have a bit of intuition on how to run a command and pass arguments to it.

Now let's ask for a directory listing: the names of files and directories in a folder. Use an ordinary filesystem directory, such as the home folder where your shell started.

Bash/zsh

`ls` lists directory contents. The argument `.` means the current directory. Run these one at a time and compare the results:

```
$ ls .
$ ls -a .
```

Normally, `ls` leaves out names beginning with a dot. The `-a` option includes them, along with the special entries `.` and `..`. We will explore what those entries mean in Chapter 3. [1]

Read the second command in pieces:

Piece	Role	Meaning here
<code>ls</code>	Command name	List directory contents
<code>-a</code>	Option	Include names normally hidden by their leading dot
<code>.</code>	Path argument	Look in the current directory

The option and path are both arguments passed to `ls`, but they do different jobs. The path says **where**; the option changes **how** the results are selected.

PowerShell

`Get-ChildItem` lists the items inside a location. For a filesystem directory, those items are files and subdirectories.

```
$ Get-ChildItem -Path .
$ Get-ChildItem -Path . -Force
```

PowerShell calls these named controls **parameters**. `-Path` takes a value: the `.` immediately after it. `-Force` is a **switch parameter**; here, it includes hidden items that the normal listing leaves out. It does not bypass access permissions. [2]

Read `-Path .` as a pair: parameter name, then its argument value. The long names make the request readable once you know where the pieces begin and end.

Command Prompt

In `cmd.exe`, `dir` lists directory contents.

```
$ dir .  
$ dir /a .
```

The /a option includes entries with hidden or system attributes that the default listing omits. With no attribute selection after /a, it includes all attribute types. [3]

The three pairs of commands have a similar purpose, but their output formats and definitions of hidden files differ. Do not expect identical listings just because the requests sound similar in English.



Trap: An option belongs to a particular command. -a, -Force, and /a all mean different things to different commands.

Short Names Are Not a Universal Code

Many UNIX tools use a single hyphen for short options, such as -a, and two for longer names, such as --all, but there are plenty of exceptions like `java -version`. GNU `ls`, commonly found on Linux, accepts --all as another form of -a. Do not assume every system's `ls` accepts the same long options. [1]

PowerShell's full parameter names normally use one hyphen, as in -Path. Many traditional Windows commands use a slash, as in /a.

You will sometimes see short options combined, but do not combine letters just because a command looks shorter that way. First check that command's help. And do not assume a word like *force* has the same consequences everywhere: showing hidden entries is very different from forcing an overwrite.

Spaces Can Be Part of an Argument

Suppose your message is `console punk`, with a space. You need a way to say that the space belongs inside the text rather than separating two arguments. Quotation marks provide that grouping.

Bash/zsh

`echo` is not a good argument counter: it puts spaces between the arguments it prints. For this experiment, use `printf`, which prints according to a **format string**. In '`<%s>\n`', %s marks a string to insert, < and > make its boundaries visible, and \n requests a new line. The format is reused for the remaining values. [4]

```
$ printf '<%s>\n' console punk  
<console>  
<punk>  
$ printf '<%s>\n' "console punk"  
<console punk>
```

Without the quotation marks, `console` and `punk` are separate arguments after the format string. With them, `console punk` is one argument containing a space. The grouping quotation marks themselves are not part of that value.

Other Ways to Keep a Space

Single quotes also group this text into one argument. Alternatively, a backslash can **escape** an unquoted space: tell the shell to treat it as a literal character instead of a separator. You can even join unquoted and quoted pieces without putting a separating space between them:

```
$ printf '<%s>\n' 'console punk'
<console punk>
$ printf '<%s>\n' console\ punk
<console punk>
$ printf '<%s>\n' console" punk"
<console punk>
```

All three pass the same value after the format string. The backslash in `console\ punk` is shell syntax; it is not part of the resulting text. [5], [6]

Quotes Inside the Text

Use the other kind of quote around a string that contains quotation marks:

```
$ printf '%s\n' 'say "hello"'
say "hello"
$ printf '%s\n' "it's working"
it's working
```

Inside double quotes, `\"` includes a literal double quote:

```
$ printf '%s\n' "say \"hello\""
say "hello"
```

A backslash does not escape an apostrophe inside ordinary single quotes. Use double quotes around that text, as in `"it's working"`, instead. Single and double quotes are not interchangeable for every string: double quotes still allow the shell to substitute some things inside them. The next section explains what those things are, starting with variables. [5], [6]

PowerShell

Write-Output writes values to PowerShell's output stream. Here, the value is one **string**, meaning a piece of text:

```
$ Write-Output "console punk"
console punk
```

The quotation marks keep the two words together as one string. [7] In a default PowerShell session, the familiar name `echo` is an alias for `Write-Output`. We will unpack aliases in a moment. [8]

Other Ways to Keep a Space

Single quotes also work for this message. PowerShell's escape character is the **backtick**, not the backslash; a backtick immediately before an unquoted space keeps it in the argument:

```
$ Write-Output 'console punk'
console punk
$ Write-Output console` punk
console punk
```

The backtick is different from the apostrophe used for single quotes. Quoting the whole string is usually easier to read. [9]

Quotes Inside the Text

As in Bash/zsh, the other kind of quotation mark can appear inside the surrounding pair:

```
$ Write-Output 'say "hello"'
say "hello"
$ Write-Output "it's working"
it's working
```

Inside a double-quoted string, escape a double quote with a backtick. Inside a single-quoted string, write an apostrophe twice to include one literal apostrophe:

```
$ Write-Output "say `"hello`""
say "hello"
$ Write-Output 'it''s working'
it's working
```

Single-quoted strings are literal; double-quoted strings allow the same kind of substitution, which the next section explains. These examples use a PowerShell cmdlet. Passing embedded quotes to external programs has additional rules and version differences; do not assume the same spelling works everywhere. [7], [9]

Command Prompt

Command Prompt has different parsing rules, and its built-in echo treats the following message differently from the UNIX printf example. For the simple message here, enter:

```
$ echo console punk
console punk
```

Do not use this result to infer how another command handles spaces. For a path containing spaces, use double quotation marks around the path, such as "C:\Program Files". Single quotation marks are not a substitute for double quotation marks in Command Prompt. [10], [11]

Unlike Bash's echo or PowerShell's Write-Output, Command Prompt's internal echo keeps double quotation marks in the displayed message:

```
$ echo say "hello"
say "hello"
$ echo "console punk"
"console punk"
```

This is useful for displaying quotes, but it is not a general rule for passing quoted text to other programs. Command Prompt uses the caret (^) to escape certain special characters, but a caret before a space is not a reliable replacement for quoting an argument to an external program. Keep using double quotes around paths with spaces, and consult the target program's rules for embedded quotes. [10], [12]



Trap: Escape characters are shell-specific. Bash/zsh use a backslash; PowerShell uses a backtick; Command Prompt uses a caret for some purposes. Copying an escape from another shell can change the text or split an argument instead of protecting it.

Use the quotation marks shown in the example for your shell, and keep the opening and closing marks paired. If an unfinished quote leaves Bash, zsh, or PowerShell waiting for more input, press Ctrl+C to cancel and start again.

What the Shell Does to Your Command

The quotation marks in the previous section disappeared before the program saw its argument. That is one example of a general rule: the shell reads your line, transforms parts of it, and only then runs the command with the results. The program never sees your quotes, your variable names, or your wildcards. It sees the arguments the shell produced.

Environment Variables

Shells also have variables — names that represent a value. We call these variables **environment variables**.

When we run commands, we can use these variables as arguments. The commands themselves can also read the arguments from the environment.

The shell fills some variables when it starts: HOME holds the path to your home directory, and PATH holds the list of directories it searches for programs. You can create your own, and a later section shows how.

Shell	Reference	Value it becomes
Bash/zsh	\$HOME	/home/student
PowerShell	\$HOME	C:\Users\student
Command Prompt	%USERPROFILE%	C:\Users\student

Commands themselves can also use environment variables. The EDITOR environment variable specifies the editor to use if the command wants to edit a file; git is a big user of this one. The LANG environment variable tells the preferred language for commands that support multiple languages.

See the Environment

Bash/zsh:

```
$ printenv HOME
$ env
```

printenv prints one variable; env with no arguments lists them all. A shell variable is created with NAME=value and becomes an environment variable when you export it: [13], [14]

```
$ DEMO=hello
$ bash -c 'echo "child sees [$DEMO]"'
child sees []
$ export DEMO
$ bash -c 'echo "child sees [$DEMO]"'
child sees [hello]
```

bash -c starts a child shell to run one command, which shows what a program started from this shell would inherit. The single quotes keep \$DEMO from being expanded by the parent before the child runs.

PowerShell:

```
$ $env:HOME
$ Get-ChildItem Env:
```

PowerShell exposes the environment as the Env: drive, and \$env:NAME reads or sets one variable. Assigning \$env:DEMO = "hello" creates an environment variable that child programs inherit; an ordinary \$demo = "hello" variable stays inside PowerShell. [15]

Command Prompt:

```
$ echo %USERPROFILE%
$ set
```

set with no arguments lists the environment, and set DEMO=hello creates a variable in this session. Command Prompt has no separate shell variables; everything set creates is inherited by child programs. [16]

Changes made this way last only until the session ends. Chapter 10 covers startup files that make settings persistent.

The Transformations

Variable expansion is one of several transformations the shell applies when it runs a command. The ones you will see most often are:

- **Variable expansion** replaces an environment variable reference with its stored value.
- **Command substitution** runs another command and replaces the reference with its output.
- **Filename expansion**, also called **globbing**, replaces a pattern such as *.md with the matching filenames. Bash and zsh do this; PowerShell and Command Prompt leave the pattern for the command to interpret.
- **Quote removal** strips the quotation marks and escape characters that controlled the other steps.

Quotes control which transformations happen. Every shell has a way to say “expand this” and “leave this alone”, but the spelling differs, so learn the rules for the shell you are using.



Trap: Single quotes block substitution in Bash, zsh, and PowerShell. `echo '$HOME'` prints the five characters `$HOME`, not your home directory, and the same goes for command substitution and, in Bash and zsh, wildcards. Use double quotes when you want the value and single quotes when you want the text exactly as typed. Command Prompt is the exception: it has no quote that blocks substitution, and single quotes are ordinary characters there, so `echo '%USERPROFILE%'` expands the variable and prints the quotes around it.

Bash/zsh

`printf '%s>\n'` shows exactly which arguments a program received, so keep using it as a probe.

Start with the `HOME` variable and compare three spellings of a reference to it:

```
$ printf '%s>\n' $HOME "$HOME" '$HOME'
</home/student>
</home/student>
<$HOME>
```

Unquoted and double-quoted references are both expanded. Single quotes turn expansion off, so the program receives the five characters `$HOME` unchanged. The difference between the first two forms appears when the value contains spaces: unquoted, the shell splits it into several arguments; inside double quotes, it stays one argument. Use `"$HOME"` by default. [5], [17], [18]

`$(...)` runs a command and substitutes its output:

```
$ printf '%s>\n' "$(date +%Y)"
<2026>
```

Your year will differ. The point is that `date` ran first and `printf` received its output as an argument. [19]

A **wildcard** pattern is expanded by the shell into matching names. `*` matches any run of characters within a name. Try this in a directory containing Markdown files, or read the output as an illustration:

```
$ printf '%s>\n' *.md
<notes.md>
<todo.md>
$ printf '%s>\n' '*.md'
<*.md>
```

The program never sees `*.md` in the first command; it receives one argument per matching file. With quotes, it receives the literal pattern. [20]

Bash and zsh disagree about a pattern that matches nothing. Bash passes the pattern through unchanged, so the program receives `*.zzz`. zsh reports no matches found and does not run the command at all, unless you change its `NOMATCH` option. [20], [21]



Trap: A wildcard is expanded before the command sees it, so a typo changes which files a command touches. `rm * .bak` with an accidental space names every file in the directory, not the `.bak` files. Before running a destructive command with a wildcard, run `printf '<%s>\n'` with the same pattern to see what the shell will pass.

PowerShell

PowerShell uses a `$` prefix for variables and expands them inside double quotes but not single quotes:

```
$ Write-Output $HOME
C:\Users\student
$ Write-Output "$HOME"
C:\Users\student
$ Write-Output '$HOME'
$HOME
```

`$(...)` inside a double-quoted string is a **subexpression**: PowerShell evaluates it and inserts the result. [7], [22]

```
$ Write-Output "Year: $(Get-Date -Format yyyy)"
Year: 2026
```

To see which arguments a command received, run a **script block** with `&` and print its `$args`:

```
$ & { foreach ($a in $args) { "<$a>" } } console punk "console punk"
<console>
<punk>
<console punk>
```

You do not need to understand the loop yet; treat it as a probe like `printf` above. [23]

Use the same probe with a wildcard:

```
$ & { foreach ($a in $args) { "<$a>" } } *.md
<*.md>
```

PowerShell did not expand the pattern. Cmdlets such as `Get-ChildItem` interpret wildcards in their own parameters, which is why `Get-ChildItem *.md` lists matching files and why `-LiteralPath` exists to switch that off. [24]

External programs receive the arguments PowerShell built, with its own rules for quotes. `--%` stops PowerShell parsing the rest of the line and passes it to the program almost as typed; it exists for cases where those rules get in the way. [9]

Command Prompt

Command Prompt reads a variable with percent signs, and it has no quote that switches expansion off:

```
$ echo %USERPROFILE%
C:\Users\student
$ echo "%USERPROFILE%"
"C:\Users\student"
```

Both forms expand the variable; the quotation marks stay in the output because `echo` displays its text as written. `^` escapes characters such as `&` and `|`, which otherwise have a special meaning that we return to in Chapter 7. [10], [16]

Command Prompt does not expand wildcards for you. `dir *.md` works because `dir` interprets the pattern itself. An external program receives `*.md` literally and decides what to do with it.



Wut: The same line can mean three things in three shells. `echo $HOME` prints your home directory in Bash, zsh, and PowerShell, but Command Prompt prints the text `$HOME` because `$` is not special there. `echo %HOME%` is a variable reference in Command Prompt and plain text in Bash and zsh.

What Is a Command?

Not every command is a separate program. A name can lead to several different kinds of thing:

- A **built-in** is implemented by the shell itself. Bash and zsh provide a built-in `echo`; Command Prompt has its own internal `echo` and `dir` commands.
- An **external program** is a separate executable the shell starts. The usual UNIX `ls` is an example. It could even be a program or script that you have written.
- An **alias** gives a command another name or, in shells such as Bash and zsh, substitutes text into the command line.
- A **function** is a named piece of shell code. We will write our own later.
- A **cmdlet**, pronounced *command-let*, is a command designed for PowerShell's command system. `Get-ChildItem` and `Write-Output` are examples.

Bash documents the distinction between its built-ins and separate programs; PowerShell's command discovery distinguishes aliases, functions, cmdlets, and applications. [25], [26]

PowerShell cmdlet names conventionally use a **Verb-Noun** pattern. `Get-ChildItem` gets child items; `Get-Command` gets information about commands. The hyphen inside the name is part of the name, not an option separator.

Ask Bash or zsh

`type` reports how the shell interprets a command name.

```
$ type echo
```

In an uncustomized Bash or zsh session, it reports that `echo` is a shell built-in. Your shell can report an alias or function instead if your configuration defines one.

Add `-a` to look for all the matches `type` can find:

```
$ type -a ls
```

You may see a path to an executable, several paths, or an alias as well. The exact wording and paths depend on your shell and installation. [4], [27]

Ask PowerShell

Get-Command reports what kind of command a name selects.

```
$ Get-Command echo
$ Get-Command Get-ChildItem
$ Get-Command dir
```

Look at the CommandType and Name columns. With the default aliases, echo points to Write-Output, while dir points to Get-ChildItem. Get-ChildItem itself is a cmdlet. Use Get-Command dir -All to investigate multiple commands sharing that name. [2], [26]



Wut: PowerShell's dir is not Command Prompt's dir. The default PowerShell alias selects Get-ChildItem, so it expects that cmdlet's parameters, not Command Prompt's /a syntax. A familiar name is a clue, not a guarantee of familiar behavior.

Ask Command Prompt

help lists many of Command Prompt's available commands and describes their use. For a particular command, give its name:

```
$ help dir
```

For external program files, where .exe searches the current directory and the directories listed in PATH. For example, find the Command Prompt executable:

```
$ where.exe cmd.exe
```

This reports matching file paths, not internal commands or shell aliases. Failure to find an echo.exe would not mean Command Prompt lacks its internal echo. [28], [29]

PATH and Command Lookup

When you type a name that is not a built-in, alias, or function, the shell walks PATH from first directory to last and runs the first matching executable.

Bash/zsh:

```
$ echo "$PATH"
$ command -v ls
/usr/bin/ls
```

PATH uses : between directories. command -v reports what the name resolves to; type -a from earlier lists every candidate. [30]

PowerShell:

```
$ $env:PATH
$ (Get-Command ls).Source
```

On Windows, PATH uses ; between directories; PowerShell on Linux and macOS uses . . Get-Command shows the path in its Source property. [31]

Command Prompt:

```
$ echo %PATH%
$ where.exe notepad.exe
```

where.exe searches the current directory and then PATH, so it finds the program Command Prompt would run. [29]

The current directory is not on PATH in Bash, zsh, or PowerShell, so a program sitting there is run by path, as ./tool or .\tool.exe; Command Prompt searches the current directory before PATH. [10], [30], [31]

Two consequences follow:

- A directory added to PATH by an installer is seen only by shells started after the change. Open a new tab when a freshly installed tool is not found.
- Order matters. When two installations of the same program are both on PATH, the first one wins.

Package Managers

A **package manager** downloads software from a catalog, installs it, records what it installed, and can update or remove it later. Prefer one over downloading installers by hand: it puts programs on PATH consistently and tells you what version you have.

Platform	Package manager	Install example
Debian, Ubuntu	APT	<code>sudo apt install ripgrep</code>
Fedora	DNF	<code>sudo dnf install ripgrep</code>
macOS	Homebrew	<code>brew install ripgrep</code>
Windows	winget	<code>winget install BurntSushi.ripgrep.MSVC</code>

sudo runs a command with administrator rights, which system-wide installation on Linux needs; Chapter 8 explains it. Homebrew and winget are separate projects from the operating system and use their own catalogs. [32], [33], [34]

Programming languages bring their own package managers, such as pip for Python and npm for JavaScript. They install libraries for that language, sometimes with command-line tools, into directories that may or may not be on PATH.

Getting Help

Knowing how to ask a command about itself is more useful than remembering every option. Different command types have different help systems.

Bash/zsh

In **Bash**, `help` describes built-in commands:

```
$ help echo
```

That is Bash's help system, not a universal command available in every shell. For **zsh**, its built-in commands are documented together in the `zshbuiltins` manual:

```
$ man zshbuiltins
```

In **either shell**, use the installed manual for a program such as `ls`:

```
$ man ls
```

`man` opens a **manual page**, usually in a **pager** that lets you move through text a screen at a time. With the common `less` pager, use Space for the next screen, Up and Down Arrow to move a line, and q to quit. [35] Different pager settings can change those keys. If `man` or the requested page is not installed, consult the tool's official documentation instead.

GNU `ls` also accepts `ls --help`. That is a feature of that program; `--help` is not a magic request that works with everything.

PowerShell

`Get-Help` reads help for a command. Start with an overview, then ask for examples:

```
$ Get-Help Get-ChildItem
$ Get-Help Get-ChildItem -Examples
```

If the local help files are missing, you may get only basic syntax rather than the full explanation. `Get-Help Get-ChildItem -Online` opens the online documentation in your browser and needs internet access. You do not need to change system settings just to read that page. [36]

Command Prompt

Use `help` for a list, then ask about the command you want:

```
$ help
$ help dir
$ dir /?
```

For `dir`, the last two forms show its help. Many Windows utilities recognize `/?`, but an independently installed program may use something else. Check that program's documentation instead of trying random switches. [3], [28]

Read the Syntax Before the Details

Help often begins with a **syntax** or **synopsis** line. It describes the allowed usage of a command, rather than giving you a line to paste unchanged.

For example, Bash's `help echo` includes this syntax fragment:

```
echo [-neE] [arg ...]
```

The brackets mark optional pieces. `arg` stands for an argument you supply, and `...` indicates that arguments can be repeated. You do not type the brackets, the word `arg`, or the dots literally.

Windows documentation also uses square brackets for optional pieces and angle brackets for values you supply, such as `<command>`. Read the syntax key when the notation is unfamiliar. [37]

In PowerShell help, a fragment such as `[-Path] <String[]>` has an extra distinction: the parameter name can be omitted when its value is supplied in the appropriate position. `String[]` means the value can be an array of strings; those inner brackets do not mean “optional”. For now, keep writing `-Path` explicitly so the command explains itself. [38]



Tip: Read help with a question in mind. Find the command's purpose, identify the argument or option you need, and read one relevant example. Predict what your version will do before you run it. You can return for the other options when you have a reason to use them.

Key Points

- A command name selects an action; arguments provide information for that action.
- Options and parameters change the request, and their syntax belongs to the command you are using.
- Quotation marks can keep spaces inside an argument, but quoting rules differ across shells.
- The shell expands variables, substitutes command output, and in Bash/zsh expands wildcards before the program runs; quotes control which of these happen.
- A path argument tells a command where to look.
- Environment variables are inherited by child programs; `PATH` decides which executable a name runs, and the first match wins.
- Install tools with a package manager, and open a new shell after an installer changes `PATH`.
- A name may select a built-in, external program, alias, function, or PowerShell cmdlet.
- Use `type`, `Get-Command`, or Command Prompt's help and file-location tools to investigate names.
- Read help as a description of valid commands, not as text to paste without interpretation.

Level Up

n00bs finish the chapter here. You can point to the pieces of a command and find help for the ones you do not understand. Hackers dig deeper.

Choose one command you have used in this chapter and investigate its identity. In Bash or zsh, compare `type echo` with `type -a echo`. In PowerShell, inspect `Get-Command echo -All` and compare the result with `Get-Command Write-Output`. In Command Prompt, compare what `help echo` tells you with what `where.exe cmd.exe` can find. Explain why a help topic, an alias, and an executable path are different kinds of answer.

Then choose one listing option from the help for `ls`, `Get-ChildItem`, or `dir` that changes only the

display or selection of entries. Predict the difference, run the command, and explain what changed. Can you make a similar request in another shell without assuming it uses the same syntax?

Finally, find a command nobody has told you about. In Bash or zsh, `apropos` searches the one-line descriptions of installed manual pages for a keyword, so `apropos directory` lists commands whose descriptions mention directories. Read the manual page of one that looks interesting and try an option from it. [39] In PowerShell, `Get-Command -Noun Item` lists the cmdlets that act on items, and `Get-Help *item*` finds help topics by wildcard. [26], [36] In Command Prompt, read the list from `help` and pick a command whose description you do not understand yet.

Explore what your shell shows you about its own work. In Bash or zsh, `set -x` makes the shell print each command after expansion and before running it; `set +x` turns it off. [40] In PowerShell, `Trace-Command` can show parameter binding, and `Get-Command -All` with `-ShowCommandInfo` exposes more about how a name resolves. Which of the transformations in this chapter can you catch in the act?

Then look at your `PATH` critically. List its directories one per line and check which ones exist. Is anything on it twice, or in an order you would not have chosen? Investigate how a tool such as `pipx`, `pyenv`, or `nvm` manages several versions of a language, and explain how it uses `PATH` to do it.

Keep a small note containing the command, what each part means, and where you found the explanation. That is the beginning of a reference you understand, not just a collection of things to paste.

Exercises

1. **What does this do?** Predict the output of `printf '<%s>\n' "a b" a b` in Bash or zsh, paying attention to the double spaces, then run it.
2. **What does this do?** Predict what `Write-Output "$HOME"` and `Write-Output '$HOME'` print in PowerShell, then explain the difference in one sentence.
3. **What is the problem?** A script written for Bash contains `echo %USERPROFILE%` and prints exactly that text. Explain why, and write the line that would work in Bash.
4. **Why is it like this?** `Get-ChildItem *.md` lists Markdown files in PowerShell, but the script-block probe shows that the pattern was passed through unchanged. Explain who expanded it.
5. **How do you** find out whether `ls` in your shell is an alias, a function, or a program, and where the program lives if it is one?
6. **How do you** make an environment variable visible to a program started from Bash, and how is that different from PowerShell and Command Prompt?

References

- [1] GNU Project, “GNU coreutils: Which files are listed.” Available: https://www.gnu.org/software/coreutils/manual/html_node/Which-files-are-listed.html
- [2] Microsoft, “Get-ChildItem.” Available: <https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.management/get-childitem>
- [3] Microsoft, “Windows commands: dir.” Available: <https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/dir>
- [4] GNU Project, “Bash reference manual: Bash builtins.” Available: https://www.gnu.org/software/bash/manual/html_node/Bash-Builtins.html
- [5] GNU Project, “Bash reference manual: quoting.” Available: https://www.gnu.org/software/bash/manual/html_node/Quoting.html
- [6] Zsh Developers, “The z shell manual: Shell grammar, quoting.” Available: <https://zsh.sourceforge.io/Doc/Release/Shell-Grammar.html#Quoting>
- [7] Microsoft, “About_quoting_rules.” Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_quoting_rules
- [8] Microsoft, “Write-output.” Available: <https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/write-output>
- [9] Microsoft, “About_parsing.” Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_parsing
- [10] Microsoft, “Windows commands: cmd.” Available: <https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/cmd>
- [11] Microsoft, “Windows commands: echo.” Available: <https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/echo>
- [12] Microsoft, “Parsing c command-line arguments.” Available: <https://learn.microsoft.com/en-us/cpp/c-language/parsing-c-command-line-arguments>
- [13] GNU Project, “GNU coreutils manual: Env: Run a command in a modified environment.” Available: https://www.gnu.org/software/coreutils/manual/html_node/env-invocation.html
- [14] GNU Project, “Bash reference manual: environment.” Available: https://www.gnu.org/software/bash/manual/html_node/Environment.html
- [15] Microsoft, “About_environment_variables.” Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_environment_variables
- [16] Microsoft, “Windows commands: Set (environment variable).” Available: https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/set_1

- [17] GNU Project, “Bash reference manual: Shell expansions.” Available: https://www.gnu.org/software/bash/manual/html_node/Shell-Expansions.html
- [18] Zsh Development Group, “The z shell manual: expansion.” Available: <https://zsh.sourceforge.io/Doc/Release/Expansion.html>
- [19] GNU Project, “Bash reference manual: Command substitution.” Available: https://www.gnu.org/software/bash/manual/html_node/Command-Substitution.html
- [20] GNU Project, “Bash reference manual: Filename expansion.” Available: https://www.gnu.org/software/bash/manual/html_node/Filename-Expansion.html
- [21] Zsh Development Group, “The z shell manual: options.” Available: <https://zsh.sourceforge.io/Doc/Release/Options.html>
- [22] Microsoft, “About_variables.” Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_variables
- [23] Microsoft, “About_automatic_variables.” Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_automatic_variables
- [24] Microsoft, “About_wildcards.” Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_wildcards
- [25] GNU Project, “Bash reference manual: Shell builtin commands.” Available: https://www.gnu.org/software/bash/manual/html_node/Shell-Builtin-Commands.html
- [26] Microsoft, “Get-command.” Available: <https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/get-command>
- [27] Zsh Developers, “The z shell manual: Shell builtin commands.” Available: <https://zsh.sourceforge.io/Doc/Release/Shell-Builtin-Commands.html>
- [28] Microsoft, “Windows commands: help.” Available: <https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/help>
- [29] Microsoft, “Windows commands: where.” Available: <https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/where>
- [30] GNU Project, “Bash reference manual: Command search and execution.” Available: https://www.gnu.org/software/bash/manual/html_node/Command-Search-and-Execution.html
- [31] Microsoft, “About_command_precedence.” Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_command_precedence
- [32] Debian Project, “Apt(8): Command-line interface.” Available: <https://manpages.debian.org/stable/apt/apt.8.en.html>
- [33] Homebrew, “Homebrew documentation: installation.” Available: <https://docs.brew.sh/Installation>
- [34] Microsoft, “Use the winget tool to install and manage applications.” Available: <https://learn.microsoft.com/en-us/windows/package-manager/winget/>
- [35] Greenwood Software, “Less: Manual page.” Available: <https://github.com/gws/w/less/blob/master/less.nro.VER>
- [36] Microsoft, “Get-help.” Available: <https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/get-help>
- [37] Microsoft, “Command-line syntax key.” Available: <https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/command-line-syntax-key>

- [38] Microsoft, “About_command_syntax.” Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_command_syntax?view=powershell-7.4
- [39] man-db Project, “Apropos(1): Search the manual page names and descriptions.” Available: <https://man7.org/linux/man-pages/man1/apropos.1.html>
- [40] GNU Project, “Bash reference manual: The set builtin.” Available: https://www.gnu.org/software/bash/manual/html_node/The-Set-Builtin.html