

way of the hax0r
n00b to console punk

Chapter 11

From Commands to Scripts

A pipeline you have typed three times is a script waiting to be written. A **script** is a file of commands that a shell runs in order, and once it exists you can run it with one word, give it arguments, and let it make decisions. Writing scripts is also where “I can explain this command” turns into “I can build something”, which is the console punk milestone.

The concepts in this chapter are:

- a script file, its **interpreter**, and how each shell finds and runs it;
- **arguments** a script receives and **variables** it keeps;
- **conditions**: testing a value, a file, or a command’s success;
- **loops** over arguments, lines, and objects;
- **functions** inside a script;
- **exit codes** and **error handling**, so that a script fails loudly instead of quietly;
- Bash scripts, PowerShell scripts, and what Command Prompt **batch files** can still do.

The scripts here read files and print results. Work in the practice directory, and write the scripts with the editor you chose in Chapter 5.

What a Script Is

Every line you type at a prompt could be a line in a file. A script adds three things: it runs the lines in order without you, it can take input from its arguments, and it can choose what to do next based on what it finds.

Scripts are also where the shells differ most. A Bash script is text commands with text conditions; the tools it calls are separate programs. A PowerShell script is written in a language with typed parameters, objects, and structured error handling. A batch file is Command Prompt’s own scripting, older and more limited, and still what many Windows installers and build steps use. Learn the form of each, and write new scripts in Bash or PowerShell.

A First Script

Bash

Create `greet.sh` with this content:

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

name="${1:-world}"
echo "Hello, $name!"
echo "You passed $# argument(s)."
```

Line by line:

- The **shebang** names the interpreter. `#!/usr/bin/env bash` finds `bash` on `PATH`, which works on Linux and macOS even though they install it in different places. [1]
- `set -euo pipefail` makes the script stop on the first failed command, treat an unset variable as an error, and count a failure anywhere in a pipeline. Start every script with it until you know why you would leave part of it out. [2]
- `$1` is the first argument, and `${1:-world}` supplies a default when it is missing. `$#` is the number of arguments, `$0` the script's own name, and `"$@"` all the arguments, each kept whole. [3], [4]

Make it executable and run it:

```
$ chmod +x greet.sh
$ ./greet.sh
Hello, world!
You passed 0 argument(s).
$ ./greet.sh punk extra
Hello, punk!
You passed 2 argument(s).
```

Assignment has no spaces around `=`, and reading a variable needs `$`. Quote every expansion, `"$name"` not `$name`, or a value with spaces splits into several words, which Chapter 2 showed. [5]

PowerShell

Create `Greet.ps1`:

```
param(
    [string]$Name = 'world'
)
Write-Output "Hello, $Name!"
Write-Output "You passed $($args.Count) extra argument(s)."
```

`param(...)` declares named parameters with types and defaults, and PowerShell binds them from the command line by name or by position. Anything left over lands in `$args`. [6], [7]

```
$ ./Greet.ps1
Hello, world!
You passed 0 extra argument(s).
$ ./Greet.ps1 -Name punk
Hello, punk!
$ ./Greet.ps1 punk extra
Hello, punk!
You passed 1 extra argument(s).
```

No `chmod` is needed, but the execution policy from Chapter 8 must allow local scripts on Windows. The `./` is required for the same reason as in Bash: the current directory is not searched. [8]

Command Prompt

Create `greet.bat`:

```
@echo off
setlocal
set "name=%~1"
if "%name%"==" " set "name=world"
echo Hello, %name%!
echo You passed %* as arguments.
endlocal
```

`@echo off` stops Command Prompt from printing each line as it runs it. `setlocal` keeps variable changes inside the script. `%~1` is the first argument with surrounding quotes removed, and `%*` is all of them. [9], [10]

Run it with `greet.bat punk`. Batch files run from the current directory without a `.\` prefix, which Chapter 2 mentioned as a Command Prompt difference.

Conditions

A script decides by testing something: a string, a number, a file, or whether a command succeeded.

Bash

```
if [ "$#" -eq 0 ]; then
    echo "usage: $0 file..." >&2
    exit 1
fi

if [ -f "$file" ]; then
    echo "$file is a regular file"
elif [ -d "$file" ]; then
    echo "$file is a directory"
else
    echo "$file does not exist" >&2
fi
```

[is a command, test by another name, so the spaces around it and its arguments are required. -eq, -lt, and -gt compare numbers; = and != compare strings; -f, -d, and -e test for a file, a directory, or anything existing. Bash also has [[...]], which adds pattern matching such as [["\$name" == a*]] and does not split variables, at the cost of not existing in plain sh. [11], [12]

The condition can be any command, because if looks only at the exit status:

```
if grep -q 'TODO' "$file"; then
    echo "$file has work left"
fi
```

grep -q prints nothing and reports through its status, which is what a condition needs. [13]

case matches one value against several patterns and is the usual way to handle a command's first argument:

```
case "${1:-}" in
    start) echo "starting" ;;
    stop)  echo "stopping" ;;
    *)     echo "usage: $0 start|stop" >&2; exit 1 ;;
esac
```

PowerShell

```
if ($Path.Count -eq 0) {
    Write-Error 'usage: Count.ps1 -Path file...'
    exit 1
}

if (Test-Path -LiteralPath $file -PathType Leaf) {
    "$file is a file"
} elseif (Test-Path -LiteralPath $file -PathType Container) {
    "$file is a directory"
} else {
    Write-Warning "$file does not exist"
}
```

Comparison operators are words: -eq, -ne, -lt, -gt, -like for wildcards, and -match for regular expressions. String comparisons ignore case unless you use the -ceq family. Test-Path answers the file questions, and a bare string in a script is output, the same as Write-Output. [14], [15]

switch handles several values:

```
switch ($Action) {
    'start' { 'starting' }
    'stop'  { 'stopping' }
    default { Write-Error 'usage: start|stop'; exit 1 }
}
```

Command Prompt

```
if "%~1"==" " (
    echo usage: %~nx0 file... 1>&2
    exit /b 1
)
if exist "%~1" (echo exists) else (echo missing)
if %ERRORLEVEL% neq 0 echo the last command failed
```

Quote both sides of a string comparison so that an empty value does not break the syntax. `if exist` tests files, `neq`, `eq`, `lss`, and `gtr` compare numbers, and `exit /b` leaves the script with a code without closing the window. [16], [17]

Loops

Bash

Loop over the script's arguments, over files from a glob, or over lines of input:

```
for file in "$@"; do
    echo "processing $file"
done
```

```
for md in *.md; do
    echo "$md"
done
```

```
while read -r line; do
    echo "got: $line"
done < notes.txt
```

`for` walks a list of words; `"$@"` keeps each argument whole, and a glob expands to filenames, which Chapter 2 explained. `while read -r line` reads one line at a time from standard input, here redirected from a file. `-r` stops `read` from interpreting backslashes. [18], [19]

`break` leaves a loop and `continue` skips to the next item. Arithmetic uses `$(( ... ))`: `total=$((total + lines))`. [20]

PowerShell

```
foreach ($file in $Path) {
    "processing $file"
}
```

```
foreach ($md in Get-ChildItem -Filter *.md) {
    $md.Name
}
```

```
foreach ($line in Get-Content notes.txt) {
    "got: $line"
}
```

```
}
```

```
$n = 0
while ($n -lt 3) { $n++; "n=$n" }
```

foreach walks any collection: an array, the objects a cmdlet returns, or the lines `Get-Content` produces. `ForEach-Object` from Chapter 7 does the same inside a pipeline. `1..5` produces a range, and arithmetic is ordinary: `$total += $lines`. [21], [22]

Command Prompt

```
for %%f in (*.md) do echo %%f
for /f "usebackq delims=" %%l in ("notes.txt") do echo got: %%l
```

Inside a batch file the loop variable is written `%%f`; at the prompt it is `%f`. `for /f` reads lines, and its option string controls how each line is split. This is where batch files become hard to read; anything more than a simple loop is a reason to switch to PowerShell. [23]

Functions in Scripts

A function inside a script names a step so that the script reads as a sequence of steps.

Bash

```
count_lines() {
    local file="$1"
    wc -l < "$file"
}

usage() {
    echo "usage: $0 file..." >&2
    exit 1
}

[ "$#" -gt 0 ] || usage
for file in "$@"; do
    echo "$file: $(count_lines "$file")"
done
```

`local` keeps a variable inside the function. A function's exit status is that of its last command, or whatever return gives, so functions work in conditions the way commands do. [24]

PowerShell

```
function Get-LineCount {
    param([string]$File)
    return (Get-Content -LiteralPath $File | Measure-Object -Line).Lines
}
```

```
foreach ($file in $Path) {
    "${file}: $(Get-LineCount -File $file)"
}
```

Define functions before the code that calls them; PowerShell reads a script top to bottom. Verb-Noun names follow the convention from Chapter 2, and `Get-Verb` lists the approved verbs. [25]

Command Prompt has no functions; `call :label` jumps to a label and `goto :eof` returns, which is the batch substitute. [10]

Exit Codes and Errors

A script is a command, so it reports an exit code. Zero means it did its job; anything else tells the caller it did not, and Chapter 7 showed how `&&`, `||`, and `if` act on that.

Bash

`exit 1` ends the script with code 1. Without an `exit`, the script's code is that of its last command. With `set -e`, the first failing command ends the script with that command's code, which is what you want from a script that copies files or builds something: stopping at the first error beats continuing with half the inputs.

Write errors to standard error with `>&2` so that they are not mixed into output someone may be piping. When a command is allowed to fail, say so: `grep -q pattern file || true`. [2], [26]

Put the pieces together in `count.sh`:

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

if [ "$#" -eq 0 ]; then
    echo "usage: $0 file..." >&2
    exit 1
fi

total=0
for file in "$@"; do
    if [ ! -f "$file" ]; then
        echo "skipping $file: not a file" >&2
        continue
    fi
    lines=$(wc -l < "$file")
    echo "$file: $lines"
    total=$((total + lines))
done
echo "total: $total"
```

```
$ ./count.sh notes.txt docs/todo.md missing.txt
notes.txt: 4
docs/todo.md: 2
```

```

skipping missing.txt: not a file
total: 6
$ ./count.sh
usage: ./count.sh file...
$ echo $?
1

```

PowerShell

PowerShell has two kinds of error. A **terminating** error stops the script; a **non-terminating** error is reported and the script continues, which is the default for most cmdlet failures. `$ErrorActionPreference = 'Stop'` near the top of a script turns non-terminating errors into terminating ones, the equivalent of `set -e`, and `-ErrorAction Stop` does it for one command. [27]

try and catch handle terminating errors:

```

try {
    $content = Get-Content -LiteralPath $file -ErrorAction Stop
} catch {
    Write-Error "could not read ${file}: $($_.Exception.Message)"
    exit 2
}

```

`exit 2` sets the exit code that `$LASTEXITCODE` shows to whoever ran the script. The full `Count.ps1`:

```

param(
    [Parameter(Mandatory = $true)]
    [string[]]$Path
)
$ErrorActionPreference = 'Stop'

$total = 0
foreach ($file in $Path) {
    if (-not (Test-Path -LiteralPath $file -PathType Leaf)) {
        Write-Warning "skipping ${file}: not a file"
        continue
    }
    $lines = (Get-Content -LiteralPath $file | Measure-Object -Line).Lines
    Write-Output "${file}: $lines"
    $total += $lines
}
Write-Output "total: $total"
if ($total -eq 0) { exit 1 }

```

```

$ ./Count.ps1 notes.txt, docs/todo.md, missing.txt
notes.txt: 4
docs/todo.md: 2
WARNING: skipping missing.txt: not a file
total: 6

```

`[Parameter(Mandatory = $true)]` makes PowerShell ask for the value when it is missing, instead of running with nothing. The commas build one array argument, which is how PowerShell passes a list. [7], [28]

Command Prompt

`exit /b 1` sets the code, and `%ERRORLEVEL%` after a command tells you whether it failed. Batch has no `try`; check `%ERRORLEVEL%` after each command that matters, and use `goto :error` to a label that cleans up. [17]



Tip: Run a script with `bash -x script.sh` or `set Set-PSDebug -Trace 1` to see each line as it runs with its variables expanded. For Bash, install **ShellCheck**: it reads a script and reports quoting mistakes and other traps before you run it.

Key Points

- A script is a file of commands; the shebang picks the interpreter for Bash, and the `.ps1` or `.bat` extension does for Windows.
- Start Bash scripts with `set -euo pipefail` and quote every variable; start PowerShell scripts with `param()` and `$ErrorActionPreference = 'Stop'`.
- `$1`, `$#`, and `"$@"` carry arguments in Bash; `param()` and `$args` do in PowerShell; `%1` and `%*` do in batch files.
- `if` tests an exit status in Bash and an expression in PowerShell; `[ ]` and `Test-Path` answer file questions.
- `for`, `while`, `read`, `foreach`, and `for /f` loop over arguments, files, and lines.
- Exit with a nonzero code on failure, send errors to standard error, and use `try/catch` in PowerShell for errors you expect.
- Batch files still run everywhere on Windows; write new scripts in Bash or PowerShell.

Level Up

n00bs finish the chapter here. You can write a script that takes arguments, checks them, loops, and fails properly. Hackers make scripts that other people can use.

Add `--help` output and argument parsing with `getopts` in Bash, or a `[CmdletBinding()]` block with `-Verbose` and `-WhatIf` support in PowerShell. Read `Get-Help about_Functions_CmdletBindingAttribute` and explain what `SupportsShouldProcess` gives you for free.

Investigate `trap` in Bash for cleaning up temporary files when a script is interrupted, and finally in PowerShell. Then read about `mktemp` and `New-TemporaryFile`.

Write the same small tool three ways: a Bash script, a PowerShell script, and a Python script. Compare the argument handling, error handling, and how each one is installed. When does a shell script stop being the right tool?

Look at **POSIX sh** versus Bash: run your script with `dash` or `sh` and see what breaks. Then read the `ShellCheck` wiki entry for each warning it gives you.

Exercises

1. **What does this do?** Predict the output of `./greet.sh "console punk"` and `./greet.sh console punk`, then run both.
2. **What does this do?** Predict what `for f in *.zzz; do echo "$f"; done` prints in a directory with no `.zzz` files, in Bash and in zsh.
3. **What is the problem?** A script begins `#!/bin/bash` and fails on macOS with “bad interpreter” after the user installed a newer Bash with Homebrew. Explain and fix the shebang.
4. **What is the problem?** `if [$x -eq 5]` gives “command not found”. Explain what the shell looked for.
5. **Why is it like this?** `Get-Item nowhere` in a PowerShell script prints an error and the script keeps going. Explain the kind of error and two ways to make it stop.
6. **Why is it like this?** `set -e` does not stop a script when a command inside an `if` condition fails. Explain why that exception exists.
7. **How do you** make a Bash script refuse to run unless it received exactly two arguments, with a usage message on standard error?
8. **How do you** return a value from a PowerShell function and from a Bash function, and how do the two mechanisms differ?

References

- [1] GNU Project, “Bash reference manual: Shell scripts.” Available: https://www.gnu.org/software/bash/manual/html_node/Shell-Scripts.html
- [2] GNU Project, “Bash reference manual: The set builtin.” Available: https://www.gnu.org/software/bash/manual/html_node/The-Set-Builtin.html
- [3] GNU Project, “Bash reference manual: Special parameters.” Available: https://www.gnu.org/software/bash/manual/html_node/Special-Parameters.html
- [4] GNU Project, “Bash reference manual: Shell parameter expansion.” Available: https://www.gnu.org/software/bash/manual/html_node/Shell-Parameter-Expansion.html
- [5] GNU Project, “Bash reference manual: Shell parameters.” Available: https://www.gnu.org/software/bash/manual/html_node/Shell-Parameters.html
- [6] Microsoft, “About_scripts.” Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_scripts
- [7] Microsoft, “About_functions_advanced_parameters.” Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_functions_advanced_parameters
- [8] Microsoft, “About_execution_policies.” Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_execution_policies
- [9] Microsoft, “Windows commands: setlocal.” Available: <https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/setlocal>
- [10] Microsoft, “Windows commands: call.” Available: <https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/call>
- [11] GNU Project, “Bash reference manual: Conditional constructs.” Available: https://www.gnu.org/software/bash/manual/html_node/Conditional-Constructs.html
- [12] GNU Project, “Bash reference manual: Bash conditional expressions.” Available: https://www.gnu.org/software/bash/manual/html_node/Bash-Conditional-Expressions.html
- [13] GNU Project, “GNU grep manual.” Available: https://www.gnu.org/software/grep/manual/html_node/
- [14] Microsoft, “About_if.” Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_if
- [15] Microsoft, “About_comparison_operators.” Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_comparison_operators
- [16] Microsoft, “Windows commands: if.” Available: <https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/if>

- [17] Microsoft, “Windows commands: exit.” Available: <https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/exit>
- [18] GNU Project, “Bash reference manual: Looping constructs.” Available: https://www.gnu.org/software/bash/manual/html_node/Looping-Constructs.html
- [19] GNU Project, “Bash reference manual: Bash builtins.” Available: https://www.gnu.org/software/bash/manual/html_node/Bash-Builtins.html
- [20] GNU Project, “Bash reference manual: Arithmetic expansion.” Available: https://www.gnu.org/software/bash/manual/html_node/Arithmetic-Expansion.html
- [21] Microsoft, “About_foreach.” Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_foreach
- [22] Microsoft, “About_while.” Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_while
- [23] Microsoft, “Windows commands: for.” Available: <https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/for>
- [24] GNU Project, “Bash reference manual: Shell functions.” Available: https://www.gnu.org/software/bash/manual/html_node/Shell-Functions.html
- [25] Microsoft, “About_functions.” Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_functions
- [26] GNU Project, “Bash reference manual: Exit status.” Available: https://www.gnu.org/software/bash/manual/html_node/Exit-Status.html
- [27] Microsoft, “About_preference_variables.” Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_preference_variables
- [28] Microsoft, “About_try_catch_finally.” Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_try_catch_finally