Introduction to Unix Shell Scripting

Day 1 - Fundamentals

Thomas Junier, Robin Engler

13-15 June 2023

Bash Fundamentals

Course Structure

1. Motivation, Definitions and Objectives
2. Fundamentals
3. Project

Assumed Knowledge

The following notions are assumed to be known:

- Simple commands
- Paths and the Filesystem
- Looking (and Moving) around
- Creating, renaming, moving, and deleting files (including directories)
- Editing ordinary files
- I/O redirection, including pipelines
- Filename Globbing
- Simple file operations with `wc`, `grep`, `cut`, and the like
- Return codes, success and failure, `$?`

Advice

- The solutions to most exercises can be found in the course material
- However, you'll get more out of this course if you at least try first without looking.

Practice

Time for a little warmup:

$\longrightarrow$ **Exercise 1.1 -** a complex pipeline.

Scripting - the next level

Suppose I wanted to perform the same task (as in Exercise 1.1) on **all other `data/sample_??.dna` files**.

What are my options? (suggestions welcome!)

Scripting - the next level

Suppose I wanted to perform the same task (as in Exercise 1.1) on **all other `data/sample_??.dna` files**.

What are my options? (suggestions welcome!)

- Type the commands again, once per file.
- Use the history mechanism ($\uparrow$, `!`) - saves a lot of typing.
- Use a loop - saves even more typing.

However:

- Typing the same command multiple times is boring, time-consuming, and most of all **error-prone**.
- The history is local and limited - what if I need to do the same task on another machine, or when it's no longer in the history?

$\rightarrow$ What we need is a form of **long-term** storage for our command. In other words, a **script**.

What are shell scripts?

Shell scripts:

- Are **plain text** files.
- Contain **shell instructions** - just like the ones we type when working interactively.
- Can be **launched from the terminal** like any other program.

To elaborate...

- The shell is a full **programming language** (not just a command interpreter) with constructs like *variables*, *control structures*, and *functions*.
- However, it is *specialized*[1] for **running other programs** (including in **concurrency** or **parallel**), **passing data** between them, doing **simple** computations, and manipulating files.
- By contrast, Python & the like deal primarily with numbers, strings, and composite structures made out of these.

[1] The shell can be considered a Domain-Specific Language, or DSL

Practice

Let's try to write our first shell script.

→ **Exercise 1.2** - a simple shell script.

Situating Shells among Languages

High-Level

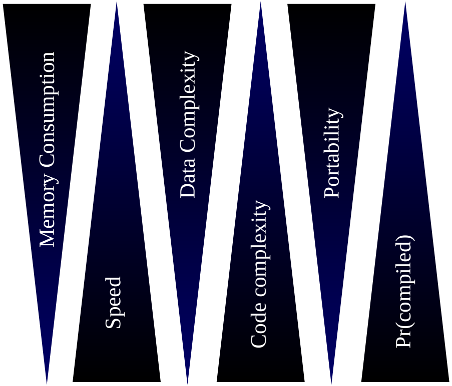
shells

Python R

Java

C C++ Rust

Low-Level



Note: these are *trends*, not absolutes.

What are shell scripts good for?

Typical Use Case

Automating tasks that we already do (but manually) *in the terminal*.

Ok, so what is automation good for?

- Saving time
- Preventing errors
- Ensuring reproducibility
- Avoiding boredom

Nevertheless:

- Interactive tasks *can* be programmed in the shell (e.g. `bashtop`)...
- ... and so can serious computations.
- Whether they *should* be is another question.

Two Styles

Shells can handle a given task in one of two ways:

- **Directly** - the shell performs the task itself.
- By **delegation** - the shell calls one or more other program(s) that do the task.

We will refer to these as the **direct** (or **"pure"**) versus **delegation** styles, respectively.

Two Styles (continued)

- Note that the same program can use both approaches.
- $\rightarrow$ The script from Exercise 1.2 is an example of the delegation style.

Delegation Style - Wall Metaphor

By Pawel Wozniak - Wikimedia Commons

Delegation Style - Conductor Metaphor

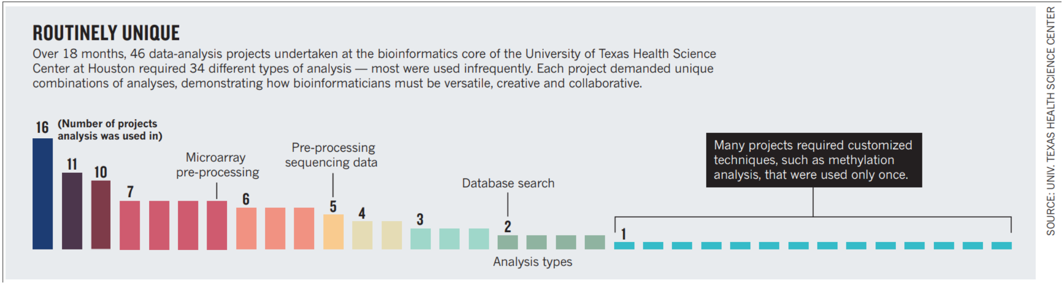
Verdi conducting *Aida* - Engraving by Adrien Marie, 1881 (public domain)

- The conductor does not play any instrument...
- ... but does play an essential part.[a]
- The shell may do no heavy computation...
- ... but still has a crucial role.

[a] And, of course, Verdi still wrote the whole opera score...

Will I need scripting skills?

Will I need scripting skills?



Chang J., *Nature* **520** (2015)

- 14 techniques ($> 40\%$) used in only one project.
- Most frequent technique used in $< 35\%$ of the projects.
- 79% of techniques used in $< 20\%$ of the projects.

Probably yes

The Curse of Uniqueness

Sooner or later you'll face a task that nobody has programmed for you.

→ Shell scripting is almost certainly part of the solution, though probably not the whole solution.

“Hidden” Shell Scripts

Besides your analysis pipelines, you may also need to write some shell code for a number of tasks, such as:

- Build systems (Make, Snakemake).
- HPC scheduling jobs.
- Workflow management systems (Nextflow).
- Containers (Dockerfile).
- Test systems (Bats).

When/How **not** to use Shells?

Pure-style shell is **not** recommended if you need:

- Speed.
- Non-trivial data types, *e.g.* floating-point numbers, any kind of structure beyond 1-D arrays (matrices, trees, database records, dates, ...).
- Functional or object-oriented programming.
- Arbitrary access to file contents.
- Mathematical operations beyond the basics (trigonometry, logs, statistics, etc.).

When/How **not** to use Shells? (continued)

In such cases, it's better to use a more adapted language (`python`, `R`, `C`, ...) for your task.

$\rightarrow$ and then **call your program/script** from in delegation-style from a shell script.

To sum up

We now know:

- What shell scripting is.
- What it can (and cannot) do for us.
- That it may well prove useful.

$\Rightarrow$ we're ready to start.

Just a Few More Points

Learning Objectives

Objectives

1. Use the shell to write programs (surprise!)
   - Know what the shell does.
   - Understand how it does it.
2. Know enough to learn the rest by yourselves.

Some terminology

Shell, n.: a computer **program** which exposes an *operating system's services* to a human user or other programs (source: Wikipedia)[2].

Shell, n.: the programming **language** that a text-based shell implements[3].

Shell, n.: a **terminal** emulator.

[2]Some shells (*sensu lato*) are graphical; in this course we mean **text shells only**.
[3]Graphical shells are typically not programmable.

Shell scripting, n.: the art of **writing programs** using **a shell language**.

Script, n.: a program, usually in a (high-level), **interpreted** language (*e.g.* Python, R, Perl, Ruby, JavaScript, and (of course) shells, but *not* C, Java, Rust...)

Which Shell?

- GNU Bash is the default shell on most Linux distributions, and it's available on MacOS X. **It's the one we'll use here.**
- But there are others[4], both older (Bourne, Korn, ...) and newer (Fish, Nu, ...).
- Shells vary in how widespread they are and what features they offer, as well as in their licenses (some proprietary, some open-source)

Warning: generally, code written for shell X **won't work** in shell Y.
$\Rightarrow$ It matters which one we use.

[4] `$ cat /etc/shells` shows you which shells are available on your system.

Course Approach

- The only way to learn programming is, well, to program...
- However, a solid grasp of fundamentals is essential.

We'll start with **fundamentals**, mixing theory and exercises in the (interactive) shell.

Once we have this under our belt, we'll move to a **coding project** where we develop an actual (reasonably) useful script for bioinformatics. New notions will be introduced as we gradually improve the script.

Disclaimer

I may simplify things a little for the sake of brevity and/or clarity.

For example, I might say:

“Word splitting occurs on unquoted expansions”.

instead of:

“Word splitting occurs on the result of unquoted expansions, except in assignments, unless assigning to an array”.

Which would be closer to reality (but still not entirely accurate).

Part I - Below the Surface

WARNING: Lots of Theory Ahead

- There is no avoiding the material presented below if we want to understand how the shell works…
- … and we need a deeper understanding for scripting than for interactive use.
- Arguably most hair-pulling bugs[5] come from our[6] not-so-complete understanding of what the shell actually does with our input.
- **Don't** try to learn all this material - rather, be aware of it.
- **Do** follow along in your terminal!

---

[5] E.g. when you yell at the computer and threaten it with defenestration…

[6] I include myself here, of course

What the Shell does for Us

At the very least, a shell allows users to **launch programs**.
But shells do much more than that...

To illustrate that, let's try the most bare-bones shell of all:

- demo: **`rush`** - the **R**ather **U**seless **SH**ell.

Rush

Command	Result
`ls -l`	ok
`exit`, Ctrl-D	ok
`ls *.pdf`	doesn't work
`grep Bash description.md > result`	doesn't work
`ls | wc -l`	doesn't work
`cd`	doesn't work
`echo $PATH`	doesn't work
completion, history, arrows	don't work
`peak=Matterhorn; echo $peak`	don't work

What the Shell Does behind the Scenes

Besides launching programs, a modern shell, among other functionalities:

- Provides variables.
- Can do arithmetic.
- Performs filename globbing (`*.fasta`, etc.).
- Allows redirection of I/O (`>`, `<`, `|`, etc.).
- Implements a history mechanism.
- Supports auto-completion (`<TAB>`).
- Has flow control structures.

→ Most of the power of Bash comes from the above features.

For Programming

Not all of the above are relevant to programming, however.

We'll focus on:

- Variables and how to use them to store, retrieve, and process data.
- Input and output.
- Control Structures ("compound commands").

Because these aspects are the most relevant.

Shell Operation: The Gist

Broadly speaking, the shell does the following:

1. Reads input (terminal, script, `-c`).
2. Splits the input into *tokens*.
3. Parses tokens into commands.
4. Performs expansions (arithmetic, parameters, etc.).
5. Removes quotes.
6. Performs redirections.
7. Executes the commands.
8. Goes back to pt. 1.

We'll survey 2-5. See the manual for details.

Step 2: Tokenizing

Divide the following sentence into its constituent words:

Alas, poor Yorick! I knew him, Horatio.[7]

[7] W. Shakespeare, *Hamlet*, Act V scene 1

Step 2: Tokenizing

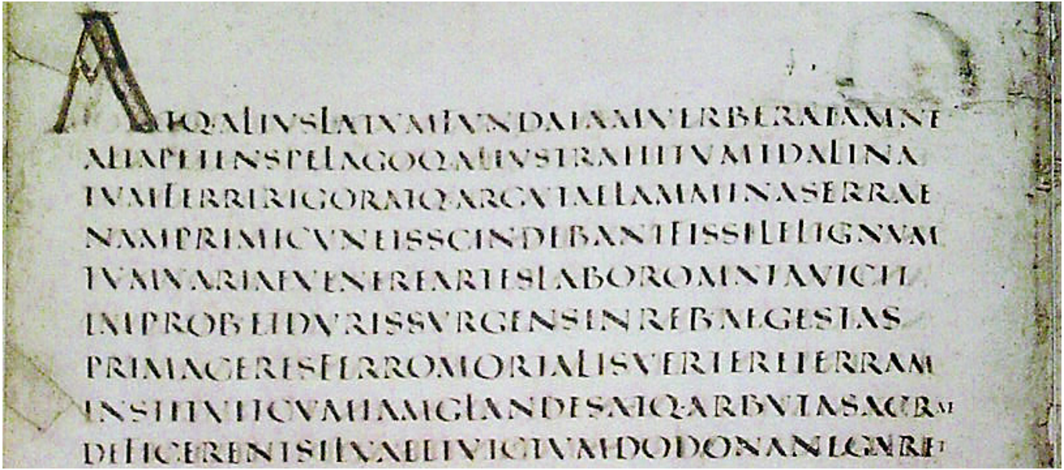
Divide the following sentence into its constituent words:

Alas, poor Yorick! I knew him, Horatio.[7]

- We do this effortlessly, without realizing our brain does any work.
- Exceptions: foreign languages, hard-to-read texts.

[7] W. Shakespeare, *Hamlet*, Act V scene 1

Scriptio continua has no spaces or punctuation…
→ it's now obvious that this requires work.



With spaces and punctuation, it's much better:

atque alius latum funda iam uerberat amnem
alta petens, pelagoque alius trahit umida lina.
tum ferri rigor atque argutae lammina serrae
(*nam primi cuneis scindebant fissile lignum*),
tum uariae uenere artes. labor omnia uicit
improbus et duris urgens in rebus egestas.
prima Ceres ferro mortalis uertere terram
instituit, cum iam glandes atque arbuta sacrae
deficerent siluae et uictum Dodona negaret.

Publius Vergilius Maro, *Georgica*, Liber I, 141-149

This, very broadly speaking, is **tokenizing**.

Tokenizing in Bash

- **Tokens** are separated according to **metacharacters**: whitespace (`space`, `tab`, `newline`) or any of `|` `&` `;` `(` `)` `<` `>`
- Whitespace *separates* tokens (hence is never *part* of tokens).
- Tokens are either wholly non-whitespace metacharacters (**operators**) or non-metacharacters (**words**).
- Metacharacters can be made literal by *quoting* (more on that in a few slides).

Tokenizing - Demo

The **tokenizing procedure** looks like this:

1. Split on whitespace.
2. Split the resulting tokens on metacharacter – non-metacharacter boundaries.

Examples:

```
ls -l | wc -l >> count
ls -l|wc -l>>count
ls-l|wc-l>>count
```

Tokenizing - Examples

```
ls                          # 1 token: 'ls'
ls -l                       # 2 tokens: 'ls' and '-l'
ls -l > list                # 4 tokens
ls -l >list                 # idem
echo $SHELL                 # 2 tokens
grep x < file >> out &      # 7 tokens
grep x<file>>out&           # idem
grepx<file>>out&            # 6 tokens - oops!
```

The **#** character is special: anything between it and the end of the line is a **comment**, and is ignored by the shell.

Literal Characters

These don't work...

```
cat data/my file.txt                # WRONG!
echo it is a fact that 3 > 2        # Surprise!
```

... because the space and > are metacharacters, but here we want them to stand for themselves: that is, to be *literal*.

Quoting

Quoting removes any special meaning of characters[8]. The main forms are:

- `\` (backslash): the next character is literal (except at end of line: continuations).
- `''` (single quotes): all characters between `''` are literal (including `\`, so cannot include `'`).
- `""` (double quotes): all characters between `""` are literal except `$`, `` ` ``, and `\` (only before `"`, `\`, `$`, `` ` ``).

[8]There are special characters other than metacharacters, *e.g.* `$`, `*`, `?` `!`, etc.

Quoting - Showing Arguments

We can use the following function[9] to look[10] at arguments. It prints each one on a separate line (type this in your terminal):

```
function showa() {
  printf "%d args\n" "$#"
  printf "%s\n" "$@"
}

# Can be written in shorter form:
showa(){ printf "%d args\n" "$#"; printf "%s\n" "$@";}
```

[9] We'll study functions on day 3.

[10] Imperfectly, because *e.g.* expansion, redirection etc. still happen normally

Quoting - Examples

```
cat > myfile     # Ok
cat > my file    # WRONG
showa my file
showa my\ file  # 1st form
showa 'my file' # 2nd form
showa "my file" # 3rd form
```

Quoting - ' ' vs " "

The important difference between `''` and `""` is that the latter allow *expansions* to occur (because $ retains its special status):

```
name=Bond
echo "my name is $name"  # Expansion
# -> my name is Bond

echo 'my name is $name'  # No expansion
# -> my name is $name
```

For this reason, single quotes are sometimes called “hard” quotes, and double quotes “soft” quotes.

Special case: expansion occurs if the single quotes are inside double quotes (because the single quotes then lose their

Practice

$\rightarrow$ **Exercise 1.3** - tokenizing and quoting.

Step 3: Parsing Commands

Natural language analogy: recognizing *grammatically correct sentences*. This assumes words have been properly identified:

Utterance	Status
myho vercr afti sful lofe els	words wrong, grammar *undefined*
hovercraft is my eels of full	words ok, grammar wrong
my hovercraft is full of eels	words ok, grammar ok

Parsing by the Shell

The shell doesn't parse sentences but **commands**[11].:

- **Simple commands:** we already known this, e.g.
  - `ls -l > out # redirections allowed`
  - `A=1         # also a simple command`
- **Pipelines:** one *or more* **commands** joined by `|` or `|&`, connecting I/O streams (already known)
- **Lists:** one *or more* pipelines joined by `;` `&` `&&` or `||`, with sequential, background and conditional execution
- **Compound commands:** lists structured by flow control keywords (`if`, `while`, etc.)

[11] For details about the grammar, see its official specification

Examples of Lists

```
long_job &  # Command sent to background.

set -b; sleep 1 & sleep 5 & sleep 8 &
mkdir new ; cd new                        # optimistic
mkdir new && cd new                       # cautious
mkdir /root/stuff && cd /root/stuff       # 1 err msgs
mkdir /root/stuff ; cd /root/stuff        # 2 err msgs
grep skywalker /etc/passwd && echo "found" || echo "not fo
```

Compound Commands

More on this later, but just to fix ideas:

```
if [ ! -d mydir ]; then mkdir mydir; fi
```

This is a *conditional statement* built around the simple commands `[ ! -d mydir ]` and `mkdir mydir`.

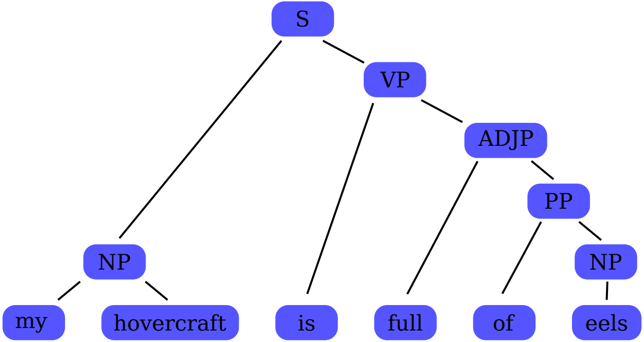
→ Compound commands start and end with *matching keywords* (`if` … `fi`, `{` … `}`, etc.)

In a Nutshell

Compound commands are made of lists, which are made of pipelines, which are made of simple commands… except that a compound command is *itself* a simple command: the grammar is *recursive*.

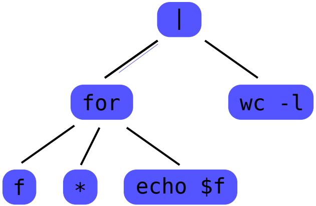
```
for f in *; do echo $f; done | wc -l
<--   compound command   -->
<--    simple command    -->   <-- spl cmd -->
<--        p  i  p  e  l  i  n  e        -->
```

Parsing a sentence converts a linear sequence of words into a tree-like structure:



The tree's structure is constrained by the grammar of English.

The same goes for Bash:[12]



If a command is syntactically wrong, **it does not parse**.

[12]And all other programming languages, for that matter

Practice

$\longrightarrow$ **Exercise 1.4** - parsing.

Step 4: Expansions

After parsing tokens into commands come the **expansions**, *i.e.* the **replacement of expressions with values**, in this order:

1. Brace expansion: `{1..10}`, etc.
2. Left to right:
   - Tilde expansion, *e.g.* `~/Desktop`
   - Parameter expansion: `$USER` & similar
   - Command substitution: `$(date)`, etc.
   - Arithmetic expansion, *e.g.* `$((2+4))`
   - Process substitution: `<(cmd)`
3. Word splitting (different from token splitting!)
4. Filename expansion (“globbing”: `*.txt`)

Brace Expansion

Used to generate sets of strings based on:

- Comma-separated strings: `file_{A,B,C}.txt` $\rightarrow$ `file_A.txt` `file_B.txt` `file_C.txt`
- A sequence: `sample_{1..9}` $\rightarrow$ `sample_1` ... `sample_9`

This is **not** the same as file globbing: the generated strings do not have to be the names of existing files.

Brace Expansion - Examples

```
echo {1..100}  # E.g. in loops (see below).
echo {a..j}    # Works on chars.
echo {10..1}   # Works in reverse.

# Create a project tree (note nesting)
mkdir -p my_project/{src,doc/{mail,ref},data}

# {} at same level -> ~ Cartesian product
echo {A..D}{1..3}
```

Tilde Expansion

This is the well-known replacement of `~` by directories:

Tilde expression	Expansion
`~`	`$HOME`
`~alice`	Alice's home, probably `/home/alice`
`~+`	`$PWD`

… and a few others that address the directory stack; we won't say more about them because they're mostly relevant for interactive use.

Parameter Expansion

An unquoted **$** followed by a parameter name is replaced by the parameter's value

```
place=Rovaniemi
echo $place
# -> Rovaniemi

echo "I'm off to $place"
# -> I'm off to Rovaniemi
```

There is **a lot** more to parameter expansion than this. We'll come back to it later on.

Command Substitution

`$(...)` substitutes the output of a command[13]

```
echo "Today is $(date -I)"
dirsize=$(du -s .)
nb_files=$(ls | wc -l)
```

An older form uses *backticks*:

```
echo "it is now `date`"
```

They're the same, but the modern form is easier to *nest*:

```
parent_dirname=$(basename $(dirname $PWD))
```

[13] Not to be confused with arithmetic expansion, `$((...))`.

Arithmetic Expansion

This doesn't work as expected:

```
a=2; b=3; echo $a+$b
```

This does:

```
a=2; b=3; echo $((a+b))  # Note: no $ needed
```

The expression between `$((...))` is evaluated using shell arithmetic - more about this later.

Process Substitution

Replace a filename argument with the output of a command.

Example: What items are common to two lists?

```
sort spc-list-1.txt > spc-list-1-sorted.txt
sort spc-list-2.txt > spc-list-2-sorted.txt
comm spc-list-1-sorted.txt spc-list-2-sorted.txt
rm spc-list-1-sorted.txt spc-list-2-sorted.txt
```

With `<(...)`, the comparison can be done *on the fly*:

```
comm <(sort spc-list-1.txt) <(sort spc-list-2.txt)
```

→ No temporary files, possibly faster.

Word Splitting

The **results** of **expansions** that **did not occur within double quotes** then undergo *word splitting*.

```
name='Ursus arctos'
showa '$name' # '': no expansion at all
showa "$name" # "": expansion, NO word splitting
showa $name   # expansion, word splitting
```

Why split on words?

Historical aside

- Early shells had no types other than strings.
- To store a *list* of values, programmers used whitespace:

```
dirs='bin doc new'
```

- But without word splitting we now have a problem:

```
# Try this in Zsh, which doesn't word-split
ls -ld $dirs
#  ls: cannot access 'bin doc new':
#  No such file or directory
```

Solutions (sort of)

- Bourne shell (`sh`, 1976): word-split unless within `""`.
- Korn shell (`ksh`, 1983): new data type (arrays) for lists, otherwise same behaviour[14]; syntax for "all elements" rather clunky: `"${array[@]}"`
- Bash, v. 2 (1996): same as `ksh`.
- Zsh (`zsh`, 1991): prefer arrays for lists, word-splitting only if required[15], cleaner syntax for all elements.

[14] Let's not break existing code

[15] If this breaks existing code, so be it!

IFS - Internal Field Separator

Word splitting uses the characters in the the **`IFS` - the Internal Field Separator**[16] (by default, `<space><tab><newline>`)[17] as word delimiters.

The value of `IFS` can be changed:

```
line='gene_name,seq_len,mol_wt'    # CSV-like
showa $line                        # 1 field
IFS=','; showa $line; unset IFS    # 3 fields
```

[16]Hence the alternative (and better) term *field* splitting.

[17]Contrary to splitting into tokens, which uses whitespace and metacharacters.

IFS - Internal Field Separator (continued)

If `$IFS` is null, no splitting is done. If unset, the above default is used.

→ To reset the value of `IFS`: `unset IFS`.

Filename Expansion ("globbing")

Words that contain **unquoted** `*`, `?`, or `[` are pattern-matched against the files in the current directory.[18]

Wildcard	meaning
`?`	any 1 character
`*`	any string, including ''
`[...]`	any character in the class

There are predefined classes, *e.g.* for letters, digits, punctuation, etc.

[18] This is done after parameter expansions so that globs can be stored in variables, *e.g.* `glob='*.pdf *.docx'; ls $glob`.

```
# No quotes -> globbing occurs.
ls *.pdf

# Quotes -> no globbing occurs.
ls '*.pdf' "*.pdf" \*.pdf
echo "$glob" # Still quoted -> no globbing

# When a variable is set, quotes not needed
glob=*.pdf
```

In the last example, parameter expansion of `"$glob"` yields `"*.pdf"` (not `*.pdf`), so no globbing occurs. The quotes are removed in the next stage.

Quote Removal

After all expansions have been performed, quotes (and backslashes) are removed (unless they are quoted or result from an expansion):

```
echo 'recA' "dnaK"   # ',"  removed
echo d\'Artagnan     # \ removed
echo "a 'quote"      # ' retained: quoted
q='"'; echo "$q"     # middle " retained: expansion
```

This is why quotes usually disappear when we use `echo` or `showa`.

Expansions can be Mixed

```
pattern=*.md
echo "Markdown files:" $(ls $pattern)
```

This mixes parameter expansion, command substitution, word splitting, and filename globbing.

Practice

→ **Exercise 1.5** - parameter expansion.

→ **Exercise 1.6** - brace expansion and command substitution.

→ **Exercise 1.7** - process substitution.

Step 5: Redirection

After all the expansions phase come the **redirections**, in which the operations specified by >, >>, |, < etc. are performed, and the corresponding operators and arguments removed from the expanded list of words:

```
# Output of ls goes into list.txt (destructive!)
# Use >> to append
ls > list.txt
```

Note: Redirection can also be done from *within* the script:

```
exec < my_input.txt
# From now on, stdin comes from my_input.txt
```

Here Documents: <<

```
cat <<END
# Everything up to END goes to the input of cat;
# The end token can be any word, not just END
# Quoting prevents expansion.
END
```

Useful to store some multiline output within the script - see `src/welcome.sh`.

Here Strings: <<<

Useful for small inputs that can fit on the command line, *e.g.* measuring the length of strings:

```
# /!\ Includes newline!
wc -l <<< CATCGACATGCA
```

Step 6: Command Execution

Finally, after all these tokenization, parsing, various expansions, quote removal, and redirection steps, the command is ready to be launched. The shell requests the kernel to do so.

Recap

The main stages of input processing:

1. Tokenizing
2. Parsing into commands
3. Expansions
   3.1 `{}`
   3.2 `~`, `${}`, `$()`, `$(())`, `<()`
   3.3 Word splitting
   3.4 Globbing
4. Quote Removal
5. Redirections

A Brief Point About Pure and Delegation Styles

- “pure” and “delegation” styles were introduced earlier.
- … but to make the following points I needed some recently-introduced notions.

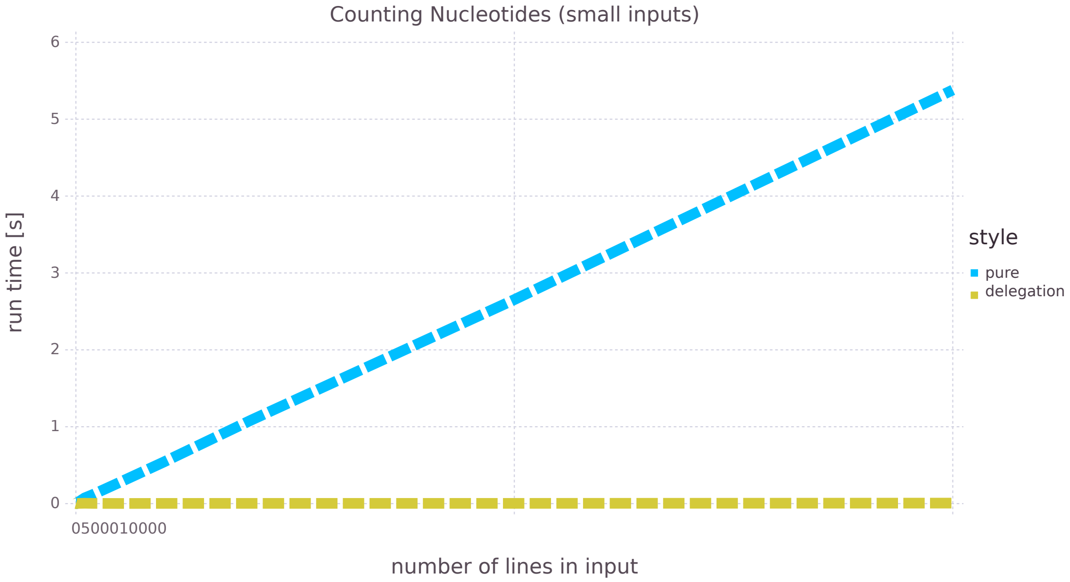
Task: count the nucleotides in a sequence file.

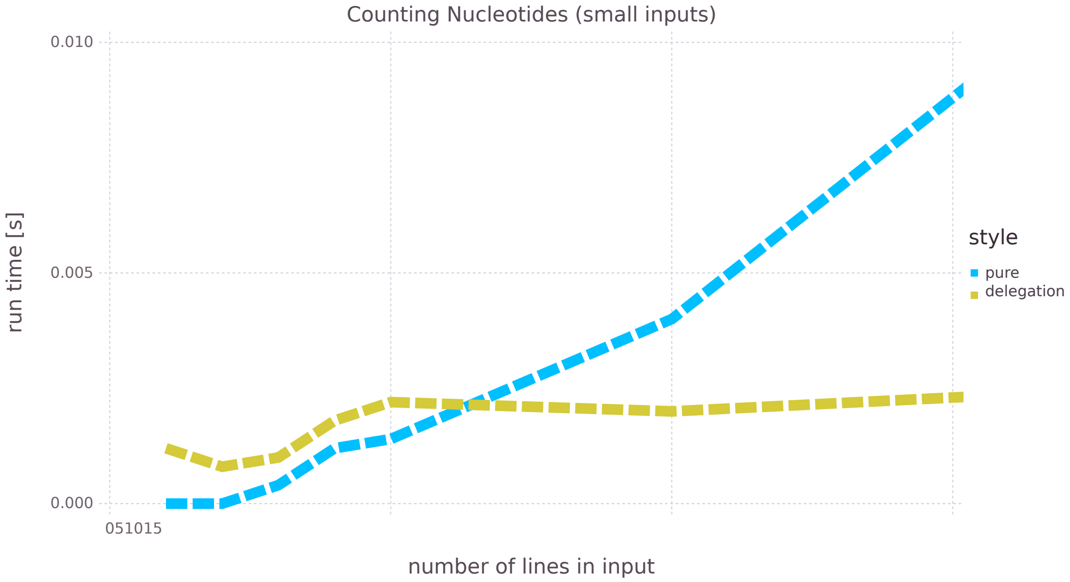
```
# Pure style: all in Bash
./ntcount-pure < genome

# Delegation style: calls tr, tee etc.
./ntcount-deleg < genome
```

The scripts could hardly be more different (see code).

Now let's measure run times, using increasing numbers of lines.

$\rightarrow$ and the winner is...


Counting Nucleotides (small inputs)
run time [s]
6
5
4
3
2
1
0
0500010000
number of lines in input
style
pure
delegation


Counting Nucleotides (small inputs)
0.010
0.005
0.000
051015
run time [s]
number of lines in input
style
pure
delegation

Interpretation

- External programs are typically (much) faster than Bash.
- *Launching* an external program has a **cost** (overhead) .
- The "delegation" style launches programs, the "pure" style doesn't.
- The extra speed more than compensates for the overhead, *except for very small inputs* and/or numerous tasks.