UNIX III: Shell programming and data processing

In UNIX I you wrote your first shell script, and in UNIX II you learned wildcards, redirection (`>`, `>>`, `2>`), pipes (`|`), `grep` and compressed files. This lecture puts those pieces together:

- **Part 1 - Programming logic in bash:** variables, testing conditions with `if`, loops, functions, arrays, and how to write scripts that stop when something goes wrong. We finish with a script that runs a step on every sample in a sequencing project.
- **Part 2 - Data processing with pipes** is a separate lab / reading: UNIX III lab: Data processing with pipes covers the classic UNIX tools (`grep`, `cut`, `sort`, `uniq`, `tr`, `paste`, `join`, `sed` and `awk`) for answering questions about tables of data, like a genome annotation or BLAST results, without writing a program.

Bash is good at running programs and gluing them together (loop over files, run a tool on each, check that it worked). When the logic gets complicated, or you need real statistics, it is time to switch to Python or R, which we cover later in the course.

Setup

Log in to the cluster and make a folder for today:

```
mkdir -p ~/bigdata/gen220/unix3
cd ~/bigdata/gen220/unix3
export LC_ALL=C
```

`export LC_ALL=C` sets the *locale* (language settings) to plain "C" rules for this session. It mostly changes how `sort` orders text (uppercase before lowercase, character by character), and it makes `sort` and `join` work together reliably. All the outputs in this lecture were made this way, so set it if you want your results to match.

Download the example data. The first four files are in the biodataprog/GEN220 repository and the yeast genome files are in biodataprog/GEN220_data. (The `for` loop runs `curl` once per file name; loops are explained below.)

```
mkdir -p data
cd data
URL=https://raw.githubusercontent.com/biodataprog/GEN220/master/data
for FILE in codon_table.txt numbers.txt \
            rice_random_exons.bed yeast_orfs-to-chr1.FASTA.tab; do
    curl -sSLO "$URL/$FILE"
done
URL=https://github.com/biodataprog/GEN220_data/raw/main
curl -sSLO "$URL/genome/S_cerevisiae.gff3.gz"
curl -sSLO "$URL/genome/S_cerevisiae.ORFs.fasta.gz"
curl -sSLO "$URL/tabular/threatened-species.csv.gz"
ls
cd ..
```

```
S_cerevisiae.ORFs.fasta.gz
S_cerevisiae.gff3.gz
codon_table.txt
numbers.txt
rice_random_exons.bed
threatened-species.csv.gz
yeast_orfs-to-chr1.FASTA.tab
```

(You can also get everything with `git clone https://github.com/biodataprog/GEN220_data.git`, which puts the genome files in `GEN220_data/genome/`.)

File	What it is
`codon_table.txt`	the genetic code: codon, one-letter amino acid, amino acid name (tab-separated)
`numbers.txt`	100 numbers between 0 and 99, one per line
`rice_random_exons.bed`	1000 rice exons in BED format: chromosome, start, end
`yeast_orfs-to-chr1.FASTA.tab`	yeast genes searched against chromosome I, BLAST-style tabular output
`S_cerevisiae.gff3.gz`	yeast genome annotation (GFF3) from SGD
`S_cerevisiae.ORFs.fasta.gz`	DNA sequences of all yeast ORFs
`threatened-species.csv.gz`	IUCN Red List species and their threat category (CSV)

Part 1: Programming logic in bash

Variables

A variable stores a value under a name. Create it with `NAME=value` (**no spaces** around the `=`) and get the value back with `$NAME`:

```
GENE="YAL001C"
echo "$GENE"
echo "The gene is $GENE"
echo 'The gene is $GENE'
```

```
YAL001C
The gene is YAL001C
The gene is $GENE
```

- **Double quotes** `"..."` group words together but still replace `$VARIABLES` (and `$(commands)`) with their values.
- **Single quotes** `'...'` keep everything exactly as typed. Use them when you want a literal `$`, for example in `awk` programs later in this lecture.
- `GENE = "YAL001C"` (with spaces) gives the error `GENE: command not found`: bash thinks you are running a program called `GENE`.
- Variable names can contain letters, numbers and `_`, and can't start with a number. Names are case sensitive: `$gene` and `$GENE` are different variables.

Always quote your variables

Without quotes, bash splits a variable's value on spaces *before* running the command. File names with spaces then break:

```
touch "my reads.txt"
FILE="my reads.txt"
ls $FILE
ls "$FILE"
rm "$FILE"
```

```
ls: cannot access 'my': No such file or directory
ls: cannot access 'reads.txt': No such file or directory
'my reads.txt'
```

Unquoted, `ls` was given two file names, `my` and `reads.txt`. Quoted, it got one. Get into the habit of writing `"$FILE"`. (Better still, don't put spaces in your file names.)

Braces: `${NAME}`

When a variable name is followed by letters, numbers or `_`, bash can't tell where the name ends. Put braces around the name:

```
SAMPLE=wt
echo "$SAMPLE_1.fastq.gz"
echo "${SAMPLE}_1.fastq.gz"
echo "$SAMPLE.fastq.gz"
```

```
.fastq.gz
wt_1.fastq.gz
wt.fastq.gz
```

The first line looked for a variable called `SAMPLE_1`, which doesn't exist, so it was replaced by nothing - without any error. A `.` can't be part of a name, so `$SAMPLE.fastq.gz` works, but `${SAMPLE}` is never wrong.

Environment variables

Some variables are already set when you log in. Run `env` to see them all. A few useful ones:

```
echo "My home folder is $HOME and my user name is $USER"
```

```
My home folder is /rhome/username and my user name is username
```

`$PATH` is the list of folders searched for programs (see UNIX I); `module load` works by adding folders to it. A variable you create is only seen by your current shell; `export NAME` makes it visible to programs you start, which is why we wrote `export LC_ALL=C`.

Command substitution and arithmetic

`$(command)` runs a command and substitutes its output. Save it in a variable to use later:

```
NCODONS=$(wc -l < data/codon_table.txt)
NVAL=$(grep -c Valine data/codon_table.txt)
echo "The table has $NCODONS codons and $NVAL of them code for valine"
```

```
The table has 64 codons and 4 of them code for valine
```

(`wc -l < file` prints just the number; `wc -l file` also prints the file name. You may also see the older backtick form `` `command` ``, which does the same thing.)

Bash does **integer** arithmetic inside `$(( ))`. Variables inside don't need a `$`:

```
echo $(( 7 + 3 )) $(( 7 * 3 )) $(( 2 ** 10 ))
echo $(( 7 / 2 ))    # integer division throws away the remainder
echo $(( 7 % 2 ))    # % is the remainder ("modulo")
LINES=400
READS=$(( LINES / 4 ))    # a FASTQ file has 4 lines per read
echo "$READS reads"
```

```
10 21 1024
3
1
100 reads
```

Bash can't do decimals. For those, use `awk` (covered in the data processing lab):

```
awk 'BEGIN { print 7 / 2 }'
```

```
3.5
```

Older scripts use `expr $a + 1` or `let`; `$(( ))` replaces both.

Working with file names

Pipelines constantly turn one file name into another: `reads/wt_1_R1.fastq.gz` becomes sample name `wt_1`, which becomes output `results/wt_1.bam`. Bash can cut pieces off the start or end of a variable's value:

Syntax	Removes	Example result for `f=/data/reads/wt_1_R1.fastq.gz`
`${f##*/}`	everything up to the last `/`	`wt_1_R1.fastq.gz`
`${f%/*}`	the last `/` and everything after it	`/data/reads`
`${f%.fastq.gz}`	`.fastq.gz` from the end	`/data/reads/wt_1_R1`
`${f%_R1.fastq.gz}_R2.fastq.gz`	replace the end	`/data/reads/wt_1_R2.fastq.gz`

The memory aid: `#` is on the left of `$` on the keyboard and removes from the left (start); `%` is on the right and removes from the right (end). A single `#` or `%` removes the *shortest* match, doubled (`##`, `%%`) removes the *longest* match:

```
f=/bigdata/gen220/shared/reads/wt_1_R1.fastq.gz
NAME=${f##*/}
echo "$NAME"
echo "${NAME%.fastq.gz}"      # shortest match of .fastq.gz from the end
echo "${NAME%%_*}"            # longest match of _* from the end
echo "${NAME#*_}"             # shortest match of *_ from the start
echo "${NAME/_R1/_R2}"        # replace the first _R1 with _R2
```

```
wt_1_R1.fastq.gz
wt_1_R1
wt
1_R1.fastq.gz
wt_1_R2.fastq.gz
```

The `basename` and `dirname` commands do the most common jobs and are easier to read. `basename` can also remove an ending:

```
basename "$f"
basename "$f" _R1.fastq.gz
dirname "$f"
```

```
wt_1_R1.fastq.gz
wt_1
/bigdata/gen220/shared/reads
```

Exit codes: did it work?

Every command finishes with an **exit code**: `0` means success and anything else means failure. The exit code of the last command is in `$?`:

```
ls data/codon_table.txt
echo "exit code: $?"
ls data/no_such_file.txt
echo "exit code: $?"
grep -c Unobtainium data/codon_table.txt
echo "exit code: $?"
```

```
data/codon_table.txt
exit code: 0
ls: cannot access 'data/no_such_file.txt': No such file or directory
exit code: 2
0
exit code: 1
```

`grep` exits with `1` when it finds no matches. That is useful for testing, but it can surprise you in scripts (see "Writing robust scripts").

`&&` runs the next command only if the first one **succeeded**; `||` runs it only if the first one **failed**. `grep -q` ("quiet") prints nothing and just sets the exit code:

```
mkdir -p results && echo "made the results folder"
grep -q Stop data/codon_table.txt && echo "the table has stop codons"
grep -q Selenocysteine data/codon_table.txt || echo "no selenocysteine in this table"
```

```
made the results folder
the table has stop codons
no selenocysteine in this table
```

Making decisions: `if`

`if` runs a command and checks its exit code. It can be any command:

```
if grep -q Stop data/codon_table.txt; then
    echo "found stop codons:"
    grep Stop data/codon_table.txt
fi
```

```
found stop codons:
TAA *   Stop
TAG *   Stop
TGA *   Stop
```

Most often the command is a **test** written inside `[[ ]]`. You saw the older single-bracket `[ ]` in UNIX I, which works too, but in bash scripts prefer `[[ ]]`: it handles empty variables and spaces in values more safely, and it understands `&&`, `||` and wildcard patterns. You need **spaces** inside the brackets: `[[ $X == 1 ]]`, not `[[$X==1]]`.

Comparing text

```
GENE="YAL001C"
if [[ $GENE == "YAL001C" ]]; then
    echo "found TFC3"
fi
if [[ $GENE == YAL* ]]; then           # patterns work on the right of == (no quotes)
    echo "$GENE is on the left arm of chromosome I"
fi
FILE=reads.fastq.gz
if [[ $FILE != *.gz ]]; then
    echo "$FILE is not compressed"
else
    echo "$FILE is compressed"
fi
```

```
found TFC3
YAL001C is on the left arm of chromosome I
reads.fastq.gz is compressed
```

`-z "$X"` is true if `X` is empty, `-n "$X"` if it is not empty.

Comparing numbers

Numbers need different operators: `-eq` (equal), `-ne` (not equal), `-lt` (less than), `-le` (less or equal), `-gt` (greater than), `-ge` (greater or equal). Use `elif` ("else if") to check several cases in order:

```
NREADS=250
if [[ $NREADS -lt 100 ]]; then
    echo "too few reads"
```

```
elif [[ $NREADS -lt 1000 ]]; then
    echo "low coverage: $NREADS reads"
else
    echo "ok: $NREADS reads"
fi
```

```
low coverage: 250 reads
```

Careful: `<`, `>` and `==` inside `[[ ]]` compare **text**, in dictionary order, not numbers. As text, "10" comes before "9" because "1" comes before "9":

```
if [[ 10 -gt 9 ]]; then echo "-gt: 10 is bigger than 9"; fi
if [[ 10 > 9 ]]; then
    echo "10 > 9"
else
    echo "> compares text: \"10\" sorts before \"9\""
fi
if (( 10 > 9 )); then echo "(( )) compares numbers"; fi
```

```
-gt: 10 is bigger than 9
> compares text: "10" sorts before "9"
(( )) compares numbers
```

So use `-lt`, `-gt` etc. inside `[[ ]]`, or use `(( ))`, which does arithmetic and understands `<`, `>`, `==` as numbers.

Testing files

These tests are how scripts check that inputs exist before starting and skip work that is already done:

Test	True if
`-e FILE`	FILE exists (file, folder, anything)
`-f FILE`	FILE exists and is a regular file
`-d FILE`	FILE exists and is a directory (folder)
`-s FILE`	FILE exists and is not empty (size > 0)
`-r FILE`	FILE exists and you can read it
`-x FILE`	FILE exists and you can execute it
`A -nt B`	file A is newer than file B (or B doesn't exist)

Combine tests with `&&` (and), `||` (or) and `!` (not):

```
echo "hello" > notes.txt
touch empty.txt
[[ -d data ]] && echo "data is a folder"
[[ -f data ]] || echo "data is not a regular file"
[[ -s empty.txt ]] || echo "empty.txt is empty"
[[ -s notes.txt && -r notes.txt ]] && echo "notes.txt is not empty and we can read it"
[[ ! -e results/wt_1.bam ]] && echo "wt_1.bam has not been made yet"
touch -d 2026-09-01 old.txt       # make a file with an old date
if [[ notes.txt -nt old.txt ]]; then
    echo "notes.txt is newer than old.txt"
fi
rm notes.txt empty.txt old.txt
```

```
data is a folder
data is not a regular file
empty.txt is empty
notes.txt is not empty and we can read it
```

```
wt_1.bam has not been made yet
notes.txt is newer than old.txt
```

`-nt` is how you decide whether something must be re-run, e.g. `if [[ genome.fa -nt genome.fa.bwt ]]` means the genome changed since it was indexed.

Choosing between many options: `case`

`case` compares one value against a list of patterns and runs the first one that matches. `|` means "or", `*)` matches anything (the default), and each choice ends with `;;`:

```
for FILE in genome.fa reads.fastq.gz genes.gff3 notes.docx; do
    case $FILE in
        *.fa|*.fasta)  echo "$FILE: FASTA sequence file" ;;
        *.fastq.gz)    echo "$FILE: compressed FASTQ reads" ;;
        *.gff|*.gff3)  echo "$FILE: genome annotation" ;;
        *)             echo "$FILE: not sure what this is" ;;
    esac
done
```

```
genome.fa: FASTA sequence file
reads.fastq.gz: compressed FASTQ reads
genes.gff3: genome annotation
notes.docx: not sure what this is
```

Loops

`for` loops

A `for` loop runs the same commands once for each item in a list, putting the item in a variable:

```
for GENE in TFC3 VPS8 EFB1; do
    echo "gene: $GENE"
done
```

```
gene: TFC3
gene: VPS8
gene: EFB1
```

The most useful list is a wildcard (glob, see UNIX II) that matches files. The shell expands `*.fa` to the matching file names, sorted, and handles spaces in names correctly:

```
mkdir -p loopdemo
cd loopdemo
touch A.fa B.fa "C D.fa"
for f in *.fa; do
    echo "found: $f"
done
```

```
found: A.fa
found: B.fa
found: C D.fa
```

Don't loop over `$(ls ...)`. You will see `for f in $(ls *.fa)` in many old scripts (including earlier versions of this class!). The output of `ls` is split on spaces, so file names with spaces break, and it is slower for no benefit:

```
for f in $(ls *.fa); do
    echo "found: $f"
done
```

```
found: A.fa
found: B.fa
```

```
found: C
found: D.fa
```

When nothing matches, bash leaves the pattern as it is, so the loop runs once with the literal text `*.bam`:

```
for f in *.bam; do
    echo "found: $f"
done
```

```
found: *.bam
```

Guard against that by skipping names that don't exist (`continue` jumps to the next item), or by turning on the `nullglob` option, which makes a pattern with no matches expand to nothing:

```
for f in *.bam; do
    [[ -e $f ]] || continue
    echo "found: $f"
done
shopt -s nullglob
for f in *.bam; do
    echo "found: $f"
done
shopt -u nullglob
cd ..
rm -r loopdemo
```

Neither loop prints anything.

Looping over numbers

Brace expansion makes a list of numbers or words. `seq` does the same and, unlike braces, works with a variable as the end point. There is also a C-style loop:

```
echo {1..5}
echo {01..10}
echo sample_{A,B,C}.txt
N=4
echo {1..$N}                  # does NOT work: braces happen before variables
seq -s ' ' 1 "$N"             # seq works (-s ' ' puts them on one line)
for (( i = 1; i <= N; i++ )); do
    echo -n "$i "
done
echo
```

```
1 2 3 4 5
01 02 03 04 05 06 07 08 09 10
sample_A.txt sample_B.txt sample_C.txt
{1..4}
1 2 3 4
1 2 3 4
```

`while` loops

A `while` loop repeats as long as its test is true:

```
N=1
while [[ $N -le 3 ]]; do
    echo "round $N"
    N=$(( N + 1 ))
done
```

```
round 1
round 2
round 3
```

Reading a file line by line

The most common use of `while` is reading a file one line at a time. Make a small table of yeast genes and their lengths (`printf` is like `echo` but `\t` becomes a tab and `\n` a new line):

```
printf 'gene\tname\tlength\n' > genes.tsv
printf 'YAL001C\tTFC3\t3483\n' >> genes.tsv
printf 'YAL002W\tVPS8\t3825\n' >> genes.tsv
printf 'YAL003W\tEFB1\t621\n' >> genes.tsv
printf 'YAL005C\tSSA1\t1929\n' >> genes.tsv
while IFS=$'\t' read -r ID NAME LEN; do
    echo "$NAME ($ID) is $LEN bp"
done < genes.tsv
```

```
name (gene) is length bp
TFC3 (YAL001C) is 3483 bp
VPS8 (YAL002W) is 3825 bp
EFB1 (YAL003W) is 621 bp
SSA1 (YAL005C) is 1929 bp
```

How this works:

- `< genes.tsv` at the end, after `done`, feeds the file into the whole loop.
- `read -r ID NAME LEN` reads one line and splits it into the variables: the first column goes in `ID`, the second in `NAME`, and **everything left over** goes in the last variable. `-r` stops `read` from treating `\` as special; always use it.
- `IFS` (the "internal field separator") says what to split on. `IFS=$'\t'` means "split on tabs only" (`$'\t'` is how bash writes a tab character), so a column containing spaces stays in one piece. For a comma-separated file use `IFS=,`.
- Writing `IFS=$'\t'` *in front of* `read`, on the same line, changes it **only for that `read` command**.

That last point matters. Older scripts (and earlier versions of this lecture) set `IFS=,` on a line by itself. That changes how bash splits words for **the rest of the script**, which quietly breaks other commands:

```
LIST="wt_1 wt_2 mut_1"
IFS=,               # don't do this!
for s in $LIST; do echo "sample: $s"; done
unset IFS           # back to normal
for s in $LIST; do echo "sample: $s"; done
```

```
sample: wt_1 wt_2 mut_1
sample: wt_1
sample: wt_2
sample: mut_1
```

Skipping a header line. Our loop also processed the header line. `tail -n +2` prints a file starting from line 2, so pipe that into the loop:

```
tail -n +2 genes.tsv | while IFS=$'\t' read -r ID NAME LEN; do
    echo "$NAME ($ID) is $LEN bp"
done
```

```
TFC3 (YAL001C) is 3483 bp
VPS8 (YAL002W) is 3825 bp
EFB1 (YAL003W) is 621 bp
SSA1 (YAL005C) is 1929 bp
```

One catch: the part of a pipeline after a `|` runs in a separate copy of the shell (a *subshell*), so **variables changed inside a piped loop are lost** when it ends. If you need a value after the loop, use `done < <(tail -n +2 genes.tsv)` instead of the pipe. (`<(command)` lets a command's output be read like a file.)

```
COUNT=0
tail -n +2 genes.tsv | while IFS=$'\t' read -r ID NAME LEN; do
    COUNT=$(( COUNT + 1 ))
done
echo "pipe: counted $COUNT genes"
COUNT=0
while IFS=$'\t' read -r ID NAME LEN; do
    COUNT=$(( COUNT + 1 ))
done < <(tail -n +2 genes.tsv)
echo "process substitution: counted $COUNT genes"
```

```
pipe: counted 0 genes
process substitution: counted 4 genes
```

`break` and `continue`

`continue` skips the rest of this round and goes on to the next item; `break` leaves the loop completely:

```
tail -n +2 genes.tsv | while IFS=$'\t' read -r ID NAME LEN; do
    if [[ $LEN -lt 1000 ]]; then
        echo "skipping short gene $NAME"
        continue
    fi
    if [[ $NAME == SSA1 ]]; then
        echo "found $NAME, stopping"
        break
    fi
    echo "$NAME is long enough"
done
```

```
TFC3 is long enough
VPS8 is long enough
skipping short gene EFB1
found SSA1, stopping
```

Functions

A function gives a name to a group of commands, so you can reuse it. Inside a function, `$1`, `$2`, … are the function's own arguments. `local` keeps a variable inside the function so it doesn't overwrite one with the same name elsewhere in your script:

```
count_lines() {
    local file=$1
    local n
    n=$(wc -l < "$file")
    echo "$file: $n lines"
}
count_lines data/codon_table.txt
count_lines data/numbers.txt
```

```
data/codon_table.txt: 64 lines
data/numbers.txt: 100 lines
```

Define functions near the top of a script, before you use them. A handy one for scripts prints an error message and stops:

```
die() {
    echo "ERROR: $*" >&2    # $* is all the arguments; >&2 sends the message to STDERR
    exit 1
}
```

Arrays

An array holds a list of values in one variable. Items are numbered starting from 0:

```
SAMPLES=(wt_1 wt_2 mut_1)
echo "first sample: ${SAMPLES[0]}"
echo "number of samples: ${#SAMPLES[@]}"
SAMPLES+=(mut_2)                        # add to the end
for s in "${SAMPLES[@]}"; do            # loop over every item
    echo "processing $s"
done
```

```
first sample: wt_1
number of samples: 3
processing wt_1
processing wt_2
processing mut_1
processing mut_2
```

Always write `"${SAMPLES[@]}"` with the quotes: each item stays separate even if it has spaces. Without `[@]`, `$SAMPLES` is only the first item.

You can fill an array from the lines of a file with `mapfile`, or from a wildcard:

```
printf 'wt_1\nwt_2\nmut_1\nmut_2\n' > samples.txt
mapfile -t SAMPLES < samples.txt       # -t removes the newline from each line
echo "read ${#SAMPLES[@]} samples; the last one is ${SAMPLES[-1]}"
TXTFILES=(data/*.txt)
echo "${#TXTFILES[@]} text files: ${TXTFILES[*]}"
```

```
read 4 samples; the last one is mut_2
2 text files: data/codon_table.txt data/numbers.txt
```

Picking the Nth sample out of a list, `${SAMPLES[$i]}`, is exactly what SLURM job arrays do (see UNIX IV).

Writing robust scripts

From here on, save each script shown with `nano`, using the file name given in its first comment line.

A script keeps going after a command fails, unless you tell it not to. Here is a script that tries to go into a folder and clean it up:

```
#!/usr/bin/env bash
# no_safety.sh - what happens when a command fails?
cd data/no_such_folder
echo "Deleting the old results"
# imagine this line was: rm -f *.txt   - in the WRONG folder!
```

```
chmod +x no_safety.sh
./no_safety.sh
echo "exit code: $?"
```

```
./no_safety.sh: line 3: cd: data/no_such_folder: No such file or directory
Deleting the old results
exit code: 0
```

The `cd` failed but the script carried on, in whatever folder it happened to be in, and even reported success. Start every script with **`set -euo pipefail`**:

- `-e` - **exit** as soon as any command fails.
- `-u` - treat using an **unset** (never created) variable as an error. This catches typos like `$OUTDRI` for `$OUTDIR`, which would otherwise silently become empty. (Imagine `rm -rf "$OUTDRI"/*`.)
- `-o pipefail` - a pipeline fails if **any** command in it fails. Normally only the last command counts, so `zcat missing.fastq.gz | wc -l` "succeeds" and prints `0`.

```
#!/usr/bin/env bash
# safety.sh - the same script, but stop at the first error
set -euo pipefail
cd data/no_such_folder
echo "Deleting the old results"
```

```
chmod +x safety.sh
./safety.sh
echo "exit code: $?"
```

```
./safety.sh: line 4: cd: data/no_such_folder: No such file or directory
exit code: 1
```

Two things to know when using `set -e`:

- Commands that "fail" as part of normal use will now stop your script. `N=$(grep -c Unobtainium file)` exits the script when there are no matches, because `grep` exits with 1. Add `|| true` when no match is OK: `N=$(grep -c Unobtainium file || true)`.
- A failing command *inside* an `if` test or before `&&`/`||` does not stop the script; that's how you check for failure yourself.

Check the inputs and explain how to run the script

A good script checks its arguments and input files before doing any work, and prints a **usage** message if something is wrong. Send messages meant for people to **STDERR** with `>&2` (see redirection in UNIX II), so they don't get mixed into the real output if you redirect it to a file, and `exit` with a non-zero code on errors.

```
#!/usr/bin/env bash
# count_seqs.sh - count the sequences in a FASTA file (plain or .gz)
# usage: ./count_seqs.sh FASTA_FILE
set -euo pipefail

if [[ $# -ne 1 ]]; then
    echo "usage: $0 FASTA_FILE" >&2
    exit 1
fi
FASTA=$1
if [[ ! -s $FASTA ]]; then
    echo "ERROR: $FASTA does not exist or is empty" >&2
    exit 1
fi

case $FASTA in
    *.gz) N=$(zcat "$FASTA" | grep -c '^>') ;;
    *)    N=$(grep -c '^>' "$FASTA") ;;
esac
echo -e "$FASTA\t$N"
```

```
chmod +x count_seqs.sh
./count_seqs.sh
```

```
./count_seqs.sh data/nothing_here.fa
./count_seqs.sh data/S_cerevisiae.ORFs.fasta.gz
```

```
usage: ./count_seqs.sh FASTA_FILE
ERROR: data/nothing_here.fa does not exist or is empty
data/S_cerevisiae.ORFs.fasta.gz 6713
```

Debugging: `bash -x` and ShellCheck

`bash -x script.sh` (or `set -x` inside the script) prints each command, after variables have been filled in, just before it runs. Lines starting with `+` are the trace; the others are normal output:

```
bash -x count_seqs.sh data/codon_table.txt
```

```
+ set -euo pipefail
+ [[ 1 -ne 1 ]]
+ FASTA=data/codon_table.txt
+ [[ ! -s data/codon_table.txt ]]
+ case $FASTA in
++ grep -c '^>' data/codon_table.txt
+ N=0
```

The trace shows the script stopped right after `grep` found 0 matches (a codon table isn't a FASTA file!), which is the `set -e` rule above. Adding `|| true` after `grep -c` would fix it.

ShellCheck finds common mistakes in shell scripts: paste your script into the web site, or run `shellcheck myscript.sh` if it is installed (check `module avail shellcheck`, or install it with conda). Here it is on a script with several of the problems from this lecture:

```
#!/usr/bin/env bash
# buggy.sh - count lines in each FASTQ file
for f in $(ls *.fastq.gz); do
    name=$(basename $f .fastq.gz)
    echo "Processing $name"
    zcat $f | wc -l > $name.lines.txt
done
```

```
shellcheck -f gcc buggy.sh
```

```
buggy.sh:3:10: error: Iterating over ls output is fragile. Use globs. [SC2045]
buggy.sh:3:15: note: Use ./*glob* or -- *glob* so names with dashes won't become
    options. [SC2035]
buggy.sh:4:21: note: Double quote to prevent globbing and word splitting. [SC2086]
buggy.sh:6:10: note: Double quote to prevent globbing and word splitting. [SC2086]
buggy.sh:6:23: note: Double quote to prevent globbing and word splitting. [SC2086]
```

(The second message is one long line, wrapped here to fit the page.) `-f gcc` prints one line per problem: file, line number, column, and the message. Plain `shellcheck buggy.sh` prints a longer report with arrows pointing at each problem and a suggested fix ("Did you mean: `zcat "$f" | wc -l > "$name".lines.txt`"). Each message has a code (like SC2086) you can look up on the ShellCheck wiki. Apart from `buggy.sh` and `no_safety.sh` (where ShellCheck warns, correctly, that the `cd` might fail: SC2164), all the scripts in this lecture pass ShellCheck with no warnings.

Putting it together: run a step on every sample

A typical project has a folder of paired-end FASTQ files, two per sample (`NAME_R1.fastq.gz` and `NAME_R2.fastq.gz`), and you need to run the same tool on every sample. Let's write a script for that.

Make some practice data

This script writes tiny FASTQ files for three samples with 3, 5 and 4 read pairs, plus a fourth sample, `mut_2`, whose R2 file is missing, so we can test the error checking. It uses several things from above: a list of `name:number` pairs split with `${PAIR%:*}` and `${PAIR#*:}`, nested loops, and `printf`.

```
mkdir -p ~/bigdata/gen220/unix3/pipeline
cd ~/bigdata/gen220/unix3/pipeline
nano make_test_reads.sh
```

```
#!/usr/bin/env bash
# make_test_reads.sh - write tiny paired FASTQ files to practice on
set -euo pipefail
mkdir -p reads
# sample name : number of read pairs
for PAIR in wt_1:3 wt_2:5 mut_1:4; do
    SAMPLE=${PAIR%:*}
    NREADS=${PAIR#*:}
    for READ in R1 R2; do
        for (( i = 1; i <= NREADS; i++ )); do
            printf '@%s_%d/%s\nACGTTGCAAGGT\n+\nIIIIIIIIIIII\n' "$SAMPLE" "$i" "${READ#R}"
        done | gzip -c > "reads/${SAMPLE}_${READ}.fastq.gz"
    done
done
# a sample where the R2 file is missing, to test our error checking
cp reads/mut_1_R1.fastq.gz reads/mut_2_R1.fastq.gz
ls reads
```

```
chmod +x make_test_reads.sh
./make_test_reads.sh
zcat reads/wt_1_R1.fastq.gz | head -n 8
```

```
mut_1_R1.fastq.gz
mut_1_R2.fastq.gz
mut_2_R1.fastq.gz
wt_1_R1.fastq.gz
wt_1_R2.fastq.gz
wt_2_R1.fastq.gz
wt_2_R2.fastq.gz
@wt_1_1/1
ACGTTGCAAGGT
+
IIIIIIIIIIII
@wt_1_2/1
ACGTTGCAAGGT
+
IIIIIIIIIIII
```

The script

`run_samples.sh` loops over the R1 files, works out the sample name and the matching R2 file name, skips samples whose R2 is missing or whose output already exists, and then runs a step on each sample. Our "step" just counts the reads (lines / 4) so you can run it anywhere; the comment shows where a real tool such as `bwa mem` would go.

```
#!/usr/bin/env bash
# run_samples.sh - run a step on each pair of FASTQ files (NAME_R1 + NAME_R2)
# usage: ./run_samples.sh READ_FOLDER OUTPUT_FOLDER
```

```bash
set -euo pipefail

if [[ $# -ne 2 ]]; then
    echo "usage: $0 READ_FOLDER OUTPUT_FOLDER" >&2
    exit 1
fi
READDIR=$1
OUTDIR=$2

if [[ ! -d $READDIR ]]; then
    echo "ERROR: $READDIR is not a folder" >&2
    exit 1
fi
mkdir -p "$OUTDIR"

for R1 in "$READDIR"/*_R1.fastq.gz; do
    if [[ ! -e $R1 ]]; then
        echo "ERROR: no *_R1.fastq.gz files in $READDIR" >&2
        exit 1
    fi
    SAMPLE=$(basename "$R1" _R1.fastq.gz)       # reads/wt_1_R1.fastq.gz -> wt_1
    R2=${R1%_R1.fastq.gz}_R2.fastq.gz           # reads/wt_1_R2.fastq.gz
    OUT=$OUTDIR/$SAMPLE.counts.tsv

    if [[ ! -f $R2 ]]; then
        echo "WARNING: $SAMPLE has no R2 file ($R2), skipping" >&2
        continue
    fi
    if [[ -s $OUT ]]; then
        echo "$SAMPLE: $OUT already exists, skipping" >&2
        continue
    fi

    echo "$SAMPLE: counting reads in $R1 and $R2" >&2
    # A real pipeline would run a tool here instead, for example:
    #   bwa mem -t 4 genome.fa "$R1" "$R2" > "$OUTDIR/$SAMPLE.sam"
    N1=$(( $(zcat "$R1" | wc -l) / 4 ))
    N2=$(( $(zcat "$R2" | wc -l) / 4 ))
    # write to a temporary file, then rename it, so a crash can't leave
    # a half-written file that looks finished the next time we run
    printf '%s\t%d\t%d\n' "$SAMPLE" "$N1" "$N2" > "$OUT.tmp"
    mv "$OUT.tmp" "$OUT"
done
echo "All done" >&2
```

```bash
chmod +x run_samples.sh
./run_samples.sh
./run_samples.sh reads results
cat results/*.counts.tsv
```

```
usage: ./run_samples.sh READ_FOLDER OUTPUT_FOLDER
mut_1: counting reads in reads/mut_1_R1.fastq.gz and reads/mut_1_R2.fastq.gz
WARNING: mut_2 has no R2 file (reads/mut_2_R2.fastq.gz), skipping
wt_1: counting reads in reads/wt_1_R1.fastq.gz and reads/wt_1_R2.fastq.gz
wt_2: counting reads in reads/wt_2_R1.fastq.gz and reads/wt_2_R2.fastq.gz
All done
```

```
mut_1   4   4
wt_1    3   3
wt_2    5   5
```

Run it again and nothing is redone. This matters when a job running 50 samples dies on sample 37: fix the problem and re-run the same command.

```
./run_samples.sh reads results
```

```
mut_1: results/mut_1.counts.tsv already exists, skipping
WARNING: mut_2 has no R2 file (reads/mut_2_R2.fastq.gz), skipping
wt_1: results/wt_1.counts.tsv already exists, skipping
wt_2: results/wt_2.counts.tsv already exists, skipping
All done
```

Do a dry run first. Before starting a loop that runs a slow tool on many files, put `echo` in front of the command. The loop prints each command instead of running it, so you can check the file names are right:

```
for R1 in reads/*_R1.fastq.gz; do
    SAMPLE=$(basename "$R1" _R1.fastq.gz)
    R2=${R1%_R1.fastq.gz}_R2.fastq.gz
    echo bwa mem -t 4 ref.fa "$R1" "$R2" ">" "results/$SAMPLE.sam"
done
```

```
bwa mem -t 4 ref.fa reads/mut_1_R1.fastq.gz reads/mut_1_R2.fastq.gz > results/mut_1.sam
bwa mem -t 4 ref.fa reads/mut_2_R1.fastq.gz reads/mut_2_R2.fastq.gz > results/mut_2.sam
bwa mem -t 4 ref.fa reads/wt_1_R1.fastq.gz reads/wt_1_R2.fastq.gz > results/wt_1.sam
bwa mem -t 4 ref.fa reads/wt_2_R1.fastq.gz reads/wt_2_R2.fastq.gz > results/wt_2.sam
```

Using a sample sheet instead

Guessing samples from file names works until the names from the sequencing center get messy. A **sample sheet** - a table with one row per sample - lists exactly what to run and can hold other information, like the condition. Make one (tab-separated, with a header):

```
printf 'sample\tcondition\tR1\tR2\n' > samples.tsv
printf 'wt_1\twildtype\treads/wt_1_R1.fastq.gz\treads/wt_1_R2.fastq.gz\n' >> samples.tsv
printf 'wt_2\twildtype\treads/wt_2_R1.fastq.gz\treads/wt_2_R2.fastq.gz\n' >> samples.tsv
printf 'mut_1\tmutant\treads/mut_1_R1.fastq.gz\treads/mut_1_R2.fastq.gz\n' >> samples.tsv
column -t samples.tsv
```

```
sample  condition  R1                      R2
wt_1    wildtype   reads/wt_1_R1.fastq.gz   reads/wt_1_R2.fastq.gz
wt_2    wildtype   reads/wt_2_R1.fastq.gz   reads/wt_2_R2.fastq.gz
mut_1   mutant     reads/mut_1_R1.fastq.gz  reads/mut_1_R2.fastq.gz
```

The script reads the sheet with the `tail -n +2 | while IFS=$'\t' read -r` pattern from above. Everything the loop prints goes into one output file, because the redirection is after `done`:

```
#!/usr/bin/env bash
# run_sheet.sh - process the samples listed in a sample sheet
# usage: ./run_sheet.sh SAMPLE_SHEET OUTPUT_FOLDER
# The sample sheet is tab-separated with a header line:
#   sample  condition  R1  R2
set -euo pipefail

if [[ $# -ne 2 ]]; then
    echo "usage: $0 SAMPLE_SHEET OUTPUT_FOLDER" >&2
    exit 1
fi
```

```
SHEET=$1
OUTDIR=$2
if [[ ! -s $SHEET ]]; then
    echo "ERROR: sample sheet $SHEET is missing or empty" >&2
    exit 1
fi
mkdir -p "$OUTDIR"

tail -n +2 "$SHEET" | while IFS=$'\t' read -r SAMPLE CONDITION R1 R2; do
    for FQ in "$R1" "$R2"; do
        if [[ ! -f $FQ ]]; then
            echo "ERROR: $SAMPLE: can't find $FQ" >&2
            exit 1
        fi
    done
    echo "$SAMPLE ($CONDITION): counting reads" >&2
    N1=$(( $(zcat "$R1" | wc -l) / 4 ))
    printf '%s\t%s\t%d\n' "$SAMPLE" "$CONDITION" "$N1"
done > "$OUTDIR/read_counts.tsv"
echo "wrote $OUTDIR/read_counts.tsv" >&2
```

```
chmod +x run_sheet.sh
./run_sheet.sh samples.tsv results
cat results/read_counts.tsv
```

```
wt_1 (wildtype): counting reads
wt_2 (wildtype): counting reads
mut_1 (mutant): counting reads
wrote results/read_counts.tsv
wt_1    wildtype    3
wt_2    wildtype    5
mut_1   mutant  4
```

A warning about `while read` loops: a program inside the loop that reads from STDIN (for example `ssh`, or a tool reading input from a pipe) will swallow the rest of the sample sheet, and the loop stops early. Give such commands `< /dev/null`.

Next step: one job per sample

These loops run the samples one after another. On the cluster you usually want each sample to run as its own job, at the same time. A SLURM **job array** runs the same script many times, and each copy gets a different number in `$SLURM_ARRAY_TASK_ID`, which you use to pick one line of the sample sheet (for example with `sed -n "${N}p"`, see the data processing lab). We'll write these in UNIX IV.

Exercises

Write your answers as scripts (with `set -euo pipefail` and comments), and check them with ShellCheck.

1. **File names.** Set `f=/bigdata/gen220/shared/run7/Sample12_S3_L001_R1_001.fastq.gz`. Using only `${...}` and/or `basename`, print (a) the file name without the folder, (b) the name of the matching R2 file, and (c) the sample name `Sample12`.
2. **Checking input.** Write `bed_summary.sh` that takes a BED file as its argument, prints a usage message if no argument is given, an error if the file doesn't exist, and otherwise prints the number of features and their total length in bp. Test it on `rice_random_exons.bed`.
3. **Loops.** Write a loop over the files in `data/` that prints, for each one, its name and number of lines, e.g. `codon_table.txt 64`. For `.gz` files, count the lines of the uncompressed contents (hint: `case` or `[[ $f == *.gz ]]`).

4. **Pipeline.** Change `run_samples.sh` so it also writes a single summary file `results/all_counts.tsv` with a header line (`sample`, `R1_reads`, `R2_reads`) followed by one line per sample. Then change it to read the sample names from `samples.tsv` instead of from the file names. Run it twice and make sure the second run doesn't redo any work.

Further reading

- Software Carpentry, The Unix Shell: Loops and Shell Scripts
- explainshell.com - paste in a command and it explains each part
- ShellCheck - finds bugs in shell scripts
- Bash Guide and Bash Pitfalls on the Wooledge wiki

Next

Continue with the UNIX III lab: Data processing with pipes, then Python I. Running these scripts as cluster jobs (and SLURM job arrays) is in UNIX IV.