Shell scripting

Childhood Cancer Data Lab

Shell scripts keep a *living* record of your UNIX-ing

- Using the command line interactively is great, but your commands aren't actually saved
  - (ok, they are saved in your command history, but command histories are NOT a reliable place to implicitly record your work! For one, they are not infinite…)

- Shell scripts are files that run shell and UNIX commands
  - Both preserves the history of the commands you ran, *and runs them!*

The anatomy of a shell script

```bash
$ example.sh  ×

Users > spielman > Projects > example_project > scripts > $ example.sh
 1   #!/bin/bash
 2
 3   # Setup overall settings for how the code should run
 4   set -euo pipefail
 5
 6   # Define variables you'll work with
 7   # This commonly includes paths and filenames
 8   DATA_DIR="../data"
 9   INPUT_FILE="processed_data.csv"
10   OUTPUT_FILE="renamed_processed_data.csv"
11
12
13   # Write your code!
14
15   # For example, maybe we are renaming the data file:
16   mv $DATA_DIR/$INPUT_FILE $DATA_DIR/$OUTPUT_FILE
```

We generally like to run scripts from the directory where they are saved to avoid path confusion

`cd ~/Projects/example_project/scripts`

Once in the right directory, we run this script from the terminal as:

`bash example.sh`

Best practices in scripting


```
$ example.sh ×
Users > spielman > Projects > example_project > scripts > $ example.sh
1   #!/bin/bash
2
3   # Setup overall settings for how the code should run
4   set -euo pipefail
5
6   # Define variables you'll work with
7   # This commonly includes paths and filenames
8   DATA_DIR="../data"
9   INPUT_FILE="processed_data.csv"
10  OUTPUT_FILE="renamed_processed_data.csv"
11
12
13  # Write your code!
14
15  # For example, maybe we are renaming the data file:
16  mv $DATA_DIR/$INPUT_FILE $DATA_DIR/$OUTPUT_FILE
```

```
1    #!/bin/bash
```

The very first line of shell scripts often contain a **shebang (#!)** indicating which shell interpreter be used when running this shell script.

This script will use the **BASH** shell

The **path to the given shell interpreter** immediately follows (no spaces!) the **#!**

The shebang can be generally used to make any script *executable*, for interpretable languages. It also cues you in to what language the code is in.

```
#!/usr/local/bin/python3          #!/usr/bin/env python
                                  #!/usr/bin/env bash
```

Best practices in scripting


```
 3    # Setup overall settings for how the code should run
 4    set -euo pipefail
```

We like to use **set** to define preferences for how errors should be handled while running this script

- The **-e** flag causes the script to exit if any step has an error
- The **-u** flag causes the script to exit if a variable isn't defined
- The **-o pipefail** option causes the entire script to fail if any step in a pipeline fails

Note these can be used one-at-a-time, or pick only some options! All of these are legit
(but **o pipefail** has to be kept together, and usually last):

```
set -e
set -u
set -o pipefail
set -eo pipefail
```

Defining variables

```
 6    # Define variables you'll work with
 7    # This commonly includes paths and filenames
 8    DATA_DIR="../data"
 9    INPUT_FILE="processed_data.csv"
10    OUTPUT_FILE="renamed_processed_data.csv"
```
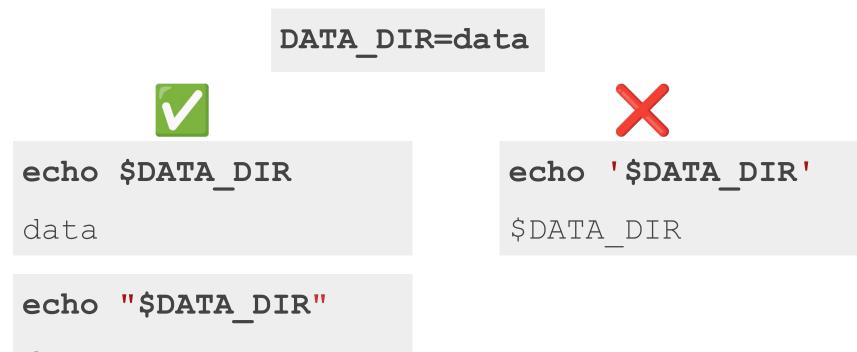
Variables are *defined* as **VARIABLE_NAME="CONTENT"** *without any spaces!!*

(Using quotes when defining variables is not strictly required, but it is best practice.)

Variables are *used* with **dollar signs: $CONTENT**, and sometimes braces: **${CONTENT}**

(Note there are more types of variables, like arrays, which are defined and used slightly differently! We'll just focus on single-value variables here).

Use double quotes when referring to variables

```
DATA_DIR=data
```

✅

```
echo $DATA_DIR
```

```
data
```

```
echo "$DATA_DIR"
```

```
data
```

❌

```
echo '$DATA_DIR'
```

```
$DATA_DIR
```

Variations on a variable

```
 6    # Define variables you'll work with
 7    # This commonly includes paths and filenames
 8    DATA_DIR="../data"
 9    INPUT_FILE="processed_data.csv"
10    OUTPUT_FILE="renamed_processed_data.csv"
```

You can combine variables with strings directly:

```
DATA_DIR="../data"
INPUT_FILE="processed_data.csv"
OUTPUT_FILE="renamed_$INPUT_FILE"
```

You can set up your variables to contain the path, if you want *(and it makes sense for the code!)*

```
DATA_DIR="../data"
INPUT_PATH="$DATA_DIR/processed_data.csv"
OUTPUT_PATH="$DATA_DIR/renamed_processed_data.csv"
```

Use curly braces **{ }** to combine variables safely

```
DATA_DIR="../data"
INPUT_FILE="processed_data.csv"
PREFIX="renamed"
OUTPUT_FILE="${PREFIX}_${INPUT_FILE}"
```

Curly braces protect the variable

```
INPUT_FILE="processed_data.csv"

PREFIX="renamed"
```

✅

```
echo "${PREFIX}_${INPUT_FILE}"
```
renamed_processed_data.csv

❌

```
echo "$PREFIX_$INPUT_FILE"
```
processed_data.csv

Putting it all together

Users > spielman > Projects > example_project > scripts > $ example.sh

```bash
1   #!/bin/bash
2
3   # Setup overall settings for how the code should run
4   set -euo pipefail
5
6   # Define variables you'll work with
7   # This commonly includes paths and filenames
8   DATA_DIR="../data"
9   INPUT_FILE="processed_data.csv"
10  OUTPUT_FILE="renamed_processed_data.csv"
11
12
13  # Write your code!
14
15  # For example, maybe we are renaming the data file:
16  mv $DATA_DIR/$INPUT_FILE $DATA_DIR/$OUTPUT_FILE
```

Now let's write a script to...

- Download paired FASTQ reads (R1 and R2 files) programmatically - no "point-and-click" in browser!!
    - https://trace.ncbi.nlm.nih.gov/Traces/sra/?study=SRP255885

- Save these files to the appropriate directory in your forked repository

- Ask how many lines are in each FASTQ file