Linux (Bash) Shell Scripts

Background: Why learn shell scripting?[1]

- It gives access to large-scale computing on many platforms, including 100% of the top-500 supercomputers and 90% of cloud infrastructure.
- It makes automating repetetive tasks easy.
- 80% of a data analyst's time is spent cleaning up data. Shell scripting for I/O and extracting data from text can be much easier than doing it in R.
- There are many data science problems with so much data that we can't use a sophisticated model, but a simple statistic (mean, median) or graph can answer the question. The issue becomes, "Can I even read the data?" For a person who can write a shell script to extract a little information from each of many files, the answer is often "Yes."
- A few years ago, R's `tidyr` and other packages introduced the pipeline to R programmers, imitating what the shell has been doing since the 1970s! Shell scripting ideas can improve your use of R: write small tools that do simple things well, using a clean text I/O interface.

"This is the Unix philosophy:

- Write programs that do one thing and do it well.
- Write programs to work together.
- Write programs to handle text streams, because that is a universal interface."

–Doug McIlroy, manager of the Bell Labs UNIX team

Linux (Bash) Shell Scripts

A *shell script* is a text file of commands run by Bash, the Linux command-line interpreter.

- To run a first script,
  - open a new file `hello.sh`, paste the text,

```
#!/bin/bash

echo 'Hello, World.' # echo displays a line of text. "#" starts a comment.
```

    and save the file. The first line tells the program loader to run `/bin/bash`.
  - run `chmod u+x hello.sh` to add "execute" (`x`) to the user's (`u`) permissions (run `ls -l hello.sh` before and after to see the change)
  - run `./hello.sh`

[1] Bash is a defective programming language. Google's Shell Style Guide says not to use it for programs of more than 100 lines. It is suited to solving problems that evolve from typing at the command prompt.

- Assign a variable via `NAME=VALUE`, where there is no space around `=`, and
  - `NAME` has letters (`a-z,A-z`), underscores (`_`), and digits (and does not start with a digit)
  - `VALUE` consists of (combinations of)
    * a string, e.g. `a=apple` or `b="apple and orange"` or `c=3`
    * the value of a variable via `$VARIABLE` (or `${VARIABLE}` to avoid ambiguity), e.g. `d=$c; echo "a=$a, b=$b, c=$c, d(with suffix X)=${d}X"`
    * a *command substitution* `$(COMMAND)` (or `` `COMMAND` ``), e.g. `files=$(ls -1); echo $files`
    * an integer arithmetic expression `$((EXPRESSION))`, using `+`, `-`, `*`, `/`, `**` (exponentiaton), `%` (remainder); e.g. `e=$(($c * $c / 2)); echo $e`
    * a floating-point arithmetic expression from the `bc` calculator (see `man bc`) via `$(echo "scale=DECIMAL_POINTS; EXPRESSION" | bc)`, e.g.
      `f=$(echo "scale=6; $c * $c / 2" | bc); echo $f`
      `g=$(echo "scale=10; 1 / s(3.14159 / 2)" | bc --mathlib); echo $g # s() is sine`
    * an indirect variable reference `${!VARIABLE}`, e.g. `g=a; h=${!g}; echo $h`
- Append to a string via `+=`, e.g. `b+=" and cherry"; echo $b`
- Quotes
  - in double quotes, `"..."`, text loses special meaning, except `$` still allows `$x` (variable expansion), `$(...)` still does command substitution (as does `` `...` ``), and `$((...))` still does arithmetic expansion; e.g. `echo "echo ls $(ls)"`
  - single quotes, `'...'`, suppress all expansion; e.g. `echo 'echo ls $(ls)'`
  - escape a character with `\`, as in R; e.g. `echo cost=\$5.00`
- Create several strings with a *brace expansion*,

  `PREFIX{COMMA-SEPARATED STRINGS, or range of integers or characters}SUFFIX`;

  e.g. `echo {Tu,Th}_Table{1..6}`
- Use *wildcards* to write *glob* patterns (not regular expressions) to specify sets of filenames (e.g. for `ls`, `rm`, `cp`, `mv`, etc.):
  - `*` matches any characters
  - `?` matches any one character
  - square brackets, `[...]`, enclose a *character class* matching any one of its characters, except that `[!...]` matches any one character not in the class; e.g. `[aeiou]` matches a vowel and `[!aeiou]` matches a non-vowel
  - `[[:CLASS:]]` matches any one character in `[:CLASS:]`, which is one of `[:alnum:]`, `[:alpha:]`, `[:digit:]`, `[:lower:]`, `[:upper:]`, etc.

  e.g. `ls *; ls *.cxx; ls [abc]*; ls *[[:digit:]]*`

- Conditional expressions

```
if [[ CONDITION_1 ]]; then
  EXPRESSION_1
elif [[ CONDITION_2 ]]; then # use 0 to several elif blocks
  EXPRESSION_2
else                    # else block is optional
  EXPRESSION_DEFAULT
fi
```

  Regarding `CONDITION`,

  - comparison operators include,
    - for strings, `==` (equal to) and `!=` ($\neq$)
    - for integers, `-eq` (equal), `-ne` ($\neq$), `-lt` ($<$), `-le` ($\leq$), `-gt` ($>$), and `-ge` ($\geq$)
  - logical operators include `!` (not), `&&` (and), and `||` (or); e.g.

```
x=3 # also try 4 for 3 and || for &&
name="Philip"
if [[ ($x -eq 3) && ($name == "Philip") ]]; then
  echo true
fi
```

  - match a regular expression via `STRING =~ PATTERN`, which is true for a match; the array `BASH_REMATCH` then contains, at position `0`, `${BASH_REMATCH[0]}`, the substring matched by `PATTERN`, and, at position `$i`, `${BASH_REMATCH[$i]}`, a *backreference* to the substring matched by the ith parenthesized subexpression, e.g.

```
file="NetID.cxx"
pattern="(.*).cxx" # putting bash regex in variable reduces backslash trouble
if [[ $file =~ $pattern ]]; then
  echo ${BASH_REMATCH[1]}
fi
```

  - the spaces in "`[[ `" and "` ]]`" are required

- Loops

  - traverse a sequence: `for NAME in SEQUENCE; do EXPRESSION; done`, e.g.
    `for file in $(ls); do echo "file=$file"; done`
  - zero or more: `while [[ CONDITION ]]; do EXPRESSION; done`, e.g.
    `x=7; while [[ $x -ge 1 ]]; do echo "x=$x"; x=$((x / 2)); done`
    e.g. There's a `while read` example at the end of this handout.
  - one or more (a hack based on the value of several statements being that of the last one and `:` being a no-effect statement): `while EXPRESSION; CONDITION; do : ; done`, e.g.
    `while echo -n "Enter positive integer: "; read n; [[ $n -le 0 ]]; do : ; done`
  - `break` leaves a loop and `continue` skips the rest of the current iteration

- Write a function via

```
function NAME {
  EXPRESSION
}
```

  Access parameters via `$1`, `$2`, ... The number of parameters is `$#`. Precede a variable initialization by `local` to make a local variable. "Return" a value via `echo` and capture it by command substitution. e.g.

```
function binary_add {
  local a=$1
  local b=$2
  local sum=$(($a + $b))

  # (Explain this code line after discussing I/O on the next page.)
  # Write debugging message to stderr (for human to read) by
  # redirecting ("1>&2", described below) stdout to stderr.
  echo "a=$a, b=$b, sum=$sum" 1>&2

  echo $sum # write "return value" to stdout (for code (or human) to read)
}

binary_add 3 4
x=$(binary_add 3 4); echo x=$x
```

- Command-line arguments are accessible via `$0`, the script name, and `$1`, `$2`, ... The number of parameters is `$#`. e.g. Save this in a script called `repeat.sh`:

```
#!/bin/bash

# Repeat <word> <n> times.

# Most scripts start by checking that the number of command-line
# arguments is correct (sometimes including other checks) and
# printing a usage line if not.
if [[ $# -ne 2 ]]; then        # Recall: "-ne" checks integer inequality.
  echo "usage: $0 <word> <n>" 1>&2 # write error message to stderr (below)
  exit 0
fi

word=$1
n=$2
for i in $(seq $n); do  # try "man seq" to see what it does
  echo $word
done
```

- Input/output (I/O), pipelines, and redirection

  - A script starts with three I/O streams, `stdin`, `stdout`, and `stderr` for standard input, output, and error (and diagnostic) messages, respectively. Each stream has an associated integer *file descriptor*: 0=`stdin`, 1=`stdout`, 2=`stderr`.

  - A *pipeline* connects one command's `stdout` to another's `stdin` via `COMMAND_1 | COMMAND_2`.

  - I/O can be *redirected*:

    * redirect `stdout` to

      · write to `FILE` via `COMMAND > FILE`, overwriting `FILE` if it exists (here "`>`" is shorthand for "`1>`")

      · append to `FILE` via `COMMAND >> FILE`

    * redirect `stderr` to write to `FILE` via `COMMAND 2> FILE`

    * redirect both `stdout` and `stderr` via `COMMAND &> FILE` (shorthand for `COMMAND > FILE 2>&1`, "redirect `COMMAND`'s `stdout` to `FILE` and redirect its `stderr` to where `stdout` goes")

    * redirect `stdout` to go to `stderr` (e.g. to `echo` an error message) via `COMMAND 1>&2` ("redirect 1 (`stdout`) to where 2 (`stderr`) goes")

    * redirect `stdin` to

      · read from `FILE` via `COMMAND < FILE` (here "`<`" is shorthand for "`0<`")

      · read from a *here string* via `COMMAND <<< "CHARACTER STRING"`, e.g. `bc -l <<< "4 * a(1)"`

      · read from a *here document* via

      ```
      COMMAND << END_NAME
      EXPRESSSION
      END_NAME
      ```

    * discard unwanted output by writing to `/dev/null`

- Evaluate a string as bash code via `eval STRING`, e.g.

  ```
  a="ls"; b="| wc"; c="$a $b"; echo "c=$c"; eval $c
  ```

  A script that uses `eval` carelessly may be exploited to run arbitrary code, so `eval` is dangerous.

Here are two more handy loop examples demonstrating `while read` loops:

- A `while read` loop reads text one line at a time into variables `name1` (first word on a line), `name2` (second word), ..., and `nameN` (Nth through last word). (Don't use `....`)

  ```
  while read name1 name2 ... nameN; do
    COMMAND    # some COMMAND involving name1, name2, ..., nameN
  done
  ```

  Follow the loop with " `< FILE`" to read from `FILE` instead of `stdin`.

  e.g. Put these lines in a file called `students.txt`:

  ```
  Name      Height Weight Major
  Michael   70     180    Economics
  Li        65     140    Math
  Elizabeth 68     130    History
  ```

  Then run this command

  ```
  while read name height weight; do
    echo "name=$name, height=$height, weight=$weight, major=$major(oops)"
  done < students.txt
  ```

- Here is a variant showing that we can pipe into a compound expression (or grouped command) in braces:

  ```
  echo "1 2 3" | sed 's/ /\n/g' | { sum=0; while read n; do sum=$(($sum + $n)); done; echo $sum; }
  ```

  The space in "`{ `" is required. The "`;`" before the "`}`" is required.

For more information,

- run `COMMAND --help` to see the usage of `COMMAND`, e.g. `seq --help`
- see the `COMMAND` man page (`M-x man Enter COMMAND Enter`)
- see the `bash` man page
- check "The Linux Command Line" by William E Shotts Jr.:
  - free online at `http://linuxcommand.org/tlcl.php`
  - for sale at `www.amazon.com/Linux-Command-Line-Complete-Introduction/dp/1593273894`
- check google