(Bash) Shell Scripts

- To run a first script,

  – open a new file `hello`, paste the text,

    ```
    #!/bin/bash

    echo 'Hello, World.'
    ```

    and save the file. The first line tells the program loader to run `/bin/bash`.

  – run `chmod u+x hello` to add "execute" (`x`) to the user's (`u`) permissions (also run `ls -l hello` before and after to see the change)

  – run `./hello`

- 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 `$VARIABLE`, e.g. `d=$c; echo "a=$a, b=$b, c=$c, d=$d"`
    * 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 ** 2 / 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; 1/sqrt(2)" | bc); echo $f`
    * an indirect variable reference `${!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 and C++; 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 {Mo,We,Fr}_Table{1..6}`

- Use *wildcards* to specify groups of filenames (which are not regular expressions):
  - `*` 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:]`

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

  ```
  if [[ CONDITION_1 ]]; then
    EXPRESSION_1
  elif [[ CONDITION_2 ]] # 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]}`, the substring matched by the ith parenthesized subexpression, e.g.

    ```
    file="NetID.cxx"
    pattern="(.*).cxx"
    if [[ $file =~ $pattern ]]; then
      echo ${BASH_REMATCH[1]}
    fi
    ```
- Loops

  - `for NAME in SEQUENCE; do EXPRESSION; done`, e.g.
    `for file in $(ls); do echo "file=$file"; done`
  - `while [[ CONDITION ]]; do EXPRESSION; done`, e.g.
    `x=7; while [[ $x -ge 1 ]]; do echo x=$x; x=$((x / 2)); done`
  - `until [[ CONDITION ]]; do EXPRESSION; done`
  - `break` leaves a loop and `continue` skips the rest of the current iteration

- Write a function via

  ```
  function NAME {
    EXPRESSION
    return
  }
  ```

  Access parameters via `$1` through `$#`. 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))
    echo $sum
  }

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

- Command-line arguments are accessible via `$0`, the script name, and `$1` through `$#`. e.g. Save this in a script called `repeat`:

  ```
  #!/bin/bash

  # Repeat <word> <n> times.
  if [ $# != 2 ]; then
    echo "usage: $0 <word> <n>"
    exit 0
  fi

  word=$1
  n=$2
  for i in $(seq $n); do
    echo $word
  done
  ```

- Input/output:
  - redirect stdout to
    - write to `FILE` via `COMMAND > FILE`, overwriting `FILE` if it exists
    - append to `FILE` via `COMMAND >> FILE`
  - redirect stderr to write to `FILE` via `COMMAND 2> FILE` (0=stdin, 1=stdout, 2=stderr)
  - redirect both stdout and stderr via `COMMAND &> FILE`
  - redirect stdin to
    - read from `FILE` via `COMMAND < FILE`
    - 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
      ```
  - *pipe* one command's output to another's input via `COMMAND_1 | COMMAND_2`
  - 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 some bash commands I use: `bc`, `cat`, `echo`, `exit`, `find`, `finger`, `for`, `function`, `grep`, `head`, `hostname`, `kill`, `ps`, `sed`, `sort`, `tail`, `time`, `top`, `wc`, `while`

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.
- check google