Open Source Philosophy

Free to use software ≠ Open Source software【Transparency,Flexibility,As good, perhaps better quality】 Licensing: GNU or Creative Commons versus copyright;UNIX is Designed for efficiency

A Process is program in execution: Programs are compiled into objects (binaries, executables, Static, single copy of each, shared) , Programs (or binaries) in execution are processes – Process image is what is found in memory **Processes show Abstraction Level Separation**: ♥Small kernel a master process, provides services to system and user processes, esp scheduling; ♥Simple data type/format (text) and simplified inter-process communication, allows chaining of processes, versus monolithic programs; ♥Separation between ordinary users, root has special privileges

Files: views Almost everything as a file. Tree-like structured. Every user got a own tree: HOME. **File Naming**: File names can be any non white-space character; no blanks in names.

Command: `<Command name> [<Options>] [<Objects>]` command name as verb; options as adverbs, Generally begin with a -; object as nouns; **Shell**: where to execute command(command interpreter);it locates the command, the object and, all being well, executes the command, e.g. ls– User process -> System process -> Kernel

Docker

Single OS: Physical Machine -> OS Kernel -> OS binaries/libraries -> User Proc Each process has own space, but most things shared Can only run apps for that architecture 【efficient】 【cannot support apps from other OperatingSystems or older version of the same OS】

VM: Physical Machine -> Host OS -> Hypervisor -> (VM1: Guest OS -> bins/libs -> User Proc) 【+ A VM can support multiple users (like a conventional architecture); each VM isolated so security enhanced. Can run any app by mounting compatible system; Centralized control】 【Carries the cost of each OS, binaries, kernel, etc. so large footprint on disk and in memory; performance may suffer; costlier to update centrally】

Docker: Physical Machine -> Host OS -> Container Engine -> (Con.1 bins/libs -> User Proc) when using docker, kernel is shared with host OS, OS base image is just some common libs of that distro. Distributed maintenance User initiates updates 【Only carry around as much of the environment as they require. App larger than for native OS, but portable; Use host kernel, smaller footprint which is distributed to users】 【Weaker security than VM】

File Permissions(u:rwx, g:rwx, o:rwx)

• u (user, i.e. file owner) • g (group) • o (other, i.e. everyone else) **For Folder:** • r can read list of files in the directory • w can write/update a file in the directory • x can pass through the directory (to get to a file or subdirectory if name already known).

Script Calling Param(start from the command itself `$0`, number is `$#`)

Shebang: `#!/usr/bin/env bash` or `#!/bin/bash`. env is used here to find bash in PATH. It allows the shell to interpret the script as being a Bash script (versus Python, Shell, etc

Variables `% cat=sherlock` NOTE: NO spaces around "=". use `$cat` to Evaluate.

`% echo "my cat is $cat"` interpolate variable contents into string, `echo "IOU \$${owed}now"` use round brackets "()" for output from a program, use braces("{}") for contents of a variable

System Variables PATH= /bin:/usr/local/bin:/usr/bin:/bin:. In a particular order, and use full path to avoid injection attack.

Conditionals, 0 => True; >0 => False (most typically 1) last command status as `$?`

```
<condition>
then
<statements>
[elif <condition>
<statements>] .....
[else
<statements>]
fi
```

```
case $DAY in
 Mon | Fri ) echo ${DAY}
day ;;
 Tue | Thu ) echo ${DAY}
sday ;;
 Wed ) echo ${DAY}nesday ;;
 S??) echo WEEKEND! ;;
 *) echo "$DAY is not a day
I understand."
 echo "April Fool's
Day?" ;;
 esac
```

```
for i in *.[ch]  # for name
in list  # * at shell level
is list of files
do
 echo $i
 diff $i ../tempdir/$i
 echo
done

foo=string
for (( i=0; i<${#foo}; i+
+ )); do
  echo "${foo:$i:1}"
done
```

```
 a=0
 while [ $a -lt 10 ]
 do
    a=`expr $a + 1`
 done
```

The is anything (i.e. variable or command) that returns a string • Each can be of the sort used in file-name generation, including wild-cards. • Alternate patterns, but with the same set of actions, are separated by |.

Loop

```
for word in $(< file) | for i in * # loop files | IFS="回车" for line in $(< file)
for i in $(seq 1 100); do sum=$(($sum+$i)) done    | for i in one 2 "2 1/2"
```

shift

shifts the arguments to the current script one place to the left (reducing argument count), that means, you read one parameter by `$1`, then shift, then depend on the parameter you got, you may want the next one to be a value or another parameter, you will get next thing by `$1` anyway.

Shell Arithmetic(Only integer) `$((<expression> ))` evaluates the expression and returns the result on stdout. `a=1 a=$((a + 1))`

Regular Expressions

A number of UNIX utilities, e.g. grep, use a different, expanded format for regular expressions derived from ed (predecessor to vi), and this is different with globbing/wildcards. sed and grep (but not awk) can record the match you have made and then recalling it for later use using `\(  \)` , it records the string that matched that regular expression. record up to 9 regular expression matches in a single operation. `\<N>` is used to refer to one of the recorded matches, that is `\1` refers to the first match recorded, `\2` for the second, and so on.

ed	shell	Description
.	?	Single character
[]	[]	Single character from set or range(s)
[^]	[^]	Single character NOT from this set/range(s)
*		Zero or more occurrences of preceding letter
.*	*	Zero or more occurrences of any letter
^		Start of a matching string
$		End of a matching string
\	\	Take special meaning away from next letter
( )		Capture match for later reuse

```
# frequently used patterns:
.* # Any string (include
empty string)
\. # Dot (as such)
[a-zA-Z][a-zA-Z]*
#Alphabetic string
[0-9][0-9]*\.[0-9][0-9]*
#Floating point number
^$ #empty line
```

Globbing/wildcards(this apply to filenames and case statements)

* matches anything in a filename, e.g. `ls x*` matches: x, x.bak, x.txt, x1, xfer_peter (The last is a directory) ? Matches a single letter. In the list above `x?` only matches `x1` `[<letters> ]` Match any one of these letters (can be a range) `[!<letters> ]` Match any one letter BUT NOT any of these (can be a range)A number of UNIX utilities, e.g. grep, use a different, expanded format for regular expressions derived from ed (predecessor to vi)

SED performs editing functions "on the fly". is intended to work as part of a Unix pipeline, i.e. a filter, reading lines one-by-one from a file (or stdin), performing edit functions on the lines and then sending the lines to stdout

```
sed <options> <file> ... -e sequentially, -i inplace
 •-n  Transformed output not sent to stdout (need pipeline? I think just --quiet)
 sed -e 's/[()]/ & /g'-e 's/ */ /g' infile
```

g means global, rather than just the first (which is the default action)

`[<address> [ , <address> ]] <function> [ <arguments> ]` If no addresses are supplied, the function is applied to all input lines. Address can be line number, .(current line), $ last input line, 0((before the first line, only some commands!)), A context address specified by a regularexpression enclosed between / / or Simple arithmetic on an address.

Take lines beginning with strings of lower-case alphabetics and place parentheses around the strings. & is whatever was matched by the regular expression, i.e. works like `\1` following `\(  \)` `s/^[a-z][a-z]*/{&}/`

d can be delete operation. Note that, after a line has been deleted there is no point applying the remainder of the script to the current line (‼), so a new line is obtained and the script started from the top. **p** means print; **l** This option is similar to p, except that non printable characters are displayed and long lines are folded

AWK math includes: `+, -, *, /, % (modulus, remainder), ^ (exponentiation)`

awk regards input lines as a sequence of fields separated by a field separator character The basic information unit for Sed is the input line; for Awk it is the fields within the input line In Awk (and C), the value FALSE is indicated by the integer value 0. no elif in awk. array elements(or array itself) could be removed by `delete a[10] or delete a`

built-in functions: length(a_string) substr(s, m, n) a string(s), starting at m, with length n(or till end, if n absent) sprintf(fmt, expr, ...) same format like printf, but return as string sub(r, t, s) Substitute(代替) t for first regex r in s gsub(r, t, s) Substitute t for all regex r in s E.g. `gsub("  *", " ", $1) # multiple spaces`

Variables in the parameter list are local to the function (i.e. distinct from other variables found in the awk-script). All other variables in a function definition are global.

Variables other than array are value-copied to function,

Gawk has relatively few command-line options: •-F – set Char to be the field separator •-f - take the rules from the file •-v = create a variable and give it a value at the command line. The variable (with that value) will be available to the Awk script

ARGC, Command-line argument count ARGV, Array of command-line arguments FILENAME, Name of input file (there may be several) FNR, Index of line in current file FS, Input field separator (default blank, tab) NF, Number of fields in the current line NR, Index of current line (from start of first file) OFS, Output field separator (default blank)

```
# it will replace $2 with fred, then read the file, copy everything to file fred
# in copy_file                    # in terminal
gawk "{print \$0 > \"$2\"}" $1   ./copy_file Alice_in_Wonderland.txt fred
```

Alternatively, Awk allows the output to be redirected into a UNIX command via a pipe internal to the awk-script. The command is a string.

```
This script calls Awk to print all the students by decreasing mark
 #!/usr/bin/env bash
 gawk -F"#" '{print $2 "#" $1 "#" $4 | "sort -t#-k 3nr"}'
marks | sed -e 's/#/ /' -e 's/#/: /'
```

formats: c is Single Character; s is string; d is integer; e is Scientific notation(6 decimal); f is floating(6 decimal); g will output e-like or f-like whenever is shorter. `awk '{printf("%d:%s\n", ++i , $0)}' test_file`

Relop condition `gawk '$0 == "" {sum++}END{print sum}' $1` Matchop condition: ~ (match) and !~ (does not match). The regular expression is contained between / /. Below counts the number of blank lines `gawk '$0 ~ /^$/{sum++}END{print sum}' $1`

GIT

`git add <files>` add files to the set of files staged, staged files will be included in next commit `git commit -a` automatically stage modified and deleted files `git status` check if a file is untracked unstaged or staged `git log` showed the commit history

`git diff --cached` Ignore unstaged files (only compare staged files against HEAD) • `git diff <commit>` Compare against a specific commit, not HEAD • `git diff <commit> <commit>` • `git diff <file_or_dir>` Limit comparison to a particular file or directory

`git checkout <commit>` Travel to a commit or branch. This command moves HEAD and updates all files in the working tree to match the new location. Options: `<commit>` The commit or branch to travel to. Defaults to HEAD. `<file>` If specified, do not travel (no change to HEAD). Instead, overwrite the specified file or directory with its contents at the specified commit. Great for restoring an old version of a file if you broke it!

`git branch` List branches • `git branch <name>` Create a new branch at HEAD Options:-m Rename the current branch to the specified name.-d Delete branch with the specified name. Branch must be fully merged • `git checkout -b <name>` Create and check out a new branch. Equivalent of: `git branch <name>` and `git checkout <name>`

`git merge <branch>` Merge the changes made in branch back into the current branch.

gitignore patterns: * matches anything except / ? Matches any single character except / # coment

Make

each program module will be compiled to binary, then linked up as an executable. `make` takes a specification of doing above. useful when intermediate files take time to generate in a workflow. makefile is the only place to define everything. it consists of rules and variables. `make -j tasks` parallel -k keepgoing to next target if an error met

a rule is like this:

```
<a target>:<pre-requisites>
<tab><command>
C14UBT_results.txt : C14UBT_clean.tsv
  analyseUBT.py C14UBT_clean.tsv > C14UBT_results.txt
```

variables defined as `data_root=/usr/home`, called as `$(data_root)`

% matches 0 or more chars, equals to ".*" in regex.

built-in: $@ target; $< the 1st precondition; $^ all the preconditions; $* last match

`.PRECIOUS %.clean_tsv` marks a file as not being auto-remove.

Commands

`ls -l (default order: number(as string), A-Z, a-z) add -r to Reverse current order`
`-rwxrwxrwx 1 ziyuanding ziyuanding 129290 May  5 13:35 'Assignment 2.pdf'`
-C Produce multicolumn output; -t List in order of decreasing time-stamp (latest first). -s show size, -S order by size from big to small

`mv <file1> <file2>` Change the name of file1 to be file2. If file2 already exists, overwrite.

ps prints out information about processes you have running. If no options are specified, only the processes for the currently active window are reported.

```
PID TTY          TIME CMD
10  pts/0    00:00:00 bash
```

- -l Provides a long, listing containing more information.

```
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S  1000    10     9  0  80   0 -  1585 do_wai pts/0    00:00:00 bash
```

- -f provides more information about the command (which options were used)

`cut -d <delim> -f <list> <files> default delim: tab` cut returns the fields indicated. `cut –d ',' -f 1,5-7 datafile.csv`

`paste -d <sep> <files> default delim: tab` concat files line by line

`tr <options> <string1> <string2> < some_file` -s Squeeze multiple contiguous occurrences of string1 characters

```
echo "abc123def" | tr -c '0-9' '#' # replace all non-number to \#
tr '[A-Z]' '[a-z]' < textfile
tr –s ' ' '#' < textfile # s is Squeeze multiple contiguous occurrences
```

`comm [<output options>] <file1> <file2>` The two sorted files are compared, and three column output is produced. Column 1 contains those lines only found in , Column 2 contains lines only in , Column 3 contains lines common to both files. The output options suppress(ignore) the named columns,e.g. –1 Suppress column unique to –23 Suppress column unique to and column of common entries

`uniq <file>` Compresses adjacent lines that are repeated in a file to a single copy (i.e. duplicates removed). The -c option outputs like this `2 pear`, that is, with counter.

`sort [options] <files> default sep is <TAB>` -u Removes duplicated entries following the sort (i.e. as if uniq had been applied) -t field separator; `-k <start>[<type>][,<end>[<type>]` Instead of sorting from the first field, sort from the `<start>` field. If `<end>` is specified, only the fields from `<start>` to `<end>` are sorted. For example: `ps| sort -k 4 # list processes alphabetically`

test is a program (/bin/test) -d Tests if the directory exists; -f Tests if the file exists as ordinary file; -s Test if the file exists and is not 0 length; -n Is the string non-zero length; -z Is the string zero length Bash also implements built tests mimicking test [[-s empty_book.txt]] [[1038 < 999]] 或者 [[1038 –lt 999]]

`find [<options>] <path> [<expression>] default -print` -name The test file matches. -type The type of the file is as specified(d or f or l(link)); -newer The test file has been accessed more recently than was modified. -print Prints the full path-name of the file. -exec Execute `<command>` on each file that survives previous tests. All command-line arguments to find after this are assumed to pertain to the `<command>`, up to a ";" • {}refers to the file to which the command is being applied. -ok Same as -exec, except that user is prompted before `<command>` is executed.

```
find . –print # list all the filesin . And subdirectories
find .-name "[Mm]akefile"-exec make\;-print # find every Makefile andcall make
find . -type f -exec ls -l '{}' \; #find every file (not dir) and list it
find . –perm 600 –exec chmod 644 '{' \;  # find rw------- file, convert to rw-r--r-
```

`grep [<options>] <regular-expression> <file> ...` -i ignore case; -n add line number in front; -v Invert the match so only non-matching lines are reported

```
grep '^\.[VABL][LIE]$' file
find.–name Makefile-exec grep awk '{}' \;-print
```

```
# process2file: std Input, descriptor 0, stdout, descriptor 1, stderr,descriptor 2
# process2process: Piping not apply to stderr, but workaround: prog 2> err | anlyse

echo "rewrite"  > a # redirect stdout        echo "append"  >> a
wc < _a # count number of lines,words,letters by Redirect standard input
qbittorrent -run 2> error.log # Redirect standard error output

ln _a _b # answer two files

gzip large_file.txt > trace 2> errs & # background files cannot read keyboard input

chmod ugo+rwx myapp # rwxrwxrwx      chmod go-rw myapp # rwx--x--x
```

Anti-Bugging is to not simply assume that the input users provide is in fact what the program expects, but to take steps to ensure it is, by adding prior tests.

– Sanity checks– This is important; GIGO is a thing Antibugging is the addition of tests, typically near the start of a program, which ensure that that data coming from the user is consistent with what is expected. It is important because otherwise, nonsense results may be computed from absent, out of range or otherwise problematic data.

Markdown A meta language which has some Portability. Aimed at blogging, doc, readme files; Markdown goes back to the explicit tag idea, but makes the tags more intuitive.– Subset of most useful features, cf LaTeX; used by Reddit, R studio, GitHub, Slack, Trello, Discord, helpOSTS.

heading: ===(level 1), ----(level 2), or use sharp, same effect; paragraph: Extra blank line; Line break: (2 spaces at end of line); Italic font style: use _ to surround your words; bold: use double *; Fixed font: `monospace` bullet list: - or * quote: use > before what you need to quote [Link to UWA](http://uwa.edu.au) ![Image](https://study.com/cimages/multimages/16/discount/2345221_960_720.png,"markdown") Many features absent cfLaTeX, HTML, e.g. tables

Examples

Shell

```
#count line number
cat file | wc -l
wc -l file | awk '{print $1}'
awk 'END{print NR}' file

# 5th line
sed -n 5p file
head -5 file | tail -1

# last five lines
tail -n5 file

# from second line to end(ignore 1st)
tail -n +2

# shell function
get_copula() {
  local year=$1
  if [[ $YEAR -le $year ]]; then
      copula="is predicted to be"
  fi
  echo $copula
}

# get content from x-th column
country=$(echo $result_row | cut -d "," -f 4)

# get all rows in file by year, then grep by
# gender ignore case, sort by 7th col, t
# specify separator, -n means treat content
# as number, -r is reverse
result_row=$(cat $PATH_TO_CSV | grep $param_year
| grep -i $param_gender | sort -nr -t ',' -k 7 |
head -n 1)


# shell counter
counter=0
counter=$((counter + 1))

# file number counter
file_count=$(ls $1| wc -l)

# get all lines between $2 and $3
diff=$(( $2 - $1 + 1))
head -n $2 $3 | tail -n $diff


# traverse folder and its subfolder
find $1 -type f -exec mv '{}' $2 \;

# check param in one line
if [[ $# -eq 0 || ! -d $1  || ! -d $2 ]]

# sed edit names all files in current dir
ls | sed -e 's/^/..\/data\//'

# file size extraction and compare
size=$(ls -s $1 | cut -d' ' -f 1)
if test $size -ge 10000

# concat two files
paste -s -d '\n' gday.txt sum.txt


# check param numbers
if [ $# != 3 ]; then
  echo "3 arguments required, $#
provided." > /dev/stderr
    exit 1
fi

# check file exist and are not empty
for arg in "$@"; do
    if ! [ -s "$arg" ]; then
        echo "file $arg not exist or
has zero length" > /dev/stderr
        exit 1
    fi
done

# if-else
readonly PAT_CTY="^[a-zA-Z]{3}$"
readonly PAT_YEAR="^[0-9]{4}$"
if [[ $1 =~ $PAT_CTY ]] then
  funt1 $1 $PARAM_GENDER
elif [[ $1 =~ $PAT_YEAR ]] then
    funt2 $1 $PARAM_GENDER
else
    err "Unknown error for $1."
fi

# tab_occurence
tab_occurence=$(head -n 1 "$filename" |
grep -P "\t" -o | wc -l)
if [[ header_tab_occurence -eq 0 ]];
then
   echo "not tab-file"
fi

# if sth is a number
expr $1 "+" 10 &> /dev/null
if [ $? -eq 0 ];then
    echo "$1 is number"
else
    echo "$1 not number"
fi

# edit line 23
sed –i '23s/te/ts/g' test.txt

# print line 25, column 3
sed –n '25p' /web.log | cut –d " " –f3

# remove first 3 lines
sed –i '1,3d' /tmp/*.tmp
```

```
# in awk, call outside
printf("%s\t%s\n", id, ids[id]) | "sort -k 1n"

# 7的倍数
awk '{i=0; while(i<=500){print i; i+=7}}'

# AWK Struct
BEGIN {
  FS="\t"; OFS="\t";
}
END {
  for (line_number in lines) {
      code_of_line = country_cod[line_number]
      if (freq[country_code_of_line]>=3 ) {
printf("%s is %.3f\n", p, avg_r[p])
      }
  }
}

# self-define function
function abs(v) {return v < 0 ? -v : v}
function std(sum_x, sum_x2, n) {
  x_bar = sum_x / n
  return sqrt(sum_x2/n-x_bar * x_bar)
}

  valid_lines=$(awk -v
header_tab_occurence="$header_tab_occurence"
-F '\t' \
          'NF ==(header_tab_occurence+1)
&& $2 != "" && $3 != "" && $3 >= 2011 && $3
<=2021 {
              print $0
          }' "$filename")
  echo "$valid_lines" > "$temp_file_name"
```