LINUX - AN INTRODUCTION TO SOME BASIC UTILITIES

THE UNIX PHILOSOPHY

- simple tools
- each doing one job well
- compose them in a pipeline and you have a powerful language.

Topics:

1. executing bash scripts (positional parameters)
2. redirection
3. pipes/pipelines
4. filtering data
5. sed ("stream editing")

We can either

1. type linux commands on the commandline, or
2.
   - type them in files,
   - make those files executable,
   - and run them as shell scripts.

For reusability, we'll focus on scripts.

SHELL SCRIPTS

Today we're using the bash shell.

```
$ echo $SHELL
```

You should see

```
$ /bin/bash
```

or

```
$ /usr/bin/bash
```

A bash script is just a sequence of bash commands with a special first line:

File append_the_date.sh:

```
#!/usr/bin/bash
filename=$1  #read first positional parameter from commandline
DATESTAMP=$(date +%y_%m_%d)
mv $filename ${filename}_${DATESTAMP}
echo "$filename moved to ${filename}_${DATESTAMP}"
```

The first line begins with "#!" ("shebang"), followed by the path to the program which is to execute the remaining code, namely `bash`.

```
$ which bash # to find the path to bash
/usr/bin/bash
```

To run (execute) the commands in the file , make the file executable:

```
$ chmod u+x append_the_date.sh
$ ls -lt append_the_date.sh
-rwxr--r-- 1 akea013 akea013 187 Sep  3 10:28 append_the_date.sh
```

and invoke it by stating its name (or path to its name), followed by any expected parameters

```
$ ./append_the_date.sh your_filename_here
```

- The parameters you provide after the script name are called "positional parameters"
- they are available in the script in the order provided as $1, $2, etc. up to $9

SHELL FUNCTIONS

To reuse this code from other code, make it a function:

contents of file append_the_date_function.sh

```
# call as append_datestamp file_name
function append_datestamp(){
  filename=$1  #read file_name from commandline
  DATESTAMP=$(date +%y_%m_%d)
  mv $filename ${filename}_$DATESTAMP
  echo "$filename moved to ${filename}_${DATESTAMP}"
}
```

Source it so that the function is in the shell's namespace, and use it:

```
$ source append_the_date_function.sh  # or replace source by .
$ append_datestamp your_filename_here
```

TIP

put your bash functions in a file, e.g. ~/bin/some_bash_functions_file, source that file from your ~/.bashrc file:

```
#line in .bashrc file
source $HOME/bin/some_bash_functions_file
```

and you can use them anywhere.

Bash programming constructs:

```
$ help
$ help for
$ help while
$ help if
```

```
$ help
```

```
job_spec [&]                            history [-c] [-d offset] [n] or >
(( expression ))                        if COMMANDS; then COMMANDS; [ el>
. filename [arguments]                  jobs [-lnprs] [jobspec ...] or j>
:                                       kill [-s sigspec | -n signum | ->
[ arg... ]                              let arg [arg ...]
[[ expression ]]                        local [option] name[=value] ...
alias [-p] [name[=value] ... ]          logout [n]
bg [job_spec ...]                       mapfile [-d delim] [-n count] [->
bind [-lpsvPSVX] [-m keymap] [-f >      popd [-n] [+N | -N]
break [n]                               printf [-v var] format [argument>
builtin [shell-builtin [arg ...]>       pushd [-n] [+N | -N | dir]
caller [expr]                           pwd [-LP]
case WORD in [PATTERN [| PATTERN]>      read [-ers] [-a array] [-d delim>
cd [-L|[-P [-e]] [-@]] [dir]            readarray [-d delim] [-n count] >
command [-pVv] command [arg ...]        readonly [-aAf] [name[=value] ..>
compgen [-abcdefgjksuv] [-o optio>      return [n]
complete [-abcdefgjksuv] [-pr] [->      select NAME [in WORDS ... ;] do >
compopt [-o|+o option] [-DEI] [na>      set [-abefhkmnptuvxBCHP] [-o opt>
continue [n]                            shift [n]
```

```
coproc [NAME] command [redirectio>  shopt [-pqsu] [-o] [optname ...]
declare [-aAfFgilnrtux] [-p] [nam>  source filename [arguments]
dirs [-clpv] [+N] [-N]              suspend [-f]
disown [-h] [-ar] [jobspec ... | >  test [expr]
echo [-neE] [arg ...]               time [-p] pipeline
enable [-a] [-dnps] [-f filename]>  times
eval [arg ...]                      trap [-lp] [[arg] signal_spec ..>
exec [-cl] [-a name] [command [ar>  true
exit [n]                            type [-afptP] name [name ...]
export [-fn] [name[=value] ...] o>  typeset [-aAfFgilnrtux] [-p] nam>
false                               ulimit [-SHabcdefiklmnpqrstuvxPT>
fc [-e ename] [-lnr] [first] [las>  umask [-p] [-S] [mode]
fg [job_spec]                       unalias [-a] name [name ...]
for NAME [in WORDS ... ] ; do COM>  unset [-f] [-v] [-n] [name ...]
for (( exp1; exp2; exp3 )); do CO>  until COMMANDS; do COMMANDS; do>
function name { COMMANDS ; } or n>  variables - Names and meanings o>
getopts optstring name [arg]        wait [-fn] [id ...]
hash [-lr] [-p pathname] [-dt] [n>  while COMMANDS; do COMMANDS; do>
help [-dms] [pattern ...]           { COMMANDS ; }
```

```
$ help for
```

```
for: for NAME [in WORDS ... ] ; do COMMANDS; done
    Execute commands for each member in a list.

    The `for' loop executes a sequence of commands for each member in a
    list of items.  If `in WORDS ...;' is not present, then `in "$@"' is
    assumed.  For each element in WORDS, NAME is set to that element, and
    the COMMANDS are executed.

    Exit Status:
    Returns the status of the last command executed.
```

```
$ help while

while: while COMMANDS; do COMMANDS; done
    Execute commands as long as a test succeeds.

    Expand and execute COMMANDS as long as the final command in the
    `while' COMMANDS has an exit status of zero.

    Exit Status:
    Returns the status of the last command executed.
```

```
$ help if
```

```
if: if COMMANDS; then COMMANDS; [ elif COMMANDS; then COMMANDS; ]... [ els
    Execute commands based on conditional.

    The `if COMMANDS' list is executed.  If its exit status is zero, then
    `then COMMANDS' list is executed.  Otherwise, each `elif COMMANDS' lis
    executed in turn, and if its exit status is zero, the corresponding
    `then COMMANDS' list is executed and the if command completes.  Otherw
    the `else COMMANDS' list is executed, if present.  The exit status of
    entire construct is the exit status of the last command executed, or z
    if no condition tested true.

    Exit Status:
    Returns the status of the last command executed.
```

REDIRECTION

By default, the shell

1. reads input from the keyboard
2. writes output to the terminal
3. writes error messages to the terminal

Redirection allows us to designate other sources and targets.

REDIRECTION EXAMPLES

```
$ command > filename # capture output to file
$ command < filename # take input from file
$ command >> filename # append output to file
```

The following code generates some data, and increments it:

```
 #!/bin/bash
for number in $(seq 1 20);
  do
    let increment=$number+1;
    echo "$number + 1  =  $increment"
  done
```

Notice that the value of the variable "number" is $number

Say our input data was in a file :

```
$ seq 1 20 > count_to_twenty
```

We can read it from the file into the script with

```
#!/bin/bash
 while read number
 do
   let increment=$number+1;
   echo "$number + 1  =  $increment"
 done < count_to_twenty
```

1. writes the numbers 1..20 to file count_to_twenty
2. reads them sequentially into the variable number
3. increments them and writes to screen

PIPELINES

The pipe symbol is "|".
The sequence of operations

```
$ command1 > filename
$ command2 < filename
$ rm filename
```

can be replaced by

```
$ command1 | command2
```

The output of the first command is used as input to the second.

The previous incrementing code is equivalent to the pipeline

```
 #!/bin/bash
seq 1 20 |
while read number
do
  echo "$number + 1 = $(($number+1))"
done
```

PIPELINE EXAMPLES WITH SOME FILTERING

```
$ cat data_file | sort | uniq  # uniq lines of an unsorted text file
$ cat data_file | sort | uniq -d    #  duplicate lines of unsorted text f
$ history | tail -n 100   # last 100 lines of history file
$ history | grep git | tail -n 100   # last 100 lines of history file con
$ ls -lt | awk '{print $5,"\t", $9}' | sort -rn|head # show directory con
#apply to 5th and 9th fields of ls -lt output, size and name
$ ls -lt | awk '{printf("%d\t%s\n",$5, $9)} | sort -rn # alternative form
```

SED THE STREAM EDITOR

What's a stream editor? -

1. put the editing commands in one file,
2. apply them to any set of files

Note: sed uses regular expressions to match strings.

Contents of sed file, replace_string.sh:

```
#!/usr/bin/bash
# replace all occurrences of $1 with $2 in files named *.$3
STRING=$1
REPLACEMENT_STRING=$2
FILETYPE=$3
find . -name '*.${FILETYPE}' -exec grep -l $STRING {} \; #show files
find . -name '*.${FILETYPE}' -exec grep  $STRING {} \; #show occurrence
sed -e "s/$STRING/${REPLACEMENT_STRING}/g" *.${FILETYPE} {} \; #show re
sed -i.bak -e "s/$STRING/${REPLACEMENT_STRING}/g" *.${FILETYPE} {} \; #
#find . -name '*.${FILETYPE}' -exec sed -i.bak -e "s/$STRING/${REPLACEM
```

- The -i does a live change. Leave it off and the changes are sent to screen, not to file.
- Using -i.bak overwrites the file and copies the original to a backup, to filename.bak
- The last line, uncommented, descends this directory and finds all files of this filetype in all subdirectories and makes this given substitution.

To replace the string "first" with the string "second" in all csv files found in this directory:

```
$ chmod u+x replace_string.sh
$ ./replace_string.sh first second csv
```

TIPS

1. the echo command is your print statement
2. the "#" symbol is a comment
    - document your code,
    - comment out failing parts while you build it up

CAVEATS

1. the bash shell treats single and double quotes differently
2. some bash variables need to be either quoted or enclosed in braces to be captured correctly

LINKS

- Software carpentry shell scripting
- system documentation
  - ```
    $ man bash
    ```
  - ```
    $ help
    ```
  - ```
    $ apropos your_term_here
    ```
  - /usr/share/doc/ packages
  - supplementary documentation packages: e.g. gawk-doc
    ```

LINKS CONTINUED

- Advanced bash scripting guide,
- download from abs on sourceforge
- in the beginning was the command line by Neal Stephenson - an early history of windows, Mac and linux

Thank you.