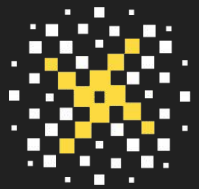
from zero to ${0##*/}

an introduction to bash scripting and HPC

alberto sartori



SISSA
DATASCIENCE
Machine Learning for the Natural Sciences

scripta manent

very good references

- bash manual https://www.gnu.org/software/bash/manual/
- bash guide https://mywiki.wooledge.org/BashGuide
- pro bash programming https://www.apress.com/gp/book/9781484201220

tricky reference

advanced bash scripting guide https://tldp.org/LDP/abs/

quick reference

bash scripting cheat sheet https://devhints.io/bash

hello.sh

```
#!/bin/bash

printf "%s\n" "hello world" # quotes are important


$ chmod u+x hello.sh

$ ./hello.sh
```

how to pass arguments?

```
$ ./my_script arg1 arg2 arg3
```

how to use the arguments?

first argument is assigned to ${1}, second argument to ${2}, and so on

note that ${10} requires the braces, $10 is interpreted as first argument with a 0 at the end

${0} is the name of the script (kind of)

special parameters

${*} a string with all the arguments

${@} one argument at a time

${#} the number of arguments

${?} exit value of the previous command

${$} ID of current process

${-} options of the current shell

hello-arg.sh

```
#!/bin/bash

printf "%s\n" "hello ${1}" # quotes are important


$ chmod u+x hello-arg.sh

$ ./hello-arg.sh alberto
```

Exercise 5: hello-arg2.sh

implement a hello-arg2.sh program that can greet people whose name contains spaces (e.g., Gian Paolo)

how many args (and which) are passed?

```
$ ./my_script one two three "four five"

$ ./my_script *

$ ./my_script *{a,b}.txt

$ ./my_script $(ls)

$ ./my_script 3+4 5 + 6 $(( 9 + 11 ))

$ ./my_script PATH ${PATH}
```

expansions

1) braces *{a,b,c}* {start..stop..step}

2a) tilde ~
2b) parameters and variables ${var}
2c) arithmetic $(( ))
2d) command substitution $() <()

3) word splitting
4) filename/pathname

check-args (or print-args, ca, pa, ...)

```
#!/bin/bash

# adapted from Pro Bash Programming

d=:

printf "${d}%s${d}\n" "${@}"
```

printf

```
$ printf "format string" "${var1}" "${var2}"

%s : generic string

%b : like %s, but escape sequences are translated (e.g., \t)

%q : quoted

%f : floating point number

%e : exponential notation

%d : integer
```

printf - width specification

```
%N[sfde]  # flush right

%-N[sfde] # flush left

%0N[fd]   # add leading zeros

%N.Df     # D number of decimals

%N.Ds     # D sets max length of a string
```

variables

```
var=value # no spaces. Why?

var=3+4

echo ${var}


var =value

var = value

var= value
```

global environment variables

```
$ var=value
$ export var


$ export var=value


$ var=value ./my_script.sh
$ var= ./my_script.sh
```

how to modify/update env vars?

```
$ source /a/file
```

```
$ . /a/file
```

assign output to a variable

```
1: printf -v var "%04d" 3


2a: filename=$(date +%Y-%m-%d)-backup.tar.gz
2b: filename=`date +%Y-%m-%d`-backup.tar.gz # deprecated


3: var=$((a+b)) # integer operations
```

function

```
func_name()
{
  …
}


func_name
```

function with args

```
func_name()
{
 # use $1, $2 etc.
}


func_name arg1 arg2
```

functions and variables

```
my_func(){
  var="new value"
}

my_func
printf "%s\n" "${var}"
```

local variables

```
my_func() {
  local var="inside value" # works nice with IFS
}
var="value"
my_func
printf "%s\n" "${var}"
```

aliases

```
alias lrt='ls -lrt'

alias ll='ls -l'
```

caveat:

they don't run inside a script, they cannot be exported, so, maybe, functions are better

parameter substitution

```
${var-default} # if var is unset uses default

${var:-default} # if var is unset or empty uses default

${var:=default} # as above + default is assigned

${var:+alternate} # uses alternate if var is non-empty

${var:?message} # print message to stderr and exit if var
                                                    empty
```

parameter substitution

```
${#var} # length of variable's content

${var%pattern} # remove the shortest match from the end

${var%%pattern} # remove the longest match from the end

${var#pattern} # remove the shortest match from the beginning

${var##pattern} # remove the longest match from the beginning
```

parameter substitution

```
${#var} # length of variable's content

${var%pattern} # remove the shortest match from the end

${var%%pattern} # remove the longest match from the end

${var#pattern} # remove the shortest match from the beginning

${var##pattern} # remove the longest match from the beginning


${0##*/}     vs     $(basename $0)
```

parameter substitution

```
${var//old/new} # replace all occurrences of old with new


${var:offset:length} # return a substring
```

read values from stdin

```
$ read # answer is stored in ${REPLY}
pippo
$ read a b # can do multiple assignment
pippo pluto
$ read opt -n1 -s
$ read -p "prompt " var
```

control flow and branching (aka if)

```
if expr1
then
    ...
elif expr2
then
    ...
else
    ...
fi
```

how to test

```
test condition

[ condition ]    # mandatory spaces

[[ condition ]]  # mandatory spaces

((c-like test)) # <, <=, >, >=, ==, !=, ternary operator ?:
```

file tests operators

[-e /path/to/file] : file exists

-f : file is regular file

-x : user has execute permissions

-d : is a directory

… more

integer comparison

`[ $a -eq $b ]` equality

`-ne` : inequality

`-lt` : less than

`-le` : less or equal

`-gt` : greater than

`-ge` : greater or equal

integer comparison

`[[ $a -eq $b ]]` equality

`-ne` : inequality

`-lt` : less than

`-le` : less or equal

`-gt` : greater than

`-ge` : greater or equal

integer comparison

((a == b)) # equality. spaces and $ are not mandatory

!= : inequality

< : less than

<= : less or equal

> : greater than

>= : greater or equal

string comparison

`[ "$a" = "$b" ]` equality

`[ "$a" != "$b" ]` inequality

`[ "$a" \< "$b" ]` # or \> strict ordering (what about "<="?)

`[ -z "$a" ]` true if $a is empty

`[ -n "$a" ]` true if $a is non-empty

quotes to avoid word splitting

string comparison

```
a="two words"

b="two words"

[ $a = $b ] # error: too many args

[ "$a" = "$b" ] # fine
```

string comparison

```
[[ "$a" = "$b" ]]  equality

[[ "$a" != "$b" ]] inequality

[[ "$a" < "$b" ]] # or >, strict ordering

[[ -z "$a" ]] true if $a is empty

[[ -n "$a" ]] true if $a is non-empty

[[ "$file" = *.png ]] # pattern matching: no quotes on rhs

quotes on lhs are not mandatory
```

string comparison

```
a="two words"

b="two words"

[[ $a = $b ]] # fine (performs a pattern matching)

[[ "$a" = "$b" ]] # fine (string comparison)

[[ two words = two words ]] # error: too many args
```

strings: less than or equal

```
[ ! "$a" \> "$b" ] # what happens if I use >?

[[ ! "$a" > "$b ]]
```

AND

```
[ condition1 -a condition2 ] # please don't

[ condition1 ] && [ condition2 ]

[[ condition1 && condition2 ]]
```

OR

```
[ condition1 -o condition2 ] # please don't

[ condition1 ] || [ condition2 ]

[[ condition1 || condition2 ]]
```

tests and conditionals reference

BashGuide/TestsAndConditionals

hello-arg3.sh

```
#!/bin/bash

if [[ $# -lt 1 ]]; then

  printf "error: I need at least one arg\n" 1>&2

  exit 1

fi

printf "hello %s\n" "$*"
```

hello-interactive.sh

```
#!/bin/bash

read -p "Insert your name and press Enter: " name

printf "hello %s\n" "${name}"
```

case statement

```
case ${var} in
  val1) … ;;
  val2|val3) … ;;
  *) … ;;
esac
```

select: interactive menus

```
select var_name in opt1 opt2 opt3
do
  # use either ${var_name} or ${REPLY}
  # break when you want to exit
done


PS3="prompt for select"
```

select: interactive menus

```
select var_name in opt1 opt2 opt3
 do
  case ${var_name} in
    opt1) …;;
    opt2) …;;
    opt3) …;;
  esac
done
```

select: interactive menus

```
select var_name in opt1 opt2 opt3
 do
  case ${REPLY} in
    1) …;;
    2) …;;
    3) …;;
  esac
done
```

Exercise 6

Write a program that guesses the number you are thinking and you answer if your number is smaller, bigger or equal.

looping

```
for var in <list>
do
  # use ${var}
done
```

looping

```
for var in a 3 hello

do

  # use ${var}

done
```

looping

```
for var in $(seq 10)
do
  # use ${var}
done
```

looping

```
for var in {0..9}
do
  # use ${var}
done
```

looping

```
for ((var=0; var<10; ++var))
do
  # use ${var}
done
```

looping

```
for var in <array elements>
do
  # use ${var}
done
```

looping

```
n=1

while [ $n -lt 10 ]

do

  …

  ((++n))

done
```

manually parse the options 1/3

```
while [[ $# > 0 ]]
do
    case $1 in
     -o | --option-no_arg)
       ovar=$1
        ;;
```

manually parse the options 2/3

```
    -a | --option_with_arg)
      …
       avar=$1
      avalue=$2
      shift # we skip the argument
       ;;
```

manually parse the options 3/3

```
    *)
        print_usage
        exit 1
        ;;
    esac
    shift  # at each iteration we shift by one
done
```

getopts

```
while getopts :ac:b opt ; do
  case "$opt" in
    a|b) printf "option %s\n" "$opt";;
    c) printf "option %s with arg :%s:\n" "$opt" "$OPTARG";;
    *) break;;
  esac
done
shift $(( OPTIND - 1 )) # then, manually parse the rest
```

Exercise 7

count the number of executable files in your $PATH

Exercise 8

compute min, max, and mean of the values for each NENE* file in the data-shell folder

arrays

```
var=('a' "b" '' 8 ' ' c)

echo "${var[*]}"

echo "${var[@]}"

echo "${#var[@]}"
```

arrays

```
var=('a' "b" '' 8 ' ' c)
echo "${var[0]}" "${var[42]}"
echo "${var[*]}"
echo "${var[@]}"
echo "${#var[@]}"
printf "%s\n" "${var[@]}"
for i in "${var[@]}"
```

how to redirect block of code?

```
for i in {0..9}
do
  printf "**** %d ****\n" $i
done
```

how to redirect block of code?

```
rm -f ofile; touch ofile

for i in {0..9}

do

  printf "**** %d ****\n" >> ofile

done
```

how to redirect block of code?

```
for i in {0..9}
do
  printf "**** %d ****\n"
done > ofile
```

how to redirect block of code?

```
{
for i in {0..9}
do
  printf "**** %d ****\n"
done
…
} > ofile
```

how to redirect block of code?

```
exec 1> ofile

for i in {0..9}

do

  printf "**** %d ****\n"

done
```

how to redirect block of code?

```
exec 3>&1 # 3 points to original stdout
exec 1> ofile
for i in {0..9}
do
  printf "**** %d ****\n"
done
exec 1>&3 # now stdout is as before
```

exec

```
exec program  # run in the same process


exec 1> ofile

exec 0< ifile

exec 2> efile
```

here document

```
command <<LimitString

…

LimitString
```

feed a block of code or text to an interactive program or command

here document

```
$ wc -l <<EOF
I write
Two lines
EOF
```

here document

```
$ wc -l <<EOF
I write
Two lines
EOF
2
```

Gutenberg project

- dataset: https://drive.google.com/file/d/17WBziFbt9nhAW5iV-yHPHmCfquBPrjJO/view?usp=sharing

- find the author which has the largest dictionary (i.e., he/she uses the maximum number of different words)

- find the name of the author who is ranked 137

go parallel

```
for i in "${files[@]}"
do
  cmd "$i"
done
```

go parallel

```
for i in "${files[@]}"
do
  cmd "$i" &
done
wait
```

go parallel

```
MAX_N_PROC=4
counter=0
for i in "${files[@]}"
do
  cmd "$i" &
  if ((++counter == MAX_N_PROC)); then counter=0; wait; fi
done
wait
```

go parallel

```
parallel cmd {} ::: "${files[@]}"
```

go parallel

```
parallel -j 4 cmd {} ::: "${files[@]}"
```

go parallel

```
a_function()

{

…

}

export -f a_function


parallel a_function :: "${files[@]}"
```

TAB-completion

```
complete -W "list of possible tokens" name_of_the_program
```

TAB-completion

```
complete -W "list of possible tokens" name_of_the_program
```

how to update the list on the fly?

TAB-completion

```
a_function()
{
  … # define array COMPREPLY using compgen
}

complete -F a_function name_of_the_program
```

TAB-completion: fundamental variables

- COMPREPLY: each element of the array is a possible completion
- $1: name of the command
- $2: current token to complete
- COMP_WORDS: whole command line splitted into an array
- COMP_CWORD: array index of the current word

```
local current=${COMP_WORDS[$COMP_CWORD]}

current == $2
```

example

```
a_function() # completes with long-name options
{
  COMPREPLY=(
            $(compgen -W "$(for w in $(command $1 --help); do
                              [[ $w =~ ^-- ]] && printf '%s\n' "$w"
                            done | awk '!x[$1]++' )" -- $2)
            )
}
complete -F a_function name_of_the_program
```

```
_pro_qpdfview()
{
    local cur prev words cword
    _init_completion || return

    if [[ $cur == -* ]]; then
         COMPREPLY=($(compgen -W '$(_parse_help "$1")' -- "$cur"))
         return
    fi
    _filedir pdf
}
```

more on TAB-completion

- https://www.gnu.org/software/bash/manual/html_node/Programmable-Completion-Builtins.html

- /usr/share/bash-completion/completions/*

try to implement completion for todo.sh

https://github.com/todotxt/todo.txt-cli

automatically run programs

- at
- cron
- anacron

cron

- put scripts in
  - /etc/cron.{hourly,daily,weekly,monthly}

- crontab
  - crontab -l
  - crontab -e
  - crontab a_file
  - crontab -r

crontab: syntax

```
min hour day_of_month month day_of_week the_command

* * * * * a_command arg1 arg2 # runs every minute

0 0 1 * * command # runs at 0:0 every first of the month

0 8 * * 0,2,4,6 command # runs at 8:00 Sun,Tue,Thu,Sat

*/5 * * * * command # every 5 minutes

* */3 * * * command # every 3 hours

59 23 * * 1-5 command # runs at 23:59 Mon-Fri
```

Exercise 9

add a script to your crontab which prints (to file) the content of ${PATH} and ${SHELL}

crontab: caveat

- runs from your home
  - you may want to cd somewhere

```
* * * * * cd /some/where && ./the_script
```

- PATH is different from the interactive shell
  - set PATH, SHELL in your crontab

- what if your pc is turned off?

anacron

edit /etc/anacrontab