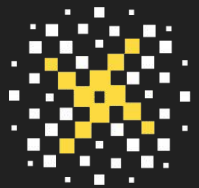
from zero to ${0##*/}

an introduction to bash scripting and HPC

alberto sartori



SISSA
DATASCIENCE
Machine Learning for the Natural Sciences

conquering the command line

what is a terminal (emulator)?

- a program that runs a shell

what is a shell?

- a program that runs others programs
  - synchronously, asynchronously
  - reads a command typed by the user, forwards it to operating system's exec function, starts a new process, and help the user to manage the process itself

- a programming language
  - utilities to combine those programs
  - write files that become programs

how it looks like?

```
alberto@sageX ~$
```

```
alberto@sageX ~%
```

```
sageX ~ #
```

how it looks like?

```
alberto@sageX ~$
```

```
alberto@sageX ~%
```

prompt (aka, PS1)

```
sageX ~ #
```

which shell?

```
$ cat /etc/shells
```

which shell?

```
$ cat /etc/shells
```

```
$ echo $SHELL
```

which shell?

```
$ cat /etc/shells
```

```
$ echo ${SHELL}
```

which shell?

```
$ cat /etc/shells

$ echo ${SHELL}

$ chsh -s /bin/bash
```

command line

```
$ command arg1 arg2 arg3 … argN
```

command line

```
$ command arg1 arg2 arg3 … argN

$ date
```

command line

```
$ command arg1 arg2 arg3 … argN

$ date

$ man date
```

command line

```
$ command arg1 arg2 arg3 … argN

$ date

$ man date

$ man man
```

command line

```
$ command arg1 arg2 arg3 … argN

$ date

$ man date

$ man man

$ man 7 man
```

basic commands

- ls
  - ls -l
  - ls -lh
  - ls -lrt
  - ls -a

basic commands

- `ls`
- `cd`
- `pwd`

basic commands

- `ls`
- `cd`
- `pwd`
- `emacs file.txt`
- `vim file.txt`
- `nano file.txt`

basic commands

- ls
- cd
- pwd
- emacs file.txt
- rm file.txt
  - rm -rf /a/dir
  - rmdir /an/empty/dir

basic commands

- ls
- cd
- pwd
- emacs file.txt
- rm file.txt
- mkdir a_dir
  - mkdir -p a_new_dir/subdir

basic commands

- `ls`
- `cd`
- `pwd`
- `emacs file.txt`
- `rm file.txt`
- `mkdir a_dir`
- `cp /path/to/a/file /other/path`
- `mv /path/to/a/file /other/path`

an alternative to cd

```
/a/long/path$ pushd /etc

/etc$ cd init.d

/etc/init.d$

…

/etc/init.d$ popd

/a/long/path$
```

an alternative to cd

```
/a/long/path$ pushd /etc

/etc$ cd init.d

/etc/init.d$

…

/etc/init.d$ popd

/a/long/path$
```

pushd add new_dir to the top of the stack of dirs

popd remove first entry of dirs and cd to the new top dir

simple text and file processing

```
$ cd somewhere
```

```
$ wget
https://swcarpentry.github.io/shell-novice/data/shell-lesson-dat
a.zip
```

```
$ unzip shell-lesson-data.zip
```

how can I find a file?

where are the files 'NENE*'?

how can I find a file?

where are the files 'NENE*'?

```
$ find . -name 'NENE*'
```

are all 'NENE*' files good?

are all 'NENE*' files good?

```
$ wc -l NENE* > lines.tmp

$ sort -n lines.tmp > sorted.tmp

$ head -n 5 sorted.tmp

$ tail -n 5 sorted.tmp
```

are all 'NENE*' files good?

```
$ wc -l NENE* | sort -n | head -n 5
```

```
$ wc -l NENE* | sort -n | tail -n 5
```

how to exclude *Z.txt files?

```
$ wc -l NENE*[AB].txt

$ wc -l *[AB].txt

$ wc -l *[AB]*
```

how to exclude *Z.txt files?

```
$ wc -l NENE*[AB].txt

$ wc -l *[AB].txt

$ wc -l *[AB]*

$ wc -l *{A,B}*

$ wc -l NENE*!(Z)*   # requires shopt -s extglob
```

how to find the min and max in *[AB].txt?

how to find the min and max in *[AB].txt?

```
$ cat *[AB]* | sort -g | head -1
$ cat *[AB]* | sort -g | tail -1


sort -g | parallel --tee --pipe ::: head tail ::: "-n 1"


$ export LC_ALL="en_US.utf8"
```

find animal-counts dir and cd there

print the animals without repetitions

hint: use uniq

print the animals without repetitions

```
$ cut -d ',' -f '2' animals.txt | uniq # does not work
```

print the animals without repetitions

```
$ cut -d ',' -f '2' animals.txt | uniq # does not work
$ cut -d ',' -f '2' animals.txt | sort | uniq
$ cut -d ',' -f '2' animals.txt | sort -u
```

is there a better way (for big files, and/or long pipelines)?

find folder writing and cd there

find folder writing and cd there

```
$ find / -name writing -type d 2>/dev/null

$ find / -maxdepth 4 -name writing -type d 2>/dev/null
```

text processing

- tr
- grep
- sed
- awk

tr

```
$ tr [option] set1 [set2] <file

$ tr A-Z a-z <file

$ tr -d '\r' <file

$ tr -cs A-Za-z '\n' <file
```

grep

```
$ grep pattern file1 file2

$ grep -c pattern file1 file2 # just count

$ grep -i pattern file # case insensitive

$ grep -v pattern file # invert match pattern

$ grep -E 'pattern1|pattern2' file
```

sed

```
$ echo day > old

$ sed 's/day/night/' old > new

$ cat new

$ echo day | sed 's/day/night/'
```

sed

```
$ sed 's/s/SSS/' haiku.txt

$ sed 's/s/SSS/g' haiku.txt

$ sed 's/s/SSS/2' haiku.txt
```

sed

```
$ sed 's/[Aa-Zz]*day/--- & ---/' haiku.txt
```

```
$ sed '3q'
```

sed

```
$ sed '1 d' haiku.txt

$ sed '$ d' haiku.txt

$ sed '/day/d' haiku.txt

$ sed '/^$/d' haiku.txt
```

sed

add a new line after the line matching a pattern

```
$ sed '/pattern/{s/$/\n/}' file
```

awk

```
$ awk 'condition1 {action1} condition2 {action2}' file

$ awk '/pattern/ {print $0}' file

$ awk '/pattern/ {print $1}' file

$ awk 'rand() < 0.01' file

$ awk 'BEGIN {...;getline} END{...}' file

$ awk '!x[$0]++' file

$ awk '{++x[$0]} END{for (w in x) print x[w], w}' file
```

awk - important built-in variables

FS field separator BEGIN{FS=";"}

OFS output field separator BEGIN{OFS="--"} {$1=$1; print $0}

NF number of fields {print NF} {print $(NF-1)}

NR number of records (lines) NR==1{...} END{print x/NR}

FNR file number of records (reset at each new file)

ENDFILE

FILENAME

```
$ awk 'BEGIN{FS=","} !x[$2]++ {print $2}' animals.txt
```

awk script

```
$ awk -f file.awk
```

```
$ cat file.awk
condition {action}
condition {action}
```

awk - much, much more

https://www.gnu.org/software/gawk/manual/gawk.pdf

count the lines of all .txt in data-shell

count the lines of all .txt in data-shell

```
$ find . -name '*txt' -exec wc -l {} \;
$ wc -l `find . -name '*txt'`
$ wc -l $(find . -name '*txt')

$ shopt -s globstar
$ cat **/*.txt | wc -l
```

Exercise 1

find which files .txt contain "Bennet"

Exercise 2

compute the mean of the entries of file NENE02043A.txt

hint: use awk

#optional: compute min, max, avg, for each "valid" NENE* file

Exercise 3

count how many times the nouns "Amy", "Beth", "Jo", "Meg", are written inside LittleWomen.txt

Exercise 4

remove empty lines from LittleWomen.txt

programming pearls

by Jon Bentley
with Special Guest Oysters
Don Knuth and Doug McIlroy

A LITERATE PROGRAM

Last month's column introduced Don Knuth's style of "Literate Programming" and his WEB *system for building programs that are works of literature. This column presents a literate program by Knuth (its origins are sketched in last month's column) and, as befits literature, a review. So without further ado, here is Knuth's program, retypeset in* Communications *style.* —*Jon Bentley*

Common Words Section

Introduction 1
Strategic considerations 8
Basic input routines 9
Dictionary lookup 17
The frequency counts 32
Sorting a trie 36
The endgame 41
Index 42

1. Introduction. The purpose of this program is to solve the following problem posed by Jon Bentley:

> Given a text file and an integer k, print the k most common words in the file (and the number of their occurrences) in decreasing frequency.

Jon intentionally left the problem somewhat vague, but he stated that "a user should be able to find the 100 most frequent words in a twenty-page technical paper (roughly a 50K byte file) without undue emotional trauma."

Let us agree that a *word* is a sequence of one or more contiguous letters; `"Bentley"` is a word, but `"ain't"` isn't. The sequence of letters should be maximal, in the sense that it cannot be lengthened without including a nonletter. Uppercase letters are considered equivalent to their lowercase counterparts, so that the words `"Bentley"` and `"BENTLEY"` and `"bentley"` are essentially identical.

The given problem still isn't well defined, for the file might contain more than k words, all of the same frequency; or there might not even be as many as k words. Let's be more precise: The most common words are to be printed in order of decreasing frequency, with words of equal frequency listed in alphabetic order. Printing should stop after k words have been output, if more than k words are present.

2. The *input* file is assumed to contain the given text. If it begins with a positive decimal number (preceded by optional blanks), that number will be the value of k; otherwise we shall assume that $k = 100$. Answers will be sent to the *output* file.

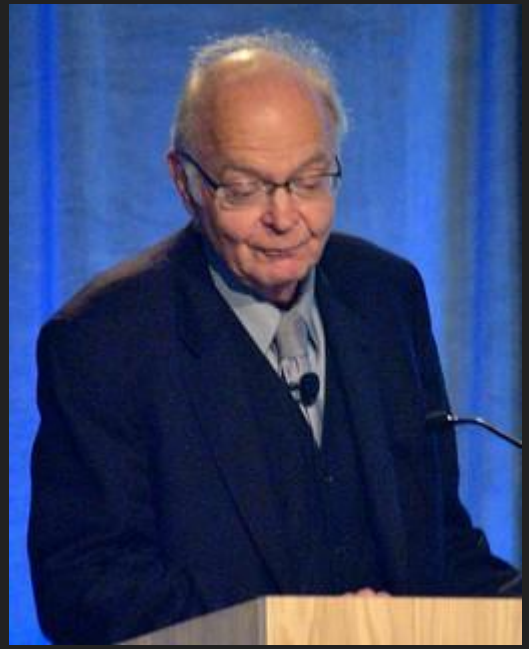
```
define default_k = 100 {use this value if k isn't
        otherwise specified}
```

3. Besides solving the given problem, this program is supposed to be an example of the WEB system, for people who know some Pascal but who have never seen WEB before. Here is an outline of the program to be constructed:

```
program common_words (input, output);
  type ⟨Type declarations 17⟩
  var ⟨Global variables 4⟩
  ⟨Procedures for initialization 5⟩
  ⟨Procedures for input and output 9⟩
  ⟨Procedures for data manipulation 20⟩
  begin ⟨The main program 8⟩;
  end.
```

4. The main idea of the WEB approach is to let the program grow in natural stages, with its parts presented in roughly the order that they might have been written by a programmer who isn't especially clairvoyant.

For example, each global variable will be introduced when we first know that it is necessary or desirable; the WEB system will take care of collecting these declarations into the proper place. We already know about one global variable, namely the number that Bentley called k. Let us give it the more descriptive name *max_words_to_print*.



Douglas' solution

```
cat file |
tr -cs A-Za-z '\n' |
tr A-Z a-z |
sort |
uniq -c |
sort -rn |
sed ${1}q
```

Douglas' solution

```
cat file |
tr -cs A-Za-z '\n' |
tr A-Z a-z |
sort |
uniq -c |
sort -rn |
sed ${1}q
```

Douglas' solution

```
cat file |
tr -cs A-Za-z '\n' |
tr A-Z a-z |
sort |
uniq -c |
sort -rn |
sed ${1}q
```

alternative solution

```
cat file |
tr A-Z a-z |
tr -cs a-z '\n' |
awk '{++x[$1]} END{for (w in x) print x[w], w}' |
sort -rn |
sed ${1}q
```

remove duplicates

```
sort | uniq # blocking, may slow down the rest of the pipeline
```

```
awk '!x[$0]++' # non-blocking
```

combine programs: recap

```
cmd1; cmd2 # cmd1 then cmd2

cmd1 | cmd2 # stdout of cmd1 to stdin of cmd2

cmd1 |& cmd2 # both stdout and stderr to stdin of cmd2

cmd1 || cmd2 # cmd2 is executed if cmd1 FAILED

cmd1 && cmd2 # cmd2 is executed if cmd1 SUCCEEDED
```

output redirection: recap

```
cmd > file # stdout to file

cmd >> file # stdout appended to file

cmd &> file # stdout and stderr to file

cmd &>> file # stdout and stderr appended to file

cmd 2> /dev/null # suppress stderr

cmd > somewhere 2>&1 # stderr to same channel of stdout
```