COMP20200 Unix Programming

Lecture 19

Ravi Reddy Manumachu (ravi.manumachu@ucd.ie)

CS, University College Dublin, Ireland

17/04/2023

Problem

Read a file of text.
Determine the **n** most frequently used words.
And print out a sorted list of those words along with their frequencies.

Solution in Bash Scripting Language

Print 4 most frequently used words.

```
cat file.txt | tr -cs A-Za-z '\n' | tr A-Z a-z | sort | uniq
    -c | sort -rn | head -n 4
```

Print the most frequently used word.

```
(tr -cs A-Za-z '\n' | tr A-Z a-z | sort | uniq -c | sort -rn
    | head -n 1) < file.txt
```

Shell Scripting

- Scripts are typically used to automate workflows involving programs compiled to native code.
- Probably the best way to glue together pre-existing working programs with well-defined functionality.
- A script is a sequence of commands.
  - It can only be run within a certain context.
  - For example: bash script is designed to run within a command shell.
  - Written in a scripting language, like bash, perl, ruby, python.
  - Executed using an interpreter.

Bash: Bourne Again Shell

- Bash is a Unix shell and command language.
- Bash was first released in 1989 as a GNU project.
- It is the default login and interactive shell for most Linux distributions.
- Bash has a massive user base and presence in production systems.

Character Sequence #!

```
#!/bin/bash
# This is the beginning of a simple shell script
```

- The special character sequence **#!** is called *hash-pling*.
- Followed by the full path of the shell (`/bin/bash`).
- **#!** marks the script for interpretation by (`/bin/bash`).
- Any line beginning with # (except *hash-pling*) is a comment.
- Comments are ignored during execution.

Script samples

```
$ var="A new variable!"
```

- Sample shown above is an example of a command executed in an interactive shell.
- The $ represents a command prompt.
- Followed by the bash command.

Shell script variables

Variables

- Creating and setting a variable is this easy:

```
MYVAR="A new variable!"
```

- New variable MYVAR can be accessed as $MYVAR

```
echo $MYVAR
```

Output:

```
A new variable!
```

- Note: no spaces around equals sign when setting variable.

Types

- Bash variables are untyped.
- Bash variables are character strings essentially.
- But, depending on context, Bash permits arithmetic operations and comparisons on variables.

```
$ nvar=20
$ let "nvar += 1"
$ echo $nvar
21

# character variable
$ cvar='A'
$ echo $cvar
A

# string variable
$ str="Hello!"
$ echo $str
```

Declare built-in

Using a declare statement, we can limit the value assignment to variables.

```
# -r represents readonly variable
$ declare -r rvar=1
$ echo "rvar = $rvar"
rvar=1
$ (( rvar++ ))
bash: var1: readonly variable

# -i declares an integer
declare -i number

# -a declares an array
declare -a values

# -f declares a function
declare -f func
```

Hello World script

- Create a new file (hello.sh).

```
#!/bin/bash
echo "Hello World!"
```

- Set file permission to allow execution of script.

```
$ chmod u+x hello.sh
```

- Execute the script.

```
$ ./hello.sh
Hello World!
```

See Lecture 3 on **chmod** command.

Hello World - String Version

```
#!/bin/bash
STRING="Hello again, world!"
echo $STRING
```

- Set permissions and run script:

```
$ chmod u+x hello2.sh && ./hello2.sh
Hello again, world!
```

- Programs *chmod* and *hello2.sh* are chained using &&.
- && is a control operator of the shell.
- *chmod* must be successful for *hello2.sh* to be executed.

Command chaining

- Other way to chain commands is to use a semicolon (;)

```
$ chmod u+x hello2.sh;./hello2.sh
Hello again, world!
```

- However, the second command is executed irrespective of the status of the first command.

```
$ chmod b+x hello2.sh;./hello2.sh
chmod: invalid mode:    b + x
Try 'chmod --help' for more information.
Hello again, world!
```

  - We add the execute permission for the user.
  - *chmod* and *hello2.sh* are chained using a semicolon (;)
  - The first program fails but the second program succeeds.

Command chaining

- We remove the execute permission for the user.

```
$ chmod u-x hello2.sh
```

- *chmod* and *hello2.sh* are chained using a semicolon (;)

```
$ chmod b+x hello2.sh;./hello2.sh
chmod: invalid mode:    b + x
Try 'chmod --help' for more information.
bash: ./hello2.sh: Permission denied
```

- Both are executed and both fail.

Shell Command Substitution

To use the output of a command within a script, set it apart using backticks `

```
`command`
$(command)
```

For example:

```
#!/bin/bash
echo date
echo `date`
DATE=$(date)
echo $DATE
```

Output:

```
date
Wed Mar 28 06:23:06 IST 2018
Wed Mar 28 06:23:13 IST 2018
```

A Backup Script

```
#!/bin/bash
tar -czf $HOME/docs-$(date +%F).tar.gz $HOME
```

- $HOME is a shell environment variable (to be covered in the following lecture).
- It contains the path of your home directory.
- Will create a backup file "docs-2018-03-28.tar.gz" in the $HOME directory.
- See `man date` for more on arguments to date.
- date %F prints date in the format (%Y-%m-%d)
- **$(date +%F)** executes the command **date +%F**.

Control flow using conditionals

Flow control using if/Then/Else

- The most compact syntax of the if command is:

```
if TEST-EXPR; then CONDITIONAL-COMMANDS; fi
```

- TEST-EXPR list is run first - it can be either a primary or any command with an exit status.
- If the exit status is true, CONDITIONAL-COMMANDS are executed.
- Otherwise, nothing happens.

Tests

- The command most frequently used with **if** is **test**.
- Tests have the following form:

```
test EXPRESSION
```

  Or:

```
[ EXPRESSION ]
[[ EXPRESSION ]]
```

- Note the spaces before/after [ ].
- [[]] is an enhanced replacement for test.

Exit status

```
#!/bin/bash
cd ${HOME}
if [ $? -eq 0 ]
  then echo  Last command exited cleanly!
fi
```

- $? checks the exit status of the last run program, “cd $HOME”.
- $? is an integer in the range 0-255.
- By convention, 0 indicates success.

if/then/else/fi

```
#!/bin/bash
cd ${HOME}
if [ $? -eq 0 ]
then echo "Last command exited cleanly!"
else echo "Uh-oh - non-zero exit status!"
fi
```

You need to use semicolons after each statement if placing commands one after another on the same line.

```
$ x=10
$ if [ $x -eq 100 ]; then echo "equals 100"; else echo "does
    not equal 100"; fi
OUTPUT: does not equal 100
```

if/then/else/fi full form

```
if TEST-COMMANDS;
then
  CONSEQUENT-COMMANDS;

elif MORE-TEST-COMMANDS;
then
  MORE-CONSEQUENT-COMMANDS;

else
  ALTERNATE-COMMANDS;
fi
```

File primaries / File expressions

- Scripts are frequently used to operate on files.
- Here is a short selection:

Expression	Returns true if:
[-e FILE]	File exists
[-r FILE]	Exists and is readable
[-s FILE]	Exists and has size > 0
[-w FILE]	Exists and is writable
[-O FILE]	Exists and is owned by you
[-G FILE]	Exists and is owned by your group

File expression example

```
#!/bin/bash
FILE=$HOME/.bashrc
if [ -e "$FILE" ]; then
    if [ -r "$FILE" ]; then
        echo "$FILE is a readable"
    fi
    if [ -w "$FILE" ]; then
        echo "$FILE is a writable"
    fi
    if [ -O "$FILE" ]; then
        echo "$FILE is owned by $USER"
    fi
else
    echo "$FILE does not exist"
    exit 1
fi
exit 0
```

- $HOME, $USER are bash environment variables (discussed in the next lecture).
- The *exit* command accepts a single, optional argument, which is the script's exit status.
- *'exit 0'* is same as just calling *exit*.

String expressions

Expression	Returns true if:
[string]	The *string* is not null.
[-n string]	The length of *string* is greater than zero.
[-z string]	The length of *string* is zero.
[string1 == string2]	string1 and string2 are equal.
[string1 != string2]	string1 and string2 are equal.

String expression example

```
#!/bin/bash
CAPITAL="DUBLIN"
if [ "$CAPITAL" == "CORK" ]; then
    echo "The capital is CORK."
elif [ "$CAPITAL" == "GALWAY" ];
    then
    echo "The capital is GALWAY."
elif [ "$CAPITAL" == "DUBLIN" ];
    then
    echo "The capital is DUBLIN."
else
   echo "There is no capital." >&2
   exit 1
fi
exit
```

- Note the use of redirection $>$ &2.
- It sends the error message to standard error.

Integer expressions

Expression	Returns true if:
[integer1 -eq integer2]	*integer1* is equal to *integer2*.
[integer1 -ne integer2]	*integer1* is not equal to *integer2*.
[integer1 -le integer2]	*integer1* is less than or equal to *integer2*.
[integer1 -lt integer2]	*integer1* is less than *integer2*.
[integer1 -ge integer2]	*integer1* is greater than equal to *integer2*.
[integer1 -gt integer2]	*integer1* is greater than *integer2*.

Integer expression example

```
#!/bin/bash

VAL=42

if [ $VAL -eq 0 ]; then
    echo "VAL is 0."
elif [ $VAL -gt 0 -a $VAL -le 25 ]; then
    echo "VAL is greater than 0 and less
        than 25."
elif [ $VAL -gt 25 -a $VAL -le 50 ]; then
    echo "VAL is greater than 25 and less
        than 50."
else
    echo "VAL is greater than 50 and equals
         $VAL."
fi
exit
```

- Note the use of logical operator "-a" to combine expressions.

Combining expressions

Operation	test	[[]]
AND	-a	&&
OR	-o	||
NOT	!	!

```
#!/bin/bash
MIN=10
MAX=50
VAL=25
if [ $VAL -ge $MIN -a  $VAL -le $MAX ]; then
    echo "$VAL is between $MIN and $MAX."
else
    echo "$VAL is out of range."
fi
```

Command-line Arguments

- Can pass arguments to your script arguments just as you pass to a program (C, Java, etc)
- Arguments assigned to variables $1, $2, $3...
- $0 stores the name of the script.
- $# gives the number of arguments.

```
#!/bin/bash
echo "\$1 = $1, \$2=$2"
```

Run:

```
shell> chmod u+x arg.sh
shell> ./arg.sh hello world
shell> $1 = hello, $2 = world
```

Test number of arguments

```
#! /bin/bash
PROGRAM=$(basename $0)
if [ ! $# -eq 3 ]; then
    echo Correct usage: $PROGRAM arg1 arg2 arg3
    exit
fi
```

- In the example, if the script is not given 3 arguments it will alert the user and exit.
- The basename command removes the leading portion of the pathname from the script.
- $# gives the number of arguments.
- Note the use of negation operator (!) in the *if* statement.

Recap of today's lecture

- Naming variables
- Using commands

  `` `command` or $(command) ``

- "test" command
  - Primary expressions ([! EXPR], [EXPR1 -a EXPR2], ...)
  - File primaries ([-e FILE], [-r FILE], ...)
- Conditionals (if/then/elif/else/fi)
- Command-line arguments

Bash environment, Integer arithmetic, Loops to follow...

Q & A