CS2043 - Unix Tools & Scripting

Cornell University, Spring 2014[1]

Instructor: Bruno Abrahao

February 21, 2014

[1] Slides evolved from previous versions by Hussam Abu-Libdeh and David Slater

Announcement

HW 4 is out. Due Friday, February 28, 2014 at 11:59PM.

Wrapping up AWK

Split

```
n = split(string, array, separator)
```

- Splits fields of `string` separated by `separator` and places them into `array`.
- `n` is the resulting number of fields
- default separator is whitespace

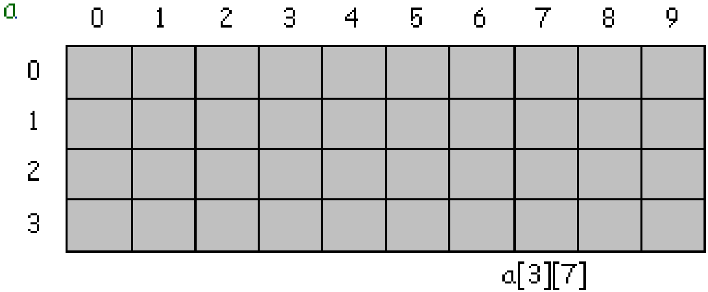
Fun

Let's reverse the order of a list of names for all groups in `restaurants.txt` !

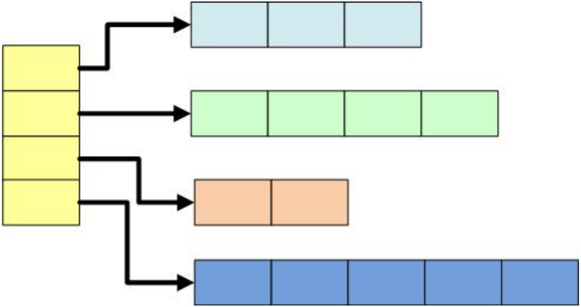
Fake Multidimensional Arrays!

```
array[key1, key2, ...]
```

This is not what AWK is doing



This is not what AWK is doing either



Fake Multidimensional Arrays!

```
array[3, 6]
```

- Multidimensional subscripts are individual strings concatenated.
- "3" and "6" in the example are concatenated together separated by the value of the system variable SUBSEP

Tests

```
if ((i, j) in array)
```

- This tests whether the key i SUBSEP j exists in the array.

That makes life a little harder!

```
for (item in array)
```

- Each item has the form `i SUBSEP j`
- You must use `split()` to extract individual subscript components.

```
n= split(item, subscr, SUBSEP)

subscr[1] # first component
subscr[2] # second component
...
subscr[n] # n-th component
```

Length of an Array

-
  ```
  awk ’BEGIN {A= "Ithaca is Gorges";print
  length(A)}’
  ```

  prints ”16”

-
  ```
  awk ’BEGIN {split("Ithaca is Gorges",A);print
  length(A)}’
  ```

  prints ”3”

Full-fledged shell scripting preliminaries

Scripting

Next week we will discuss bash scripting. Before we begin, will discuss a few preliminaries.

Agenda:

- Shell variables
- Shell expansion
- Quotes in bash
- Running commands sequentially & exit codes
- Passing arguments to scripts

Variables

- To get anything done we need variables.
- To read the values in variables, precede their names by a dollar sign ($).
- We can print the contents of any variable using the `echo` command
- Two types of variables: **Local** and **Environment**.

Example:

```
echo $SHELL
/bin/bash
```

Local Variables

Local variables exist only in the current shell:

Example:

```
~$ x=3
~$ echo $x
3
```

Note: Bash is picky! There cannot be a space after the x nor before the 3!

Environment Variables

- Used by the system to define aspects of operation.
- The Shell passes a copy of environment variables to its child processes
    - Every command that is launched from the shell becomes its child.
    - If you kill the parent, all its children will die.
    - There is a way to decouple a process from the shell (more on this later).

Environment Variables

- Examples:
  - $SHELL - which shell will be used by default
  - $PATH - a list of directories to search for binaries
  - $HOSTNAME - the hostname of the machine
  - $HOME - current user's home directory
- To get a list of all current environment variables type `env`

New Environment Variable:

To set a new environment variable use `export`

```
~$ export X=3
~$ echo $X
3
```

Again: NO spaces around the = sign.

A Word About the Difference

Environment variables are passed as copies across shell invocations while local variables are not:

Local Variable:

```
~$ x=3
~$ echo $x
3
~$ bash
~$ echo $x

~$
```

Environment Variable:

```
~$ export x=myvalue
~$ echo $x
myvalue
~$ bash
~$ echo $x
myvalue
~$
```

Environment Variables Again...

If the environment variable is changed in the new shell it is **not** changed for the old shell (caller)

Example:

```
∼$ export x=value1
∼$ bash
∼$ echo $x
value1
∼$ export x=value2
∼$ exit
∼$ echo $x
value1
```

Listing and Removing Variables

- `env` - displays all environment variables
- `set` - displays all shell/local variables
- `unset name` - remove a shell variable
- `unsetenv name` - remove an environment variable

Environment Variables Example - Modifying your Prompt

The environment variable $PS1 stores your default prompt. You can modify this variable to spruce up your prompt if you like:

Example

First `echo $PS1` to see its current value
`\s-\v\$` (default)

It consists mostly of backslash-escaped special characters, like `\s` (name of shell) and `\v` (version of bash). There are a whole bunch of options, which can be found at

http://www.gnu.org/software/bash/manual/bashref.html#Printing-a-Prompt

Environment Variables Example - Modifying your Prompt

Once you have a prompt you like, set your $PS1 variable

Define your prompt

```
∼$ export PS1="New Prompt String"
```

- Type this line at the command prompt to temporarily change your prompt (good for testing)
- Add this line to `~/.bashrc` or `~/.bash_profiles` to make the change permanent.

Note: Parentheses must be used to invoke the \ characters.

Examples

```
PS1="\u \w \t_"  ⇒ abrahao ~ 12:12:12_
PS1="\W \j \d\:" ⇒ ~ 0 Oct 02:
```

Environment Variables Example - Where is my program?

The environment variable $PATH lists the directories to search for binaries

Example

```
echo $PATH
/Users/abrahao/bin:/usr/bin:
/bin:/usr/sbin:/sbin:/usr/local/bin
```

Where is my program?

- If it's in the path, use the command `which`
- Else, use `locate`

The database `locate` uses needs to be updated regularly by the super user.

- Linux: `updatedb`
- Mac OS X `/usr/libexec/locate.updatedb`

Shell Expansions

The shell interprets $ in a special way.

- If var is a variable, then `$var` is the value stored in the variable var.
- If `cmd` is a command, then `$(cmd)` is translated to the result of the command `cmd`. (Same as backticks)

Example

```
~$ echo $USER
abrahao
~$ echo $(pwd)
/home/abrahao
```

Arithmetic Expansion

The shell will expand arithmetic expressions that are encased in $((expression))

Examples

```
~$ echo $((2+3))
5
~$ echo $((2 < 3))
1
~$ echo $((x++))
3
```

And many more.

Note: the post-increment by 1 operation (++) only works on variables

Quotes

3 different types of quotes to enclose strings, and they have different meanings:

- Single quotes ('): preserves the literal value of each character. A single quote may not occur between single quotes, even when preceded by a backslash.
- Double quotes (“): preserves the literal value of all characters within the quotes, with the exception of $ ' \ !
- Back quotes (\`): Executes the command within the quotes. Like $().

Quotes

Example

```
~$ echo "$USER owes me $ 1.00"
abrahao owes me $ 1.00

~$ echo ’$USER owes me $ 1.00’
$USER owes me $ 1.00

~$ echo "I am $USER and today is `date`"
I am abrahao and today is Wed Feb 11 16:23:30 EST 2009
```

Running Commands Sequentially

The && Operator

```
<command1> && <command2>
```

- `command2` executes **only if** `command1` executes successfully

The ; Operator

```
<command1> ; <command2>
```

- Immediately after command1 completes, execute command2

Examples

Example:

```
mkdir photos && mv *.jpg photos/
```

- Creates a directory and moves all jpegs into it

Example: hello.sh

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

Set your permissions and run:

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

Exit Codes

The command after a && only executes if the first command is successful, so how does the Shell know?

- When a command exits it always sends the shell an exit code (number between 0 and 255)
- The exit code is stored in the variable $?
- An exit code of 0 means the command succeeded
- The man page for each command tells you precisely what exit codes can be returned

Exit Codes

Example:

```
~$ ls ~/Documents/cs2043
2003 2004 2007 2008 2009
~$ echo $?
0
```

Example:

```
~$ grep ’Gorges’ ~/Documents/Ithaca.txt
Ithaca is Gorges!
~$ echo $?
0
```

Example:

```
~$ grep ’George’ ~/Documents/Ithaca.txt
~$ echo $?
1
```

Script Comments

Scripts begin with a **shebang** (#!), followed by the full path of the interpreter we'd like to use: e.g., `/bin/bash`

- Any line that begins with # (except the shebang) is a comment
- Comments are ignored during execution - they serve only to make your code more readable.

Remember: you know what your code does today, but you won't quite remember next month.
Remember 2: Other readers have limited knowledge of what your script is supposed to do.

Passing arguments to scripts

When we pass arguments to a bash script, we can access them in a very simple way:

- $1, $2, ... $10, $11 : are the values of the first, second etc arguments
- $0 : The name of the script
- $# : The number of arguments
- $* : All the arguments, "$*" expands to "$1 $2 ... $n",
- $@ : All the arguments, "$@" expands to "$1" "$2" ... "$n"
- You almost always want to use $@
- $? : Exit code of the last program executed
- $$: current process id.

Simple Example

multi.sh

```
#!  /bin/bash/
echo $(( $1 * $2 ))
```

- Usage: ./multi.sh 5 10
- Returns first argument multiplied by second argument

uptolow.sh

```
#!  /bin/bash
tr ’[A-Z]’ ’[a-z]’ < $1 >  $2
```

- Usage: `./uptolow.sh file filelow`
- translates all upper case letters in `file` to lowercase and writes to `filelow`