Expansion, Quoting, and Escaping

OS Practice CSC 420 Spring 2024

Table of Contents

README

- You can code along with the file tinyurl.com/5-expansion-org

  You can save it to an Org-mode file `5_expansion.org`

- The solutions and results herein were all obtained on a Lenovo laptop running Linux Mint 21.3 with 4 Intel Core i3-6006U CPUs.
- This lab is based on chapter 7 (pp. 61-72) of Shotts (2019).
- You can get the completed file from `tinyurl.com/5_redirection.org`

What we'll cover:

- Pathname expansion or 'globbing' of filenames, regular and hidden
- Tilde ~ expansion
- Arithmetic expansion with `$(( ))`
- Brace expansion with `{ }`
- Parameter expansion with `$`
- Command substitution
- Quoting with double and single quotes
- Escaping characters

Pathname expansion ('globbing') with `echo`

- Expansion is the process of substitution that the shell performs whenever you press `ENTER` while you're on the shell.
- 'Globbing' is a part of many languages (e.g. the database language and RDBMS SQLite has a `GLOB` expansion parameter).

- `echo` is a shell-builtin (check `type echo`) that prints its text argument on `stdout`. There's both a `help` for the built-in and a `man` page for the standalone program.
- `echo *` does not print `*` but the list of files in `$PWD` instead: the shell expands the `*` into "match any filenames in the current directory", and the `echo` command never sees the `*`:

```
echo *
```

```
1_fundamentals.org 1_fundamentals_practice.org 2_manipulate_practice.org 3_shell_c
```

- Let's try a few more pathname expansions: in the case of `D*`, there's nothing to find, so the string `D*` is printed, in `$HOME`, there is:

```
echo D* ~/D*
```

```
D* /home/marcus/Desktop /home/marcus/Documents /home/marcus/Downloads
```

- Expansion does not just work from an initial character:

```
echo *s  # prints all files ending in `s`
echo ~/*p # prints all files in PWD ending in `p`
echo /usr/*/share # prints all files in /usr with a /share subdirectory
```

```
5_expansion.fls assignments Photos
/home/marcus/Desktop
/usr/local/share /usr/racket/share
```

- In the following argument, `[[:lower;]]` matches any single character that is a lowercase character from the current language *locale*. It is a so-called POSIX (standard) character class for regex.

```
echo ~/.[[:upper:]]*   # all hidden files that start with a upper case char
```

```
/home/marcus/.Xauthority
```

- 'Regex',or regular expressions, are a key `bash` scripting skill. The quickest way to learn is to go to `regex101.com` and get to work.

Pathname expansion of hidden files

- Filenames that begin with a period character `.` are hidden: to see them e.g. with `ls` you need to use the `-a` flag.
- `echo` respects that behavior: `echo *` does not print hidden files.

```
touch .hidden ..very_hidden
```

```
echo .*
```

```
. .. .hidden ..very_hidden
```

- But the period is not like other characters: the current and the parent directory (`./` and `../`) are also printed - and they're not hidden:

```
ls -d .* | less  # list all files starting with a period
```

```
.
..
.hidden
..very_hidden
```

- A well performing expansion of hidden files has to use a different pattern:

```
echo .[!.]*  # files beginning with a period followed by other characters
```

```
.hidden
```

- Read `.[!.]*` as "starts with . but not .. followed by any other characters". This will not find filenames starting with multiple periods - this is too hard with `echo` - see however `glob(3)`.

```
touch ..very_hidden
ls -ad .*
echo
echo .[!.]*
```

```
.
..
.hidden
..very_hidden

.hidden
```

Tilde expansion

- The tilde character (~) has a special meaning: it expands into the name of the home directory of the named (or current) user:

```
echo ~      # expands the home directory of current user
echo ~root
```

```
/home/marcus
/root
```

Arithmetic expansion

- The shell can also do arithmetic: we can use it as a calculator, both on the command line and in shell scripts:

```
echo $((2+2))
```

```
4
```

- The syntax (where *expression* consists of values and arithmetic operators):

```
$((expression))
```

- Alas, arithmetic expansion only supports integers but it can add, subtract, multiply, divide, modulo and exponentiate, spaces are insignificant, and expressions can be nested.
- Example: 5 squared by 3

```
echo $((5**2))
echo $(($((5**2)) * 3))
```

```
25
75
```

- Example: 5 divided by 2

```
echo Five divided by two equals $((5/2))
echo with $((5%2)) left over
```

```
Five divided by two equals 2
with 1 left over
```

```
echo $((5.2 + 1))   # decimal not accepted as input
```

Brace expansion

- With brace expansion, you can create multiple text strings from a pattern with braces:

```
echo Front-{A,B,C}-Back
```

```
Front-A-Back Front-B-Back Front-C-Back
```

- Patterns to be brace expanded may contain a leading portion (*preamble*) and a trailing portion (*postscript*). The brace expression can be a comma separated list of strings or a range of integers or single characters.

No unquoted whitespace is allowed.

- Examples:

```
echo Number_{1..5} # range of numbers
echo {01..15}  # zero padding
echo {001..15} # double zero padding
echo {Z..A}  # invert alphabet letters
```

```
Number_1 Number_2 Number_3 Number_4 Number_5
01 02 03 04 05 06 07 08 09 10 11 12 13 14 15
001 002 003 004 005 006 007 008 009 010 011 012 013 014 015
Z Y X W V U T S R Q P O N M L K J I H G F E D C B A
```

- Brace expansions may be nested:

```
echo a{A{1,2},B{3,4}}b
```

```
aA1b aA2b aB3b aB4b
```

- What is brace expansion good for? Most commonly for making lists: for example if you wanted to organize a large collection of images into years and months:

```
if [ -d Photos ]; then
    rm -rf Photos;
fi
mkdir -v Photos
cd Photos
mkdir -v {2007..2009}-{01..12}
```

```
mkdir: created directory 'Photos'
mkdir: created directory '2007-01'
mkdir: created directory '2007-02'
mkdir: created directory '2007-03'
mkdir: created directory '2007-04'
mkdir: created directory '2007-05'
mkdir: created directory '2007-06'
mkdir: created directory '2007-07'
mkdir: created directory '2007-08'
mkdir: created directory '2007-09'
mkdir: created directory '2007-10'
mkdir: created directory '2007-11'
mkdir: created directory '2007-12'
mkdir: created directory '2008-01'
mkdir: created directory '2008-02'
mkdir: created directory '2008-03'
mkdir: created directory '2008-04'
mkdir: created directory '2008-05'
mkdir: created directory '2008-06'
mkdir: created directory '2008-07'
mkdir: created directory '2008-08'
mkdir: created directory '2008-09'
mkdir: created directory '2008-10'
mkdir: created directory '2008-11'
mkdir: created directory '2008-12'
```

```
mkdir: created directory '2009-01'
mkdir: created directory '2009-02'
mkdir: created directory '2009-03'
mkdir: created directory '2009-04'
mkdir: created directory '2009-05'
mkdir: created directory '2009-06'
mkdir: created directory '2009-07'
mkdir: created directory '2009-08'
mkdir: created directory '2009-09'
mkdir: created directory '2009-10'
mkdir: created directory '2009-11'
mkdir: created directory '2009-12'
```

    - More information about this (and other) expansion methods is in the `bash(1)` man page or in the online reference manual for bash.

- Python has adopted this notation for the formatted or *f-string*:

```
greeting = "world"

print(f"Hello, {greeting}")
```

- Emacs tables are also using this feature to turn a table into a spreadsheet. The following table computes the sum of two numbers. To turn formula debugging on/off, use `C-c }`

Number	Number	Sum
100	500	600
10	20	200
110	2000	400

  Formulas are achieved with Lisp (the language Emacs is written in). It's very handy to have active tables in Org-mode files.

Parameter expansion

- Expansion of environment parameters is useful in scripts:

```
echo $USER  $HOME $PWD
```

```
marcus /home/marcus /home/marcus/GitHub/os24/org
```

- To see a list of environment variables:

```
printenv
```

```
SHELL=/bin/bash
SESSION_MANAGER=local/marcus-Vostro-3470:@/tmp/.ICE-unix/1175,unix/marcus-Vostro-3
QT_ACCESSIBILITY=1
XDG_CONFIG_DIRS=/etc/xdg/xdg-cinnamon:/etc/xdg
```

```
XDG_SESSION_PATH=/org/freedesktop/DisplayManager/Session0
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
LANGUAGE=en_US
MANDATORY_PATH=/usr/share/gconf/cinnamon.mandatory.path
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
DESKTOP_SESSION=cinnamon
GTK_MODULES=gail:atk-bridge
XDG_SEAT=seat0
PWD=/home/marcus/GitHub/os24/org
XDG_SESSION_DESKTOP=cinnamon
LOGNAME=marcus
QT_QPA_PLATFORMTHEME=qt5ct
XDG_SESSION_TYPE=x11
GPG_AGENT_INFO=/run/user/1000/gnupg/S.gpg-agent:0:1
_=/usr/bin/printenv
XAUTHORITY=/home/marcus/.Xauthority
XDG_GREETER_DATA_DIR=/var/lib/lightdm-data/marcus
GDM_LANG=en_US
HOME=/home/marcus
LANG=en_US.UTF-8
XDG_CURRENT_DESKTOP=X-Cinnamon
XDG_SEAT_PATH=/org/freedesktop/DisplayManager/Seat0
XDG_SESSION_CLASS=user
TERM=dumb
DEFAULTS_PATH=/usr/share/gconf/cinnamon.default.path
USER=marcus
DISPLAY=:0
SHLVL=1
XDG_VTNR=7
DESKTOP_AUTOSTART_ID=10a4adb1a05fa1129e171192102981329700000011750006
XDG_SESSION_ID=c1
XDG_RUNTIME_DIR=/run/user/1000
GTK3_MODULES=xapp-gtk3-module
XDG_DATA_DIRS=/usr/share/cinnamon:/usr/share/gnome:/home/marcus/.local/share/flatp
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/
GDMSESSION=cinnamon
DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/bus
GIO_LAUNCHED_DESKTOP_FILE_PID=1765
GIO_LAUNCHED_DESKTOP_FILE=/home/marcus/.config/autostart/Emacs Server.desktop
```

- Mistyping a pattern will result in an empty string:

```
echo $USRE  $pwd $Home
```

```

```

Command substitution

- You can also use the output of a command as a substitution:

```
echo $(ls)  # list command printed (on one line)
```

```
1_fundamentals.org 1_fundamentals_practice.org 2_manipulate_practice.org 3_shell_c
```

- You can use this e.g. to get the listing of a command without knowing its full pathname:

```
ls -l $(which echo) # list the echo program
```

```
-rwxr-xr-x 1 root root 35128 Feb  7 21:46 /usr/bin/echo
```

- You can build pipelines, too: how many files are in `$PWD` - use `ls` and `wc`

```
echo $(ls -a | wc -l)  # how many files are in $PWD
```

```
36
```

- Find all `zip` executables in `/usr/bin` and show the `file` characteristics of the top 10 results:

```
file $(ls -d /usr/bin/* | grep zip | head)
```

```
/usr/bin/bunzip2:           ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV
/usr/bin/bzip2:             ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV
/usr/bin/bzip2recover:      ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV
/usr/bin/funzip:            ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV
/usr/bin/gpg-zip:           POSIX shell script, ASCII text executable
/usr/bin/gunzip:            POSIX shell script, ASCII text executable
/usr/bin/gzip:              ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV
/usr/bin/lzip:              symbolic link to /etc/alternatives/lzip
/usr/bin/lzip-compressor:   symbolic link to /etc/alternatives/lzip-compressor
/usr/bin/lzip-decompressor: symbolic link to /etc/alternatives/lzip-decompressor
```

- `bash` supports an older version of command substitution with backquotes instead of `$()`

```
ls -l `which cp`
echo `ls -l | wc -l`
```

```
-rwxr-xr-x 1 root root 141832 Feb  7 21:46 /usr/bin/cp
33
```

Quoting

- In this example, whitespace is removed from the `echo` command argument list: the shell performs word splitting to do this.

```
echo this is a    test
```

```
this is a test
```

- In the next example, `$1` is replaced by an empty string because it is undefined.

```
echo The total is $100.00
echo $1
```

```
The total is 00.00
```

- The shell provides a mechanism called *quoting* to selectively suppress unwanted expansions.

Double quotes

- Text inside double quotes is treated as ordinary characters: all special characters except `$`, `\` and `` ` `` lose their special meaning.
- Word splitting, pathname expansion, tilde expansion, brace expansion are suppressed, but parameter expansion, arithmetic expansion, and command substitution are still carried out.

- With double quotes, we can handle filenames containing spaces (which Windows likes so much) - otherwise word splitting would mess us up.

  ```
  ls -l two words.txt
  ```

  Yields the result:

  ```
  ls: cannot access 'two': No such file or directory
  ls: cannot access 'words.txt': No such file or directory
  ```

- To fix it, use double quotes (works only if you have a file `two words.txt`):

  ```
  > 'two words.txt'
  ls -l "two words.txt"
  ```

  ```
  -rw-rw-r-- 1 marcus marcus 0 Apr  1 22:24 two words.txt
  ```

- You can even repair the damage:

  ```
  mv -v "two words.txt" two_words.txt
  ls -l two*
  ```

  ```
  renamed 'two words.txt' -> 'two_words.txt'
  -rw-rw-r-- 1 marcus marcus 0 Apr  1 22:24 two_words.txt
  ```

- Parameter, arithmetic expansion and command substitution still work:

  ```
  echo "$USER"    # parameter expansion inside double quotes
  echo "$((2+2))" # arithmetic expansion inside double quotes
  echo "$(date)"  # command substitution inside double quotes
  ```

```
marcus
4
Mon Apr  1 10:24:47 PM CDT 2024
```

Word splitting

- Word splitting looks for spaces, tabs, and newlines, and treats them as word delimiters - unquoted, they are simply ignored.
- When we put a text with whitespace in quotes, word splitting is suppressed, and the whitespace becomes part of the argument:

```
echo this is a     test   # with wordsplitting
echo "this is a     test" # without wordsplitting
```

```
this is a test
this is a     test
```

- Command substitution suffers some side effects. Compare the two versions:

```
echo $(cal)    # command substitution with word splitting
echo "$(cal)"  # command substitution without word splitting
```

```
April 2024 Su Mo Tu We Th Fr Sa 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
     April 2024
Su Mo Tu We Th Fr Sa
    1  2  3  4  5  6
 7  8  9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30
```

Single quotes

- To suppress *all* expansions, use single quotes
- Compare unquoted and quoted commands:

```
echo text      # print argument
echo ~/*.txt   # pathname expansion
echo a{1..4}b  # brace expansion
echo $(echo hello) # command substitution
echo $((2+2))  # arithmetic expansion
echo $USER     # parameter expansion
```

```
text
/home/marcus/superuser.txt
a1b a2b a3b a4b
```

```
hello
4
marcus
```

- Unquoted:

```
echo text ~/*.txt a{1..4}b $(echo hello) $((2+2)) $USER
```

```
text /home/marcus/superuser.txt a1b a2b a3b a4b hello 4 marcus
```

- Double quotes:

```
echo "text ~/*.txt a{1..4}b $(echo hello) $((2+2)) $USER"
```

```
text ~/*.txt a{1..4}b hello 4 marcus
```

- Single quotes:

```
echo 'text ~/*.txt a{1..4}b $(echo hello) $((2+2)) $USER'
```

```
text ~/*.txt a{1..4}b $(echo hello) $((2+2)) $USER
```

```
touch 'n3wt0N1  phy31cc!!.txt'   # don't do this though
ls -l n3w*
```

```
-rw-rw-r-- 1 marcus marcus 0 Apr  1 22:24 n3wt0N1  phy31cc!!.txt
```

Escaping characters

- To quote a single character, precede it with a backslash `\` a so-called *escape character* (it escapes the normal encoding).
- This is often done in quotes to prevent an expansion:

```
echo "The balance for user $USER is \$100.00" # with escaping
echo 'The balance for user $USER is \$100.00' # no escaping
```

```
The balance for user marcus is $100.00
The balance for user $USER is \$100.00
```

- You can also escape the special meaning of characters in filenames:

```
touch "bad&filename"; rm good_filename
ls *filename
```

```
mv -v bad\&filename good_filename
ls *filename
```

```
bad&filename
renamed 'bad&filename' -> 'good_filename'
good_filename
```

- To escape a backslash itself, escape it with a backslash. With single quotes however, it loses its meaning:

```
echo "The home of root is not \\ but /"  # in Linux
echo 'The home of root is not \\ but /'
```

```
The home of root is not \ but /
The home of root is not \\ but /
```

- The backslash is also part of the notation to represent *control codes* to transmit commands to Teletype (`tty`) or terminal devices:

Escape sequence	Meaning
\a	Bell (beep)
\b	Backspace
\n	Newline
\r	Carriage return
\t	Tab

- This use of \ comes from the C programming language and has been adopted by many others.
- To interpret these codes, add the `-e` flag to `echo` or place them inside `$''`.

```
sleep 10; echo -e "Time's up\a"  # countdown using sleep(1)
sleep 10; echo -e "Time's up again" $'\a'
```

SUMMARY

Command	Explanation
`echo *`	Lists all files in the current directory by expanding *
`echo .*`	Lists hidden files, including . and ..
`echo .[!.]*`	Lists hidden files excluding . and ..
`echo ~`	Prints the current user's home directory
`echo $((2+2))`	Performs arithmetic expansion to print the result of 2+2
`echo $(ls)`	Uses command substitution to list directory contents in a single line
`echo "text"`	Prints the text as is, treating everything inside quotes as a single string

Command	Explanation
`echo 'text'`	Similar to double quotes, but prevents all expansions
`echo \$variable`	Escapes $ to prevent variable expansion and print $variable literally
`echo {2007..2009}`	Uses brace expansion to print a sequence from 2007 to 2009
`echo {A..C}`	Uses brace expansion to print letters A through C
`echo $USER`	Prints the current user's username by expanding the USER environment variable
`echo $(echo hello)`	Nested command substitution to print "hello"
`echo -e "text\nnew"`	Uses -e option to enable interpretation of backslash escapes like \n for newline

Author: Marcus Birkenkrahe
Created: 2024-04-01 Mon 22:24
Validate