Part One

Scripting in the Bourne-Again Shell (bash)

Part 1: Outline

1. file expansion
2. variables, redirection, and substitution
3. types of quotation
4. shellscript
5. getting arguments from terminal
6. for loops and if statements

File Expansion

File Expansion (1)

Filename expansion. When bash sees

```
$ ls *.txt
```

It expands the command to, for example:

```
$ ls a.txt b.txt
```

This is the simplest type of file expansion

File expansion (2)

A fairly complete list of expansion terms

*	Zero or more characters
?	One of any character
\	Escape following special chararacter
[xyz]	Matches any of the enclosed chararacters
{a,b,c}	Matches any of the enclosed strings

File expansion: examples

```
*.faa
*.{faa,fna}
$ ls ?[frs]??[b-f]*.{txt,py}
Africa-stuff.txt  psyceval.py
$ rm ??_[0-9]*.gff
```

Escaping special chars

```
$ rm Harry Potter.pdf  # two files
$ rm Harry\ Potter.pdf # escape space
$ rm \*\&\?.txt # removes file ‘*&?.txt’
```

Never put spaces in filenames. Generally only use letters, ‘.’, ‘_’, and ‘-’

Variables, redirection and substitution

Variables

Declaring variables:

```
x='/path/to/some/file.txt'
head $x
```

Environmental variables:

```
echo $PATH
```

Command Substitution $()

Evaluate a command in a subshell, output appears as a variable

- an anonymous variable

```
rm $(grep -L ‘error’ *log)

echo `date`
rm `grep -L ‘error’ *.log`
```

Redirection

Given A and B are programs and f is a file

```
A | B     Pipe STDOUT from A to STDIN of B
A > f     Overwrite f with A's STDOUT
A >> f    Append A's STDOUT to end of f
A < f     Send contents of f to A's STDIN
A <<< s   Send contents of string f to A
```

Redirection Examples

```
# read a.txt into tr, write to b.txt
tr ‘a’ ‘b’ < a.txt > b.txt
# you can do the same thing like this
cat a.txt | tr ‘a’ ‘b’ > b.txt
# send a string into tr (see for yourself)
tr ‘a’ ‘b’ <<< ‘asasasdf’ > b.txt
echo ‘asasasdf’ | tr ‘a’ ‘b’ > b.txt
```

Process Substitution <()

Evaluates a command in a subshell, output appears as a *file*

- an anonymous file

```
# head reads from tail's stdout as if it were a file
head <(tail a.txt)
join <(sort a.txt) <(sort b.txt)
```

Quotation: single, double and ANSI C Quotes

Types of Quotes: Single

Returns the inside string *exactly*

```
$ x=5
$ echo ‘\t$x\n&*’
\t$x\n&*
```

Types of Quotes: Double

- No file expansion within quotes
- Special characters interpretation
- Variable interpolation

Quotes Example

```
$ ls
a.txt b.txt c.txt d.pdf
$ x=*.txt
$ ls $x
a.txt b.txt c.txt
$ ls “$x”
ls: cannot access “*.txt”: No such file or directory
$ ls ‘$x’
ls: cannot access “$x”: No such file or directory
```

Newline and TAB

\n - commonly represents a newline

\t - commonly represents a TAB

These are two very widely used special characters.

ANSI C Quotes ($’ ‘)

```
# Interprets \t as TAB
$ echo $’a\tb’
a	b
# Yay! Unicode emoticons
$ echo $’\U1F608\u2661 Windows’
□♡ Windows
# Chinese characters
$ echo $’\u7535\u8111’
电脑
```

String Concatenation

```
x=$HOME/src/git/auw
# Space MUST be quoted
x=’My home directory is:’$HOME
x=”$HOME/bin is in \$PATH”
# {} prevent premature concatenation
x=${x}.faa
```

- Adjacent strings are concatenated
- Where illegal characters follow variable, use {}

Quotes Overview

quote type	File Expansion	Variable Expansion	Special characters
None	YES	YES	NO
“ ”	NO	YES	NO
‘ ’	NO	NO	NO
$‘ ’	NO	NO	YES

Shellscripting

What are shellscripts

Anything you type into your terminal, can be pasted into a file and executed

The code in the shellscript is read line-by-line by the bash interpreter, just like the lines you type into your terminal

Hello World in shellscript

```
#!/bin/bash
echo “hello world”
```

- Copy the above two lines into a file
- Make it executable (**`chmod`** `755 hw.sh`)
- Call it (`./hw.sh`)

Hashbang (#!)

You need to tell the system what program should interpret your script

Syntax:

```
#! /path/to/executable
```

```
#! /bin/bash
```

Calling a script (example)

```
$ cat lsall.sh
  #!/bin/bash
  ls *
$ chmod 755 lsall.sh  # make executable
$ ./lsall.sh  # execute! (why ‘./’?)
```

Command line arguments

```
$ cat scr.sh
#!/bin/bash
echo “$2 $1 $3”
$ ./scr.sh 12 56 89
56 12 89
```

Arguments must be space-separated

For-loops and if statements

Bash for-loops

Syntax:

```
for x in <list>; do
    <code>
done
```

for-loop example

```
# *.fa will expand to space separated list
for q in *.fa; do
    blastp -query $q -db mydb > $q.output
done
```

blastp is a bioinformatics tool, which I won't discuss further

for-loop example (2)

```
# Find all pdfs that contain ‘Waldo’
for j in *.pdf; do
    lesspipe $j   |
        grep ‘Waldo' > /dev/null && echo $j
done

lesspipe  - extracts text data from almost anything
/dev/null - a place where output disappears
```

If-else statements

```
if [[ <condition> ]]; then
    <code>
elif [[ <condition> ]]; then
    <code>
else
    <code>
fi
```

Useful tests

```
-r file is readable
-f file exists*
-d directory exists
-s file exists and is not empty
-z test is a variable is empty
```

* **`-f`** tests for existence of a file, but it doesn't recognize anonymous files, so it prevents command substitution. Generally use **`-r`** instead.

Dying gracefully

```
# If myfile.txt doesn't exist, stop the script
if [[ ! -f myfile.txt ]]; then
   exit 1 # exit code 1 indicates error
fi
```

The spaces around the brackets matter!

for-loop (3)

```
# find any .mp3 files that are not real
for j in $(find Home/ -iname “*.mp3”); do
   if [[ ! $(file $j) =~ 'Audio' ]]; then
      echo $j
   fi
done
```

Example shellscript (1)

```
#!/bin/bash
# If the file is ASCII, then use normal less
if [[ $(file $1) =~ 'ASCII text' ]]; then
   less $1
# Otherwise run preprocessor
else
   lesspipe $1 | less
fi
# Doesn’t work on mac
```