

Lecture 3B:

Shell scripts



Lecture 3B outline

1. Using scripts
2. Script anatomy
3. Writing scripts

Shell scripts

A ***shell script*** is a file that contains a sequence of commands that can be executed by the Unix shell

```
1 #!/bin/bash
2 # store first argument into VAR
3 VAR=$1
4 # print VAR, with too much enthusiasm
5 echo $VAR!! | tr "[:lower:]" "[:upper:]"
```

shell script, *yell.sh*

```
$ ./yell.sh 'Hello, world'
HELLO, WORLD!!
```

calling *yell.sh*

When you should write or use a script?

Scripts are useful for tasks that

- need to be **reproduced** by others (or *yourself!*)
- are **complex** and/or **repetitious**
- are sensitive to **user error** (e.g. typos)
- rely heavily upon **programming constructs**, like variables, if-statements, for-loops, etc.
- adhere to **standard file formats**

Scripts vs. command line

Unix commands generally behave the same way, whether executed through the command line or through a script

Like the command line, scripts are executed:
(1) line-by-line, (2) top-to-bottom, (3) left-to-right

Complex problems often require programming constructs (*if-statements, for-loops*): easier to organize complex logic through a script

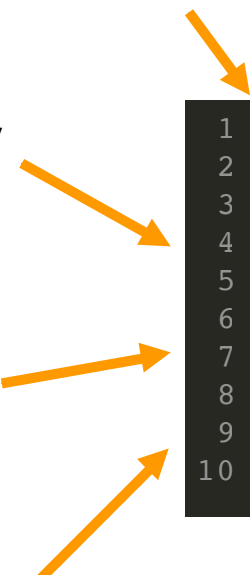
Anatomy of shell script

Hashbang (#!) designates
which shell program
should interpret script

Create variables \$FILE1
and \$FILE2 initialized by
arguments \$1 and \$2

Text after comment
(#) is ignored

Run *echo*, *cp*, *rm*
commands using variables
\$FILE1 and \$FILE2



```
1 #!/bin/sh
2
3 # set arguments to local variables
4 FILE1=$1
5 FILE2=$2
6
7 # run commands against FILE1 and FILE2
8 echo "received \'${FILE1}\' and \'${FILE2}\' as input"
9 cp ${FILE1} ${FILE2}"_copy.txt"
10 rm ${FILE1}
```

contents of *my_script.sh*

Executing a script

Scripts are run much like programs are run;
some scripts are written to accept
arguments and/or options

```
# call script
$ ./my_script.sh
$ sh my_script.sh

# call script with arguments
$ ./my_script.sh file1.txt file2.txt

# call script with arguments and options
$ ./my_script.sh --verbose file1.txt

# redirect script output to file
$ ./my_script.sh file1.txt file2.txt > output.txt

# use script in pipeline
$ find dir1 | ./my_script.sh file1.txt file2.txt > output.txt
```

Set file as executable

Permission must be granted to execute a file

Add 'x' to
permission
bitset

```
# file lacks execute permissions
$ ./process_files.sh
-bash: ./process_files.sh: Permission denied

# check file permissions
$ ls -lart process_files.sh
-rw-r--r--  1 mlandis  staff      0 Sep 20 22:06 process_files.sh

# add execute permissions (+x) to script
$ chmod +x process_files.sh

# we now see +x permissions for script
$ ls -lart process_files.sh
-rwxr-xr-x  1 mlandis  staff      0 Sep 20 22:06 test2.sh

# execute with no problems
$ ./process_files.sh
Processing files...
...done!
```


Variables

Variables exist in memory, and can store and report user-defined values

create \$MY_DIR
and \$MY_FILE as
**local (shell)
variables**

create new
variables in
terms of other
variables

\$HOME is an
**environment
variable**

```
1 #!/bin/sh
2
3 # you can define your own variables
4 MY_DIR=/home/mlandis/docs
5 MY_FILE=my_file.txt
6
7 # access the value of a variable using $
8 echo "Value of MY_FILE is ${MY_FILE}"
9
10 # variables may be assigned values of other variables
11 SAME_FILE=${MY_DIR}/${MY_FILE}
12
13 # those variables can be environmental variables
14 SAME_FILE_AGAIN=${HOME}/docs/${MY_FILE}
```

Operators

Apply **operators** against numerical values to compute ner values

```
1 #!/bin/sh
2 # =, assignment
3 let "V0 = 6";      echo "Result for =6? $V0"
4 # +, addition
5 let "V1 = 1 + 2";  echo "Result for 1+2? $V1"
6 # *, multiplication
7 let "V2 = 2 * 3";  echo "Result for 2*3? $V2"
8 # -, subtraction
9 let "V3 = 5 - 4";  echo "Result for 5-4? $V3"
10 # /, division
11 let "V4 = 10 / 5"; echo "Result for 10/5? $V4"
12 # **, power
13 let "V5 = 2**10";  echo "Result for 2**10? $V5"
14 # %, modulus
15 let "V6 = 10 % 3"; echo "Result for 10%3? $V6"
```

script

```
$ ./operators.sh
Result for =6? 6
Result for 1+2? 3
Result for 2*3? 6
Result for 5-4? 1
Result for 10/5? 2
Result for 2**10? 1024
Result for 10%3? 1
```

output

if-statements

Execute code *if* the condition evaluates as true;
An essential tool when exact value of input is unknown!

```
1 #!/bin/sh
2
3 # modify the value of $FLAG as desired
4 FLAG=0
5
6 # evaluate condition contained in `[ [ ... ] ]`
7 if [[ $FLAG -eq 0 ]]
8 then
9     # if condition is true
10    echo "\$FLAG equals 0"
11 else
12    # otherwise
13    echo "\$FLAG does not equal 0"
14 fi
```

script

```
$ ./condition.sh
$FLAG equals 0
```

output

if-statement conditions

Integer comparisons

```
# is equal to
if [ "$a" -eq "$b" ]
# is not equal to
if [ "$a" -ne "$b" ]
# is greater than
if [ "$a" -gt "$b" ]
# is greater than or equal to
if [ "$a" -ge "$b" ]
# is less than
if [ "$a" -lt "$b" ]
# is less than or equal to
if [ "$a" -le "$b" ]
```

String comparisons

```
# NOT operator
if [ ! $a ]
# OR operator
if [ $a || $b ]
# AND operator
if [ $a && $b ]
```

Boolean algebra

```
# is not equal to
if [ $a != $b ]
# is equal to
if [ $a == $b ]
# is not empty
if [ -n $a ]
```

(only first line of if-statement shown, for brevity)

for-loops

Apply a block of commands **for** each element in a set;
An essential tool for repetitious tasks!

```
1 #!/bin/sh
2 for FILE in file1.txt file2.txt
3 do
4     echo "Processing \"$FILE\""
5     cp $FILE $FILE.bak
6     echo " - backup \"$FILE.bak\" created"
7     rm $FILE
8     echo " - original \"$FILE\" removed"
9 done
```

script

```
$ ./forloop.sh
Processing "file1.txt"
 - backup "file1.txt.bak" created
 - original "file1.txt" removed
Processing "file2.txt"
 - backup "file2.txt.bak" created
 - original "file2.txt" removed
```

output

for-loop styles

General for-loop structure (*for*, *do*, *done*) does not change, but there are various ways to **iterate** over set-elements

```
1 for VARIABLE in file1 file2 file3
2 do
3     command1 $VARIABLE
4     command2 $VARIABLE
5     commandN
6 done
```

expanded set

```
1 for OUTPUT in $(SOME_COMMAND)
2 do
3     command1 $OUTPUT
4     command2 $OUTPUT
5     commandN
6 done
```

set as variable

```
1 N=10
2 for i in {1..$N}
3 do
4     echo "Welcome $i times"
5 done
```

set as number range

```
1 N=10
2 for (( c=1; c<=$N; c++ ))
3 do
4     echo "Welcome $c times"
5 done
```

C-style for-loop


Arguments

shell scripts store **arguments** into the local variables \$1, \$2, ...

```
1 #!/bin/bash
2 # first user argument
3 FILE1=$1
4 # second user argument
5 FILE2=$2
6 # fixed local variable
7 DIR1=data_170727
8 DIR2=data_200203
9 # combine arguments, local variables!,
10 # and environmental variables
11 FILEPATH1=$HOME/$DIR1/$FILE1
12 FILEPATH2=$HOME/$DIR2/$FILE2
13 # execute command
14 echo "Copying"
15 echo " - src: \"$FILEPATH1\""
16 echo " - dst: \"$FILEPATH2\""
17 cp $FILEPATH1 $FILEPATH2
18 echo "...done!"
```

script

\$1 \$2



\$./example.sh file.txt file_copy.txt

```
Copying
- src: "/home/mlandis/data_170725/file.txt"
- dst: "/home/mlandis/data_200203/file_copy.txt"
done!
```

output

Commands

All text delimited by a pair of back-ticks (`) will be executed as **commands**; output from any *command substitution* can be stored into local variables

```
1 #!/bin/bash
2 # where is new directory?
3 NEW_DIR=$1
4 # store current directory
5 CWD=`pwd`
6 # change directory, and get local files
7 cd $NEW_DIR
8 FILES=`ls`
9 # loop over files
10 for FILE in $FILES
11 do
12     # sort each file
13     OUTPUT=$OUTPUT`cat $FILE | sort`"\n"
14 done
15 # print sorted files
16 echo -e $OUTPUT
17 # change to original directory
18 cd $CWD
```

script

```
$ cat tmp/a.txt
whale
alligator
bear
$ cat tmp/b.txt
banana
watermelon
apple
$ ./example.sh tmp
alligator bear whale
apple banana watermelon
```

output

First, write pseudocode

Outline your script with commented ***pseudocode*** before populating your script with working code

```
1 #!/bin/sh
2
3 # store arguments as named variables
4
5
6 # loop over all files
7
8
9
10
11     # if file passes test, do this
12
13
14     # if file fails test, do this
15
16
17
18 # report to user
19
20
```

Then, write code

Add code/commands to execute the tasks defined by pseudocode

```
1  #!/bin/sh
2
3  # store arguments as named variables
4  FILE1=$1
5  FILE2=$2
6  # loop over all files
7  for $f in $FILE1 $FILE2
8  do
9      if [[ -z $file ]]
10     then
11         # if file passes test, do this
12         OUTPUT=$file" not empty;"$OUTPUT
13     else
14         # if file fails test, do this
15         OUTPUT=$file" empty;"$OUTPUT
16     fi
17 done
18 # report to user
19 echo $OUTPUT | tr ";" "\n" | cat > output.txt
20 echo "task complete"
```

Lab 3B

github.com/WUSTL-Biol4220/home/labs/lab_03B.md