Lecture 06
shell scripts

Course: Practical Bioinformatics (BIOL 4220)
Instructor: Michael Landis
Email: michael.landis@wustl.edu

Lecture 06 outline

Last time: formats, pipelines

This time: shell scripts

shell scripts

- script anatomy
- variables
- operators
- control structures
- functions

Shell scripts

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

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

shell script, *yell.sh*

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

calling *yell.sh*

When should you use or write 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 on ***programming constructs***, such as variables, if-statements, for-loops, etc.
- operate on ***standard file formats***

Scripts vs. command line

Unix commands behave identically whether executed through the command line or a script

Like the command line, scripts are executed:

1. line-by-line
2. top-to-bottom
3. left-to-right

Complex problems often use programming constructs (*if-statements*, *for-loops*) to reduce and simplify the contents of the script

Anatomy of a shell script

Hashbang (#!) identifies which program will interpret the script by default

Text after other comments (#) are ignored

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

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

```
1  #!/bin/sh
2
3  # set arguments to local variables
4  FILE1=$1
5  FILE2=$2
6
7  # report to use that filenames were received
8  echo "received \'${FILE1}\' and \'${FILE2}\' as input"
9
10 # copy and rename file
11 cp ${FILE1} ${FILE2}"_copy.txt"
12
13 # delete original file
14 rm ${FILE1}
```

contents of *my_script.sh*

Executing a script

Scripts run much like Unix programs 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

Grant permission to execute a file as a program

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

Variables

Variables store user-defined values in memory

create $MY_DIR and $MY_FILE as ***local variables***

define new variables using values from other variables

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

$HOME is an ***environment variable*** that exists outside the script

Operators

Apply ***operators*** against values to produce new values

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

content of *operators.sh*

```
$ ./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
```

executing *operators.sh*

if-statements

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

```
 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
```

content of *condition.sh*

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

executing *condition.sh*

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" ]
```

Boolean logic

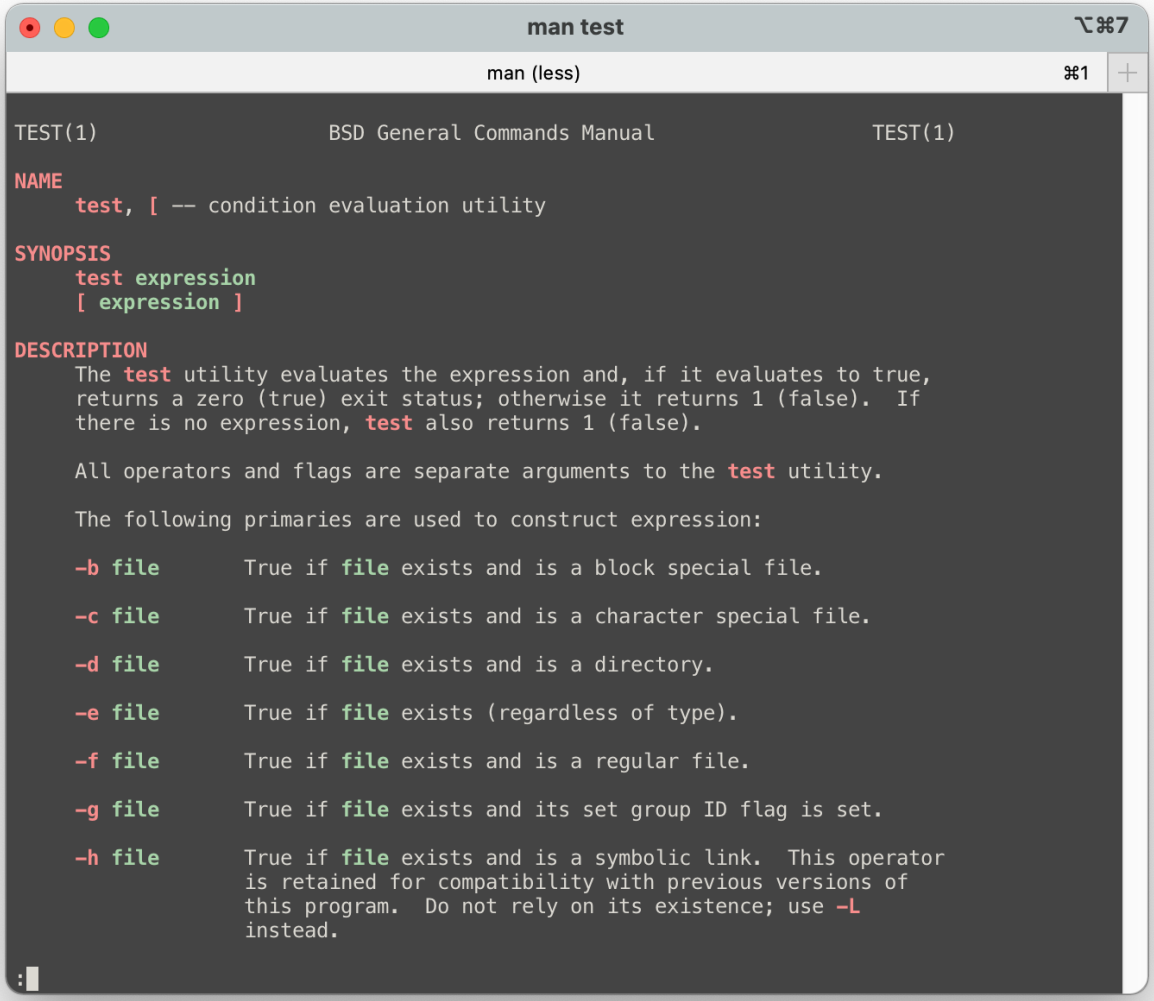
```
# NOT operator
if [ ! $a ]
# OR operator
if [ $a || $b ]
# AND operator
if [ $a && $b ]
```

string comparisons

```
# 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*)

man test for full list of conditions



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
```

contents of *forloop.sh*

```
$ ./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
```

running *forloop.sh*

for-loop styles

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

```
1 for VARIABLE in file1 file2 file3
2 do
3     command_a $VARIABLE
4     command_b $VARIABLE
5     command_c
6 done
```

list each element

```
1 for OUTPUT in $(SOME_COMMAND)
2 do
3     command_a $OUTPUT
4     command_b $OUTPUT
5     command_c
6 done
```

list of elements

```
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

Script arguments

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

```
#!/bin/bash
# first user argument
FILE1=$1
# second user argument
FILE2=$2
# fixed local variable
DIR1=data_170727
DIR2=data_200203
# combine arguments, local variables,
# and environmental variables
FILEPATH1=$HOME/$DIR1/$FILE1
FILEPATH2=$HOME/$DIR2/$FILE2
# execute command
echo "Copying"
echo " - src:\"$FILEPATH1\""
echo " - dst: \"$FILEPATH2\""
cp $FILEPATH1 $FILEPATH2
echo "...done!"
```

contents of *example.sh*

running *example.sh*

Command substitutions

surround a command with back-ticks (e.g. `ls`) to create a ***command substitution***; the output can be stored into variables

```
#!/bin/bash
# where is new directory?
NEW_DIR=$1
# store current directory
CWD=`pwd`
# change directory, and get local files
cd $NEW_DIR
FILES=`ls`
# loop over files
for FILE in $FILES
do
    # sort each file
    OUTPUT=$OUTPUT`cat $FILE | sort`"\n"
done
# print sorted files
echo -e $OUTPUT
# change to original directory
cd $CWD
```

content of *example.sh*

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

running *example.sh*

Whitespace

shell uses whitespace to distinguish between commands, options, and arguments

```
1 #!/bin/bash 2
2 # valid assignment (no spaces)
3 VAR="my_file.txt"
4
5 # invalid assignment (extra spaces);
6 # shell will attempt to execute the
7 # program `VAR`
8 VAR = "my_file.txt"
```

variable assignment must not contain spaces

```
 1 # valid if-statement (spaces)
 2 if [ $VAR == "my_file.txt" ]
 3 then
 4     echo "match!"
 5 fi
 6
 7 # invalid if-statement (no spaces)
 8 # the syntax `test` using `[]` brackets
 9 # is `[ EXPRESSION ]` not `[EXPRESSION]`;
10 # shell will not recognize the `[$VAR` command
11 if [$VAR == "my_file.txt"]
12 then
13     echo "match!"
14 fi
```

if-statement brackets must be separated from the condition by spaces

First, write pseudocode

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

```
#!/bin/sh

# store arguments as named variables


# loop over all files




        # if file passes test, do this

        # if file fails test, do this


# report to user

```

Then, write code

add code/commands to execute tasks defined by the pseudocode

```
#!/bin/sh

# store arguments as named variables
FILE1=$1
FILE2=$2
# loop over all files
for $f in $FILE1 $FILE2
do
    if [[ -z $file ]]
    then
        # if file passes test, do this
        OUTPUT=$file" not empty;"$OUTPUT
    else
        # if file fails test, do this
        OUTPUT=$file" empty;"$OUTPUT
    fi
done
# report to user
echo $OUTPUT | tr ";" "\n" | cat > output.txt
echo "task complete"
```

Overview for Lab 06