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 (#!) gives path to default program to interpret script

Text after other comments (#) are ignored

```
#!/bin/sh

# set script arguments to local variables
FILE1=$1
FILE2=$1

# report which filenames were received
echo "received \'${FILE1}\' and \'${FILE2}\' as input"

# copy and rename file
cp ${FILE1} ${FILE2}"_copy.txt"

# delete original file
rm ${FILE1}
```

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

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

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
$ ./some_script.sh
$ sh some_script.sh
$ # call script with arguments
$ ./some_script.sh file1.txt
$ # call script with arguments and options
$ ./some_script.sh --verbose file1.txt
$ # redirect script output to file
$ ./some_script.sh file1.txt
$ # use script in pipeline
$ find dir1 -name "*.txt" | ./some_script.sh > 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 -l`` is long list format)
$ ls -l process_files.sh
-rw-rw-r-- 1 mlandis mlandis 43 Sep 14 11:01 process_files.sh
$ # change file mode to include execute permission bits
$ chmod +x process_files.sh
$ # we now see the +x permission bits are set
$ ls -l process_files.sh
-rwxrwxr-x 1 mlandis mlandis 43 Sep 14 11:01 process_files.sh
$ # execute script without issue
$ ./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

```
#!/bin/sh
# define your own variables
MY_DIR="/home/mlandis/docs"
MY_FILE="my_file.txt"

# access the value of a variable using `$``
# enclose variable with `{}` to ensure
# the variable name is properly delmited
echo "Value of MY_FILE is ${MY_FILE}"

# variables may be assigned values of other variables
SAME_FILE=${MY_DIR}/${MY_FILE}

# those ariables can be environmental variables
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

```
#!/bin/bash
# addition
V1=$((1 + 2)); echo "Results for 1 + 2? ${V1}"
# multiplication
V2=$((2 * 3)); echo "Results for 2 * 3? ${V2}"
# subtraction
V3=$((5 - 2)); echo "Results for 5 - 2? ${V3}"
# division
V4=$((10 / 3)); echo "Results for 10 / 3? ${V4}"
# modulus (remainder)
V5=$((10 % 3)); echo "Results for 10 % 3? ${V5}"
# exponentiation
V6=$((2**10)); echo "Results for 2**10? ${V6}"
```

content of *operators.sh*

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

executing *operators.sh*

if-statements

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

```
#!/bin/bash
# modify `if` VALUE as desired
VALUE=-1

# evaluate condition defined in `[ ... ]`
if [ ${VALUE} -lt 0 ]
then
    # if condition 1 is met
    echo "\$VALUE is negative"
    elif [ ${VALUE} -gt 0 ]
then
    # else-if condition 2 is met
    echo "\$VALUE is positive"
else
    # else no neither condition 1/2 is met
    echo "\$VALUE is zero"
fi
# terminate if/elif/else/fi block
```

content of *condition.sh*

```
$ ./condition.sh
$VALUE is negative
```

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

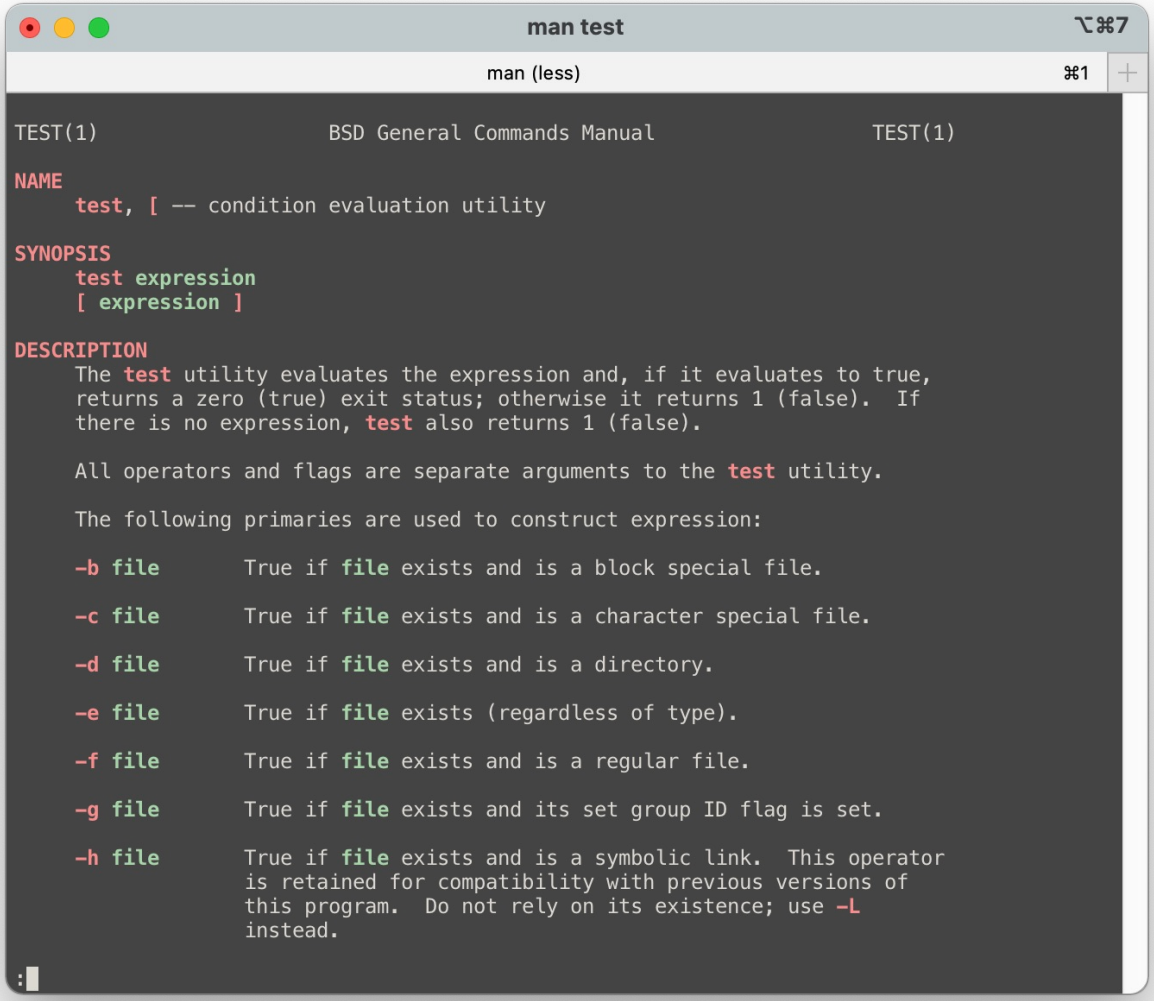
```
# NOT operator (not true)
if [ ! ${a} ]
# OR operator (either true)
if [ ${a} || ${b} ]
# AND operator (both true)
if [ ${a} && ${b} ]
```

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!

```
#!/bin/sh
for FILE in file1.txt file2.txt
do
  echo "Processing \"${FILE}\""
  cp ${FILE} ${FILE}.bak
  echo " - backup \"${FILE}.bak\" created"
  rm ${FILE}
  echo " - original \"$FILE\" removed"
done
```

contents of *forloop.sh*

```
$ touch file1.txt file2.txt
$ ./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

```
for i in file1 file2 file3
do
  command_a ${i}
  command_b ${i}
  command_c
done
```

list each element

```
for i in $(ls)
do
  command_a ${i}
  command_b ${i}
  command_c
done
```

list of elements

```
N=10
for i in {1..${N}}
do
  echo "Welcome ${i} times"
  command_a ${i}
  command_b ${i}
done
```

set as number range

```
N=10
for (( i=1; <=${N}; i++ ))
do
  echo "Welcome ${i} times"
  command_a ${i}
  command_b ${i}
done
```

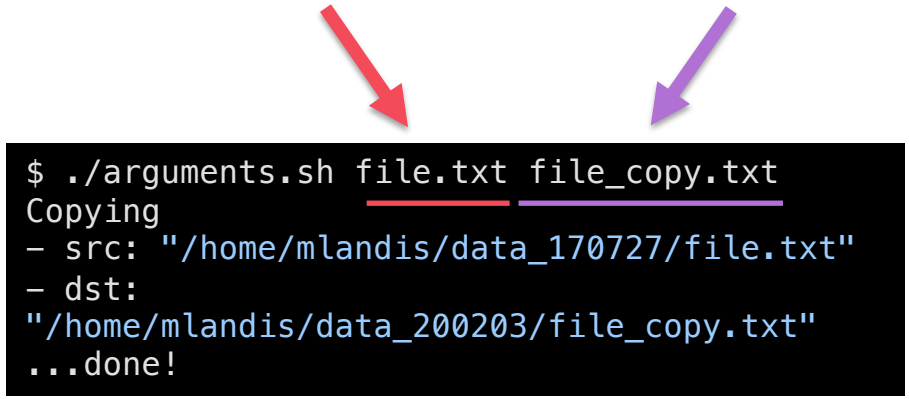
C-style for-loop

Script arguments

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

```
#!/bin/bash
#first user argument
FILE1=$1
# second user argument
FILE2=$2
# fixed local variables
DIR1=data_170727
DIR2=data_200203
# combine arguments, local variables
# and environmental variables
FILEPATH1=${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*

Script options

the *getopts* will parse options from command string;
while-loop and case-statements to handle options

```
#!/bin/bash
VAR=$1
while getopts ":hz" opt
do
    case ${opt} in
    h ) # process option h
        echo "Options:"
        echo " -h prints this help message"
        echo " -z snores (OK, rude)"
        exit 0;;
    z ) # process option z
        echo "Zzzzzz......."
        exit 0;;
    \? ) # unknown argument
        echo "Usage: ./print_input.sh [-h] [-z]"
        exit 0;;
    esac
done
# do main task if script has not called exit
echo ${VAR}
```

```
$ ./options.sh "Hello, world!"
Hello, world!
$ ./options.sh -h
Help:
  -h prints this help message
  -z snores (OK, rude)
$ ./options.sh -z
Zzzzzz.......
$ ./options.sh -X
Usage: ./print_input.sh [-h] [-z]
```

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 the new directory?
NEW_DIR=$1
# store current directory
CWD=$(pwd)
# change current directory
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}
# return to original directory
cd ${CWD}
```

content of *example.sh*

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

running *example.sh*

Whitespace

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

```
#!/bin/bash
# valid assignment (no spaces)
VAR="my_file.txt"

# invalid assignment (extra spaces);
# shell will attempt to execute the
# program `var`
VAR = "my_file.txt"
```

variable assignment must not contain spaces

```
#!/bin/bash
# valid if-statement (spaces)
if [ ${VAR} == "my_file.txt" ]
then
    echo "match!"
fi

# invalid if-statement (no spaces)
# the syntax for `test` using `[]` brackets
# is `[ EXPRESSION ]` not `[EXPRESSION]`;
# shell will not recognize `[${VAR}` command
if [${VAR} == "my_file.txt"]
then
    echo "match!"
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/bash

# 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/bash

# store arguments as named variables
FILE1=$1
FILE2=$2
# loop over all files
for file 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