

# Unix scripting

Jon Chernus (adapted from Ryan Minster)

Department of Human Genetics  
School of Public Health  
University of Pittsburgh

Document created: September 29, 2024

This slide set is called `unix_scripting.pdf` and is located in the “24\_unix\_scripting\_variables” folder of our Lectures repository.

# Objectives

- Using control structures in Unix scripting
- Using variables in Unix
- Command substitution and grouping commands
- `find` and `xargs`

# Grouping commands

Suppose you have a *list* of commands you want to execute as a unit. There are two ways to do this.

With a *subshell* (where if a command in the list creates a variable, it exists only in the subshell environment):

```
( list )
```

Or with curly braces and a semicolon (the spaces and the final semicolon are absolutely necessary):

```
{ list; }
```

# Grouping example

Suppose you want to look at the last 2 lines of a file, but you also want to keep the header line.

The “old way” - manually combine `head` and `tail` with redirection:

```
head -n1 data/subjects.txt > data/last2.txt  
tail -n2 data/subjects.txt >> data/last2.txt  
  
cat data/last2.txt | column -t  
fid      iid      sex  pheno  
FAM99    IND99    2    2.48  
FAM100   IND100   2    2.12
```

Or use grouping - which doesn't require making any files:

```
{ head -n1 data/subjects.txt; tail -n2 data/subjects.txt; } | column -t  
fid      iid      sex  pheno  
FAM99    IND99    2    2.48  
FAM100   IND100   2    2.12
```

Check out the “Output Grouping” section here:

<https://www.linux.com/topic/desktop/all-about-curly-braces-bash/>.

# Variables (and arithmetic)

- Assign a variable with =
- Reference a variable with \$
- Command-line arguments in a script are named \$1, \$2, \$3, and so on
- There are also environmental variables (e.g., \$? is the exit code of the last-run command)

You can also evaluate numerical expressions inside of `$(( ))`

# Variables and arithmetic example

```
# Set variable
number=10

# Print it out (wrong)
echo number
number

# Print it out (right)
echo $number
10

# Make a new copy (wrong)
number_copy=number

# Oops! Forgot the $ above!
echo $number_copy
number

# Correctly copy it
number_copy=$number; echo $number_copy
10

# Do arithmetic
x=$(( $number + 5*$number_copy + 3))
echo $x
63
```

# Command substitution

Command substitution replaces a command with its output, which helps build up more powerful commands from simpler ones.

The syntax is

```
$(command)
```

or

```
`command`
```



# Command substitution examples

```
cd ./data
```

```
# Prints 1 2 3 4 5
```

```
seq 1 5
```

```
1
```

```
2
```

```
3
```

```
4
```

```
5
```

```
# Print out exactly what follows echo
```

```
echo seq 1 5
```

```
seq 1 5
```

```
# Command substitution! The command is replaced by its output
```

```
echo `seq 1 5`
```

```
seq 1 5
```

```
1 2 3 4 5
```

```
# Use command substitution to make 5 folders
```

```
mkdir `seq 1 5`
```

```
seq 1 5
```

```
# Use command substitution to delete them
```

```
rmdir `seq 1 5`
```

```
seq 1 5
```

# for loops

```
# You can list the values to loop over
for i in 0 1 2 hi
do
    echo $i
done
```

```
0
1
2
hi
```

```
# You can use brace expansion
for i in {0..2}
do
    echo $i
done
```

```
0
1
2
```

## Example: command substitution and looping

```
# Loop overall our scripts and get their line counts
cd ./scripts
for file in `ls .`
do
    n_lines_temp=$(cat $file | wc -l)
    echo The file $file has $n_lines_temp lines.
done
```

The file chromosome.sh has 29 lines.

The file double.sh has 8 lines.

The file factorial.sh has 9 lines.

The file generate\_plink\_filters.sh has 13 lines.

The file same\_strings.sh has 8 lines.

The file search\_hamlet.sh has 7 lines.

# if-then-else conditionals

- Logical expressions (see next slide) need to be put inside of brackets like this: [ \$x -le \$y ] (the spaces are *required*)
- Or you can leave out the brackets and just put test in front of the expression (e.g., test \$x -le \$y)

```
# Print script
cat scripts/same_strings.sh
#!/bin/sh
a=$1 b=$2
if [ $a = $b ]
then
    echo "the strings are the same"
else
    echo "the strings are not the sames"
fi

# Run it once
bash scripts/same_strings.sh water H2O
"the strings are not the sames"

# Run it again
bash scripts/same_strings.sh dihydrogenmonoxide dihydrogenmonoxide
"the strings are the same"
```

# Conditional statements

| Common operators/expression                 | Meaning                                             |
|---------------------------------------------|-----------------------------------------------------|
| <code>\$EXPRESSION1 -a \$EXPRESSION2</code> | EXPRESSION1 and EXPRESSION2 are both true           |
| <code>\$EXPRESSION1 -o \$EXPRESSION2</code> | At least one of EXPRESSION1 and EXPRESSION2 is true |
| <code>! \$EXPRESSION</code>                 | EXPRESSION is false                                 |
| <code>-n \$STRING</code>                    | Length of STRING > 0                                |
| <code>-z \$STRING</code>                    | Length of STRING = 0                                |
| <code>\$STRING1 = \$STRING2</code>          | STRING1 and STRING2 are the same                    |
| <code>\$STRING1 != \$STRING2</code>         | STRING1 and STRING2 are not the same                |
| <code>\$INTEGER1 -eq \$INTEGER2</code>      | INTEGER1 equals INTEGER2                            |
| <code>\$INTEGER1 -gt \$INTEGER2</code>      | INTEGER1 is greater than INTEGER2                   |
| <code>\$INTEGER1 -lt \$INTEGER2</code>      | INTEGER1 is less than INTEGER2                      |
| <code>-d \$FILE</code>                      | FILE exists and is a directory                      |
| <code>-e \$FILE</code>                      | FILE exists                                         |
| <code>-s \$FILE</code>                      | FILE exists and is not empty                        |

# if-then-else example

Exit codes can also be used as conditions in `if` statements

- 0 means success/TRUE
- anything else means fail/FALSE

```
# grep has exit status 0 if there is a match, 1 otherwise
cat search_hamlet.sh
#!/bin/bash
word=$1
if grep $word ../data/hamlet.txt > /dev/null
then
    echo $word occurs in Hamlet
else
    echo $word does not occur in Hamlet
fi

# Run it twice
bash search_hamlet.sh nunnery
nunnery occurs in Hamlet

bash search_hamlet.sh bitcoin
bitcoin does not occur in Hamlet
```

# while and until loops

Basic structure of a `while` loop, where the commands are executed as long as condition evaluates to true.

```
while [ condition ]  
do  
    command1  
    command2  
    command3  
done
```

Basic structure of an `until` loop, where the commands are executed as long as condition evaluates to false.

```
until [ condition ]  
do  
    command1  
    command2  
    command3  
done
```

See the examples `factorial.sh` and `double.sh` in `./scripts/`.

# Chaining commands with && and ||

You can run multiple commands sequentially in one line with ;

```
echo Hello; echo there
Hello
there
```

But sometimes you only want to run the second command if the first is (un)successful:

```
# Only run the second if the first is successful
echo "hello" | grep "he" && echo "There was a match, so this echo command got executed!"
hello
There was a match, so this echo command got executed!
```

```
# Only run the second if the first is unsuccessful
echo "hello" | grep "zzz" || echo "There wasn't match, so this echo command got executed!"
There wasn't match, so this echo command got executed!
```

For a discussion, see the section “Exit Status: How to Programmatically Tell Whether Your Command Worked” in Buffalo’s *Bioinformatics Data Skills*.



# case-esac conditionals

```
cat scripts/chromosome.sh
```

```
#!/bin/bash
# Input: a valid chromosome
# Output: a short description
chr=$1
case $chr in
1|2|3) echo "Chromosome $chr is a large, metacentric autosome."
;;
4|5) echo "Chromosome $chr is large, submetacentric autosome."
;;
6|7|8|9|10|11|12) echo "Chromosome $chr is a medium-sized, submetacentric autosome."
;;
13|14|15) echo "Chromosome $chr is a medium-sized, acrocentric autosome."
;;
16|17|18) echo "Chromosome $chr is a moderately short, (sub)metacentric autosome."
;;
19|20) echo "Chromosome $chr is a short, metacentric autosome."
;;
21|22) echo "Chromosome $chr is a short, acrocentric autosome."
;;
23|"X") echo "Chromosome $chr is a medium-sized, submetacentric sex chromosome."
;;
24|"Y") echo "Chromosome $chr is a short, acrocentric sex chromosome."
;;
25|"XPAR") echo "Chromosome $chr refers to the pseudoautosomal region of the X chromosome."
;;
26|"MT") echo "Chromosome $chr refers to the circular mitochondrial chromosome."
;;
*) echo "$chr is not a valid chromosome."
;;
esac
```

## case-esac conditionals (con't.)

```
scripts/chromosome.sh 2  
Chromosome 2 is a large, metacentric autosome.
```

```
scripts/chromosome.sh 21  
Chromosome 21 is a short, acrocentric autosome.
```

```
scripts/chromosome.sh X  
Chromosome X is a medium-sized, submetacentric sex chromosome.
```

```
scripts/chromosome.sh 26  
Chromosome 26 refers to the circular mitochondrial chromosome.
```

```
scripts/chromosome.sh Steve  
Steve is not a valid chromosome.
```

# find

- `find` searches flexibly and recursively for files
- General syntax: `find path expression`, where `expression` can be quite complex
- Use `-maxdepth n` (where `n` is an integer  $> 1$ ) to say how deep to search (1=current directory, 2=subdirectories 1 level down, and so on)

| Expression                                | Meaning                                                                                                                                                               |
|-------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>-name &lt;pattern&gt;</code>        | Match a filename to a pattern (includes bash wildcards)                                                                                                               |
| <code>-iname &lt;pattern&gt;</code>       | Same as above, but case-insensitive                                                                                                                                   |
| <code>-empty</code>                       | Matches empty files/directories                                                                                                                                       |
| <code>-type &lt;x&gt;</code>              | Matches type <code>x</code> ( <code>f</code> for files, <code>d</code> for directories, <code>l</code> for links)                                                     |
| <code>-size &lt;size&gt;</code>           | Match files at this size (shortcuts: <code>k</code> , <code>M</code> , <code>G</code> , <code>T</code> ). Prepend <code>+/-</code> for files at least/most this size. |
| <code>-regex</code>                       | Match by regex (for extended regex add <code>-E</code> )                                                                                                              |
| <code>-iregex</code>                      | Same as above, but case-insensitive                                                                                                                                   |
| <code>-print0</code>                      | Separate results with null byte (not newline)                                                                                                                         |
| <code>expr1 -and expr2</code>             | Logical "and"                                                                                                                                                         |
| <code>expr1 -or expr2</code>              | Logical "or"                                                                                                                                                          |
| <code>-not expr</code> or <code>!"</code> | Negation                                                                                                                                                              |
| <code>expr</code><br>(expressions)        | Group a set of expressions                                                                                                                                            |

(This table is based on Table 12-3 from Buffalo's *Bioinformatics Data Skills*.)

# find examples

```
# Find a file containing 'needle' in a complicated file tree
find data/haystack/ -name "*needle*"

# Try ignoring case
find data/haystack/ -iname "*needle*"
data/haystack//haystack6/haystack9/haystack10/haystack4/not_a_Needle.txt
data/haystack//haystack8/haystack4/haystack2/haystack1/neEdLe.txt

# Try to exclude the unwanted match
find data/haystack/ -iname "*needle*" -not -iname "*not*"
data/haystack//haystack8/haystack4/haystack2/haystack1/neEdLe.txt

# Where is the 'stuff' folder?
find data/haystack -iname stuff -type d
data/haystack/haystack1/haystack3/stuff
```

## find -exec to run commands on search results

`find path expression -exec command {} \;` will

- Find all files in the `path` that match the `expression`
- Run `command` on each of them

Notes:

- `{}` is a placeholder referring to a result file (you can use it more than once if you need to)
- `\;` is a delimiter that terminates the `exec` commands
- `exec` can only run fairly simple commands “by itself”, so more complex commands need to be wrapped in a “child shell” with `bash -c` (see <https://unix.stackexchange.com/questions/389705/understanding-the-exec-option-of-find>)

# find -exec example

Suppose we named some important files and want to make backup copies in another folder

- Put `bash -c` to run a child shell and then the command in single quotes
- After `bash -c` comes a placeholder argument and then `{}`, which will be `$1` inside the child shell

```
find data/haystack -type f -iname "*important*"
data/haystack/haystack1/haystack3/stuff/IMPORTANT_data.csv
data/haystack/haystack1/haystack1/haystack1/more_important_stuff.txt

find data/haystack -type f -iname "*important*" \
-exec bash -c 'name=$(basename $1); cp $1 data/important_files/backup_$name' placeholdername {} \;

ls -l data/important_files/
total 0
-rwxr-xr-x@ 1 jonathanchernus  staff  0 Sep 29 14:22 backup_IMPORTANT_data.csv
-rwxr-xr-x@ 1 jonathanchernus  staff  0 Sep 29 14:22 backup_more_important_stuff.txt
```

This is tricky - again, see

<https://unix.stackexchange.com/questions/389705/understanding-the-exec-option-of-find> for details.

- `xargs` takes standard in and supplies it as arguments to other commands
- `find -exec command` and `find | xargs command` are similar (`xargs` has some advantages)
- `find -exec` doesn't let you check the found files before running the command (with `xargs` you can)
- By default, `xargs` passes all arguments at once, so add `-n 1` if you need to pass them one at a time
- You can also parallelize with `xargs` by adding `-P m` where `m` is the number of parallel processes to run at a time
- Note that as with `exec` you may need to build small scripts to supply to `xargs` to perform more complex tasks
- You can also use `{}` with `xargs` like with `exec`, but you need to add `-I {}` first

For example, these two commands both find and delete any `.fastq` files:

```
find . -name "*.fastq" -exec rm {} \;  
find . -name "*.fastq" | xargs rm
```

# find and xargs example

Suppose you want to find all .fastq files containing temp in their names and delete them *after* peeking at the list of files to delete.

```
# Make some files to find and delete
touch data/fastq/temp_{1,2}{A,B}.fastq

# Find the files and save their paths
find data -name "temp*.fastq" -type f > data/files_to_delete.txt

# Look at the file names
cat data/files_to_delete.txt
data/fastq/temp_1A.fastq
data/fastq/temp_1B.fastq
data/fastq/temp_2A.fastq
data/fastq/temp_2B.fastq

# Now delete them with xargs
cat data/files_to_delete.txt | xargs rm

# Confirm they're gone
find data -name "temp*.fastq" -type f
```



# Exercise

Write a script that

- takes a minor allele frequency threshold as its argument
- finds all of the PLINK binary data sets in data
- writes out a text file with a PLINK command for each data set that
  - applies the given MAF threshold to the file
  - writes out a binary format data set with `_filtered` appended to the name

# Suggested solution

```
# View solution script
cat scripts/generate_plink_filters.sh
#!/bin/bash
# Input: a number between 0 and 1 to use as MAF filter
maf=$1

# Find any bed files, print a PLINK command for each
find data -type f -name "*.bed" |\
xargs -I{} \
bash -c 'name=$(basename $1) ;\
newname=$(echo $name | sed s/.bed//)_filtered ;\
echo plink --bfile $1 --maf $2 --make-bed --out $newname' \
placeholder {} $maf \
> plink_commands.txt

# Run it and look at the output
bash scripts/generate_plink_filters.sh 0.02

cat plink_commands.txt
plink --bfile data/a.bed --maf 0.02 --make-bed --out a_filtered
plink --bfile data/c.bed --maf 0.02 --make-bed --out c_filtered
plink --bfile data/haystack/haystack7/haystack4/f.bed --maf 0.02 --make-bed --out f_filtered
plink --bfile data/b.bed --maf 0.02 --make-bed --out b_filtered
plink --bfile data/e.bed --maf 0.02 --make-bed --out e_filtered
plink --bfile data/d.bed --maf 0.02 --make-bed --out d_filtered
```