Lecture 22 - Programming and Scripting (Part 1)

DSE 511

Drew Schmidt
2022-11-10

Announcements

- Nothing unresolved from last time
- Schedule:
  - Nov 10 and 15 - shell
  - Nov 17 and 22 - databases
  - Nov 24 - No class for US Thanksgiving
  - Nov 29 and Dec 1 - more databases
  - Dec 6 - course wrapup
- New homework (last one)
  - Coming "soon" (late next week?)
  - Due Mon Dec 5? (fairly hard last date)
  - No homework on last modeule (databases)
- Questions?

Content

- Background
- Variables
- Exit Codes
- Logic

Background

Scripting

- Commands we've used interactively can be put into a *script*.
- Just like R, or Python, or Matlab, or ...
- There are a few differences here.

Why Script?

- Reproducibility
- Automation
- ...

Content of a Shell Script

- Comment with `#`
- First Line
  - The top line should indicate which shell runs the file
  - It should look like `#!/bin/sh` or `#!/bin/bash`
  - This is sometimes called a "shebang" or "hashbang" (hash symbol + exclamation mark)
  - Looks like a comment *but it isn't*!
- Body
  - Can be anything you run interactively
  - Variables, logic, commands, ...

Permissions

- The file should be executable e.g. `chmod +x my_script.sh`
- How to execute:
  - Relative path: `./my_script.sh`
  - Absolute path: `/full/path/to/my_script.sh`
- Can also do `bash my_script.sh` which *ignores the shebang*!

Example

```
cat hw.sh
```

```
#!/bin/bash

# this is a comment
echo "hello world"
```

```
./hw.sh
```

```
bash: ./hw.sh: Permission denied
```

```
chmod +x hw.sh
```

```
./hw.sh
```

```
hello world
```

```
pwd
```

```
/tmp
```

```
/tmp/hw.sh
```

```
hello world
```

```
./tmp/hw.sh
```

```
bash: ./tmp/hw.sh: No such file or directory
```

Scripting

- You know how to do some things at the command line
- You know how to run a shell script
- We just need to learn the programming syntax for shell

Variables

Variables

- A named symbol containing some programmatic information
- Just like variables in R, Python, ...
- Because *the shell is your environment*, these are sometimes called environment variables

Assignment and Expansion

Assignment

- Use `=`
- Naming is very conservative
  - Start with letter
  - letters + numbers + underscores
- *Don't* include spaces around the `=`
- For strings, use quotes if there are spaces
- For commands, use backticks

Expansion

- Retrieve the value with `$`
- Put squiggly brackets around the var name for safety (optional)
- Arithmetic is a little tricky...
- Single and double quotes *do different things*
- Undefined variables expand to `""`

Example: Basics

```
x=1
echo $x
```

```
1
```

```
y=2
echo $y
```

```
2
```

```
echo $x $y
```

```
1 2
```

```
echo $x $y $z
```

```
1 2
```

Example: Brackets

```
x=1
xx=2
echo $xx
```

```
2
```

```
echo $x
```

```
1
```

```
echo ${x}x
```

```
1x
```

Example: Quotes

```
x=1
echo $x
```

```
1
```

```
echo "$x"
```

```
1
```

```
echo '$x'
```

```
$x
```

```
echo '$x='$x
```

```
$x=1
```

Example: Commands

```
ARCH=`uname -m`
echo "My computer's architecture is $ARCH"
```

```
My computer's architecture is x86_64
```

```
NUM_DIRS=`ls -d */ | wc -l`
WD=`pwd`
echo "My $WD has $NUM_DIRS directories"
```

```
My /tmp has 33 directories
```

Example: Arithmetic

```
x=1
y=2
echo $(( $x + $y ))
```

```
3
```

```
echo $(( $x / $y ))
```

```
0
```

```
echo "scale=1; ($x / $y)" | bc
```

```
.5
```

Example: Environment Variables

R

```
x=1 Rscript -e "print(Sys.getenv('x'))"
```

```
[1] "1"
```

```
y=1 Rscript -e "print($y)"
```

```
[1] 1
```

```
z=3 Rscript -e 'print($z)'
```

```
Error: unexpected '$' in "print($"
Execution halted
```

Python

```
x=1 python -c "import os; print(os.enviro
```

```
1
```

```
y=1 python -c "print($y)"
```

```
export y=1
python -c "print($y)"
```

```
1
```

Scoping

- By default, variables are local to the executing shell
- If you start a new shell, the variable will be gone

```
myvar=1
echo $myvar
```

```
1
```

```
bash
echo $myvar
```

```
exit
export myvar=$myvar
bash
echo $myvar
```

```
1
```

Exit Codes

Exit Codes

- Every command has an *exit code*
- Exit code:
  - An integer between 0 and 255 (8-bit unsigned)
  - 0 means ok
  - Non-zero means not ok
- Exit code always stored in `$?`
- You can utilize this in your scripts
  - `exit` defaults to 0
  - `exit 1` quits with error code 1

Example

```
bash
exit 255
echo $?
```

```
255
```

```
bash
exit 911
echo $?
```

```
143
```

```
echo "911 % 256" | bc
```

```
143
```

Example

```
uname -m
```

```
x86_64
```

```
echo $?
```

```
0
```

```
command-definitely-does-not-exist
```

```
command-definitely-does-not-exist: command not found
```

```
echo $?
```

```
127
```

Example

```
Rscript -e "1+1"
```

```
[1] 2
```

```
echo $?
```

```
0
```

```
Rscript -e "stop()"
```

```
Error:
Execution halted
```

```
echo $?
```

```
1
```

Logic

Logic Syntax

```
if [ condition ]; then

elif [ condition ]; then

else
  alternate
fi
```

- There is also `test`
- It's mostly like bracket syntax

Some Logical Operators

Strings

- `==` - Same
- `!=` - Different
- `<` and `>` - lexicographic ordering

Files

- `-e` - File exists?
- `-d` - Exists and is a dir?
- `-f` - Exists and is a regular file?
- `-r` - Exists and is readable?
- And many, many more…

Integers

- `-eq` - Equal
- `-ne` - Not equal
- `-lt` - Less than
- `-le` - Less than or equal to
- `-gt` - Greater than
- `-ge` - Greater than or equal to

See also `man test`

Example

```
ARCH=`uname -m`
if [ "X$ARCH" == "X" ]; then
  echo "Unable to determine arch!"
  exit 1
elif [ "X$ARCH" == "Xx86_64" ]; then
  echo "Your computer is x86"
elif [ "X$ARCH" == "Xaarch64" ]; then
  echo "Your computer is ARM"
else
  echo "Unknown arch!"
fi
```

```
## Your computer is x86
```

Example

```
LN2=`echo "l(2)" | bc -l`
if [ "$LN2" = "1" ]; then
  echo "ln(2) > 1"
else
  echo "ln(2) <= 1"
fi
```

```
## ln(2) <= 1
```

A Warning

- When testing a variable, make sure it's not empty!
- Common pattern `if [ "X$MYVAR" == "X" ]`
- In bash `if [ -z "$MYVAR" ]` does the same
- Each of these checks to see if the string is empty!

Example: Empty Strings

```
x=1
echo "#$x-$MYVAR#"
```

```
#1-#
```

```
test -z "$x"
echo $?
```

```
1
```

```
test -z "$MYVAR"
echo $?
```

```
0
```

```
test ! -z "$MYVAR"
echo $?
```

```
1
```

Wrapup

Wrapup

- Scripting in the shell is just like scripting in R/Python/etc
- Well…mostly…
- Check your exit codes!
- Next time: more shell scripting

Questions?