Intro to Bash Scripting

CS14 - S26

Notes on Notation, Topic Coverage

- Code and commands will be displayed in `monospace blue font` or `white font on dark backgrounds.`
- Places where you substitute text will be displayed in `<angle brackets>`

We will cover basic syntax that is useful for operating on files and directories, calling commands and programs within a script, and constructing pipelines.

We *will not* be covering arithmetic operations, functions, or other more advanced features.

Examples and Exercises

Copy `/home/ckazer/public/cs14/inclass/bash_scripts` to somewhere in your home directory. It contains example code and exercise starter files for today.

Making a bash script

Bash script files conventionally end with the `.sh` extension.

To make sure a script is interpreted as bash code, you should add a "shebang" directive to the top, typically:

```
#!/bin/bash
```

Bash scripts need to be given the executable permission (with `chmod`) in order to run them using `./`

* You can also make python programs runnable with `./` by adding a shebang `#!/bin/python3` and giving the program the executable permission!

Making a bash script

Bash code can also be written directly to the shell.

Variables and settings will be retained as long as the current session is open.

Variables

- Variables are created using `=`.
- All variables are strings.
- Use `$` to access variable contents. Without a `$`, variable names will be treated as normal strings.

```
#!/bin/bash

# fruit.sh

my_fruit="banana"

echo "My favorite fruit is the $my_fruit"

echo "I like ${my_fruit}s"
```

Variable Peculiarities

- Variable declarations cannot have spaces before or after `=`.
- Variables retain meaning inside of double quotes.

Word Splitting, Double Quotes, and Curly Braces

By default, variables undergo "word splitting" (and globbing) if applicable.

If a variable's contents have whitespace, then it will be split into tokens that are operated on separately. This has odd effects for many commands.

To avoid word splitting, put variables in double quotes. E.g.,

`"$my_var"` instead of `$my_var`

Word Splitting Example

Suppose you had a filename with a space character in it.

```
echo "one two" > "two words.txt"
```

```
filename="two words.txt"
```

Now try the following two commands.

```
cat $filename
```

```
cat "$filename"
```

Word Splitting, Double Quotes, and Curly Braces

Curly braces `{}` can be used to separate variable names from other text, so that variable substitution is interpreted correctly. E.g.,

```
echo "I like $my_fruits"
```

vs

```
echo "I like ${my_fruit}s"
```

Command Line Arguments

Any space-separated tokens* that come after a bash script on the command line are interpreted as command line arguments.

These can be accessed in the script using a `$` followed by the position of the argument.

Similar to other languages, `$0` is the name of the script, `$1` is the first argument to the script, and so on.

* If you put single or double quotes around tokens separated with a space, it will be stored in a single command line argument. This is one of the many edge cases that makes bash scripting error prone.

Command Line Arguments

`"$#"` contains the number of command line arguments, not counting the script name.

`"$@"` expands to all of the arguments.

```
#!/bin/bash

# cmdargs.sh

echo "The first arg is $1."

echo "You entered $@."

echo "$# arguments total."
```

Exercise 1

Write a bash script named "summary.sh" that takes the name of a text file as a command line argument.

This program should print the first line of the file, and then print how many lines it contains.

```
$ ./summary.sh ~/cs14_materials/labs/bash_scripts/bash_scripts.adoc
Summary of /home/ckazer/public/cs14/labs/bash_scripts/bash_scripts.adoc
First line:
= Bash Scripts

/home/ckazer/public/cs14/labs/bash_scripts/bash_scripts.adoc contains the following
number of lines:
211 /home/ckazer/public/cs14/labs/bash_scripts/bash_scripts.adoc
```

Exit Codes

Every operation has an exit code that indicates the outcome of the operation.

Normally, exit code 0 indicates success.

The exit code of the most recent operation can be accessed in `$?`

```
$ cat ~/.vimrc
...
$ echo "$?"
0
$ cat fake_file
cat: fake_file: No such file or directory
$ echo "$?"
1
```

The `set` command

`set` can configure options for how bash behaves. Several settings are useful for scripts in particular.

`set -e` stops a script if any operation exits with a code other than 0

> Assuming all operations should have exit code 0, this can make your scripts "fail fast".

`set -u` causes an error if any variable is undefined. The default behavior is to replace undefined variables with the empty string.

`set -eu` combines both!

If/Else

```
if [ <condition> ]; then
    <body>
fi
```

If `<condition>` has return code 0, then `<body>` executes.

There must be a space after `[` and before `]`

* Three common ways of checking conditions exist: `test`, `[ ]`, and `[[ ]]`. There are small differences, but for simplicity we will use `[ ]`.

```
if [ <condition> ]; then
    <body>
elif [ <condition> ]; then
    <body>
else
    <body>
fi
```

Common Conditionals

- `[ -f <arg> ]` checks if `<arg>` is a regular file.
- `[ -d <arg> ]` checks if `<arg>` is a directory.
- `[ <arg1> == <arg2> ]` checks if `<arg1>` and `<arg2>` are equal.
  - `<arg1>` and `<arg2>` must be *strings*.
  - Also, `!=`, `<`, `>`

- Flag conditionals can be negated with `!`
  - The following would check if the argument *is not* a directory: `[ ! -d <arg> ]`
- `[ <expr1> ] && [ <expr2> ]` checks if *both* expressions have return code 0
- `[ <expr1> ] || [ <expr2> ]` checks if *either* expression has return code 0

A note about `&&`

The second expression in an `&&` statement only evaluates if the first expression returns 0. This is commonly used outside of `if` statements to combine commands where you only want to execute the second command if the first succeeds.

E.g.,

```
[ -f "$filename" ] && cat "$filename"
```

```
make my_prog && ./my_prog
```

Arithmetic Conditionals

- `[ <arg1> -eq <arg2> ]` checks if `<arg1>` and `<arg2>` are equal.
  - `<arg1>` and `<arg2>` must be interpretable as *integers*.
  - Also, `-ne`, `-lt`, `-le`, `-gt`, `-ge`

```
#!/bin/bash

# hello.sh

set -eu

if [ $# -ne 1 ]; then

    echo "usage: ./hello.sh <name>"

    exit 1

fi

echo "Hello $1!"
```

Exercise 2

Write a bash script named "check_same.sh" that takes exactly two arguments, and reports whether the two arguments are identical. If more or less than two arguments are provided, print a usage message and exit.

```
$ ./check_same.sh
usage:./check_same <string_1> <string_2>
$ ./check_same.sh apple banana
strings apple and banana do not match
$ ./check_same.sh cs14 cs14
strings match!
```

For word loops

```
for <var_name> in <words>; do
    <body>
done
```

For each space separated token in `<words>`, `<var_name>` will take on that value and then `<body>` will execute.

This is especially useful with globbing to perform operations on sets of files/directories.

```
#!/bin/bash
# my_ls.sh

set -eu
# Command line argument checking omitted for brevity

dir="$1"

for name in "$dir"/*; do
    if [ -d "$name" ]; then
        echo "$name is a directory"
    else
        echo "$name is a file"
    fi
done
```

Exercise 3

Write a bash script named "first_lines.sh" that takes a single directory as a command line argument, and for each file in that directory, prints the first line of the file.

If there is not exactly one argument that is a directory, print a usage message and exit.

While read loops

It's often useful to perform an operation on each line of an input file or each line of standard input. This can be accomplished with a `while read` loop.

```
<input op> | while read <var_name>; do

    <body>

done
```

```
#!/bin/bash
# wc_by_line.sh

set -eu
# Command line argument checking omitted for brevity

filename="$1"

cat "$filename" | while read line; do
    echo "$line" | wc -w --total=never
done
```

Exercise 4

In `bash_scripts/`, you have been provided a file called `veggies.txt`

Write a script named “fileify.sh” that makes a directory called `vegetables/`, then makes one text file for each line of `veggies.txt`, containing that line.

Command Substitution

Sometimes, you may want to use the result of a command in part of your script. Naively storing a command will save the tokens of the command, not the command's result.

```
$ first_line="head -n1 my_ls.sh"
$ echo "$first_line"
head -n1 my_ls.sh
```

Command Substitution

We can use *command substitution* to execute and store the result of a command. To perform command substitution, put `$( )` around the command.

```
$ first_line=$(head -n1 my_ls.sh)
$ echo "$first_line"
#!/bin/bash
```

```
#!/bin/bash
# better_wc_by_line.sh

set -eu
# Command line argument checking omitted for brevity

filename="$1"

cat "$filename" | while read line; do
    count=$(echo "$line" | wc -w --total=never)
    echo "$line  #($count word(s))"
done
```

Literal Strings

Conversely, single quotes can be used to make literal strings. This avoids splitting, substitution, etc..

```
$ my_fruit=apple
$ echo $my_fruit
apple
$ echo '$my_fruit'
$my_fruit
```

Further Topics

We have not covered:

- Functions
- Arrays
- Specific details of expansion and splitting
- Etc.

See the Gnu Bash Manual for more info!