Bash Scripting

OPS102 Week 8 Class 1

Chris Tyler/John Sellens

July 2, 2024

Seneca Polytechnic

Outline

Script Basics

What is a Script?

- A shell script is a simple computer **program** which is **interpreted** by an operating system shell.
- Scripts are used to automate procedures that could be manually performed from the command line.

How will scripts save hours of my life?

If you're developing a computer program which requires 25 steps to build and test, and you're going to iterate through the build and test process 100 times, you can:

- Perform 2500 steps manually – a lot of work, and error-prone; or
- Write a script containing those 25 steps and then execute it each time you want to build and test your program (100 times)
- You can even set things up so the script is executed automatically when certain conditions are met – such as saving a change to your source code!

Basic Requirements for Shell Scripts

1. Create a text file containing shell commands.
2. Tell the operating system which shell to use to execute the commands.
3. Ensure that the script file has the appropriate permissions.

1. Create a file containing shell commands

- Use any text editor
- Use the same commands that you would type at the command-line
- Save the file

2. Tell the operating system which shell to use

- Add a “shebang” line to the start of the file
    - First character is a **sh**arp: #
    - Second character is a **bang**: !
    - The rest of the line is the absolute pathname of the shell / interpreter
    - `#!/bin/bash`
- The name “shebang” comes from “sharp” and “bang”
    - Also known as a “hashbang” line
- The #! characters form a “magic number” which the kernel notices
    - And runs the command: `/bin/bash name-of-file`
- The # character causes the shell to interpret the line as a comment, and ignore it

3. Set the correct permissions on the file

- The **kernel** needs execute permission to analyze the shebang line and execute the correct shell/interpreter.
- The **shell** needs read permission to read the contents of the file.
- Set these permissions with the `chmod` command for any user(s) that should be able to execute the file:

```
chmod u+rx scriptname
```

Demo: Basic scripts

- Let’s write some simple scripts using commands that we know
- Scripts act like any other executable file, so we can type the script name as a command. However, unless the script is in a directory that is normally searched by the operating system, it won’t be able to find it. We’ll talk about how to adjust this later, but for now, we can use a pathname that includes a slash to execute a script we’ve written:

```
./scriptname
```

Bash Details

Setting Variables

- To set a variable:
  `VARIABLE=VALUE`
- Variable names start with a letter and can contain letters, numbers, and underscores.
- Case matters! `A` and `a` are different variables.
- Don’t put spaces on either side of the equal sign.
  - Unlike assignments in most other languages.

Accessing Variables

- To use a variable value in a command, precede it with a dollar sign: `$VARIABLE`
- You can use a variable anywhere in a command:
    - Arguments
    - Command name
- The value of the variable is substituted into the command.

Variables: A Simple Example

```
$ what=World
$ echo Hello $what
Hello World
```

Word Splitting / Tokenization

- The shell uses certain separator characters (whitespace by default) to split commands into words.
- For example, spaces are used to break this line into a three parts (or tokens): a command and two arguments

  ```
  $ ls -l filename
  ```
- But this means that arguments, such as filenames, that contain spaces will be interpreted as multiple arguments:

  ```
  $ ls -l file one
  ```

Preventing Word Splitting with Quoting

- Quoting prevents words from being split.
- It’s needed whenever we’re dealing with text that contains separators (such as space or tabs).
- You can also use quoting to hide other special characters from the shell, such as the file globbing characters: `* ? [ ]`

Types of Quotes: Single vs Double

- Single or double quote characters may be used to quote strings:
  ```
  B='Hello World'
  A="Hello World"
  ```
- When double quotes are used, variables **will be** expanded inside the quotes.
- When single quotes are used, variables **will not be** expanded inside the quotes.

Quoting: Examples

```
$ what="World"
$ echo "Hello $what"
Hello World
$ echo 'Hello $what'
Hello $what
```

Quoting: More Examples

```
$ what="World"
$ message="Hello $what"
$ what="There"
$ echo $message
Hello World
$ echo "$message"
Hello World
$ echo '$message'
Hello $message
```

Note that the variable substitution of `$what` happened during the assignment to `message` and not when `message` was used.

Quoting: One argument vs. Multiple

When a string contains a separator such as a space, and it is unquoted, the shell will interpret it as multiple words (tokens). When used as an argument, this will be interpreted as multiple arguments:

```
$ touch "new file"
$ ls -l new file
ls: cannot access 'new': No such file or directory
ls: cannot access 'file': No such file or directory
$ ls -l "new file"
-rw-r--r--. 1 chris chris 0 Jun 18 22:47 new file
```

Quoting: One argument vs. Multiple

You should always double quote variables that may contain a space in their value when using them as command arguments.

- There are one or two slightly obscure exceptions to this rule.

This is especially true for filenames -– you never know when a user is going to put a space (or a special character) in a filename! Many scripts work fine with opaque filenames (those containing no whitespace) but fail with non-opaque filenames.

Quoting: Backslashes (Escaping)

A backslash character outside of quotes or inside double quotes instructs the shell to ignore any special meaning that the following character may have.

```
$ touch "new file"
$ ls -l new file
-rw-r--r--. 1 chris chris 0 Jun 18 22:49 new file
```

```
$ echo "This string contains a \"quoted\" string"
This string contains a "quoted" string
```

```
$ A=Testing
$ echo "   \$A"
   $A
```

Shell Variables vs Environment Variables

- By default, a variable is local to the shell in which it is running.
    - Somewhat similar to local variables in C functions.
- You can export variables to make them **environment variables**. That means that they are passed to programs (child processes) that are executed by the shell.
- Environment variables are commonly used to pass configuration information to programs and to configure how programs operate.
- Environment variables are used by all processes, not just the shell!
- By convention, we use UPPERCASE to name environment variables.

Common Environment Variables

Environment Variable	Purpose
EDITOR	Name of the default text editor (often `/usr/bin/nano`)
PATH	A colon-separated list of directories that will be searched when looking for a command
LANG	The default language – used to select message translations as well as number, currency, date, and calendar formats.
HOME	The user’s home directory -– used for relative-to-home pathnames.

Viewing and Creating Environment Variables

- See all current environment variables and their values:

```
$ printenv | less
```

  - The `set` command shows *all* variables.

- Create an environment variable with `export`:

```
$ X=500
$ export X
$ export Y=123
```

Example: PATH Environment Variable

```
$ cat showdate
#!/usr/bin/bash
date
$ ./showdate
Sun 18 Jun 2023 11:20:18 PM EDT
$ showdate
bash: showdate: command not found
$ echo $PATH
/home/chris/bin:/usr/local/bin:/usr/bin:/bin
$ PATH="$PATH:."
$ showdate
Sun 18 Jun 2023 11:20:45 PM EDT
```

Note: Having . in your $PATH is a security risk. But is the default on matrix.

Example: LANG Environment Variable

```
$ echo $LANG
en_CA.UTF-8
$ date
Sun 18 Jun 2034 11:32:27 PM EDT
$ foobarbaz
bash: foobarbaz: command not found
$ LANG=fr_CA.UTF-8
$ date
ven nov  3 01:45:08 EDT 2023
$ foobarbaz
bash: foobarbaz: commande introuvable
```

See also the `LC_*` environment variables:

https://unix.stackexchange.com/questions/87745/what-does-lc-all-c-do

Example: HOME Environment Variable

```
$ echo $HOME
/home/chris
$ echo ~
/home/chris
$ HOME=/
$ echo ~
/
$ ls ~
afs bin boot dev etc home lib lib64 lost+found
media mnt opt proc root run sbin srv sys tmp
usr var
```

Note: changing $HOME can cause you problems.

Example: PS1 Local Shell Variable

Change the shell prompt:

```
$ PS1="Enter a command: "
Enter a command: date
Sun 18 Jun 2023 11:17:14 PM EDT
Enter a command: PS1="[\u@\h \W]\$ "
[chris@toronto ~]$ PS1="$ "
$
```

Note: PS1 looks like an environment varible because it's uppercase, but it isn't.

Summary

Summary

- Scripts are handy, and easy to create – yay text files!
- The “shebang” line is a “magic number” that guides the kernel on how to execute a script.
- Variables are handy – local and environment.
- Quoting and backslash escaping hide special characters and whitespace.