Shell Scripting

Overview

- An Introduction to Shell Scripting
- Why Shell Scripting is Used
- Arguments
- Variables and Environment Variables
- Conditionals and Loops

Learning Objectives

We will be able to:

- Describe what shell scripting is used for, and why we might choose it.
- Practice running scripts in a shell
- See how Arguments and Variables work with scripts
- Describe which conditonals and loops can be used in shell scripts
- Write our own shell script

What is shell scripting?

- Incredibly powerful and frequently used by developers.
- Often also called **Bash** scripting.
- Remember all those wonderful commands we've seen previously? (`cat`, `sed`, `|`, `>`)
- We can write some code to run the commands automatically, in sequence.

Shell Script

A computer program designed to be run by the Unix shell

What is shell scripting used for?

A lot of things!

- Configuring applications
- Deployments
- Installing packages
- Automating repetitive tasks

Generally, almost anything we could do with manual commands on the command line can be made into a script.

discuss some examples of how we could do some of these things.

```
Deployments: commands to build, upload build to server

Install packages: instructions on the server to install python and
dependencies

Repetitive task: create nightly backups, git process
```

Why Shell Scripts?

The previous tasks we discussed, we could write a program in another language to do them instead.

Why don't we just write a program in Python, or Java, or something else?

- Unix shell is available anywhere, on any Linux machine
- No dependencies required, so we know our scripts are always going to work
- Allows us to use all the powerful tools and commands that are available in Unix
- Almost anything we can do with manual commands on the command line we can do in a script

Shell scripts are often used for simple tasks where writing a full app may be overkill

```
They may also be used to do things like set up the dependencies and
environment required by a more complex app to allow that app to then run
```

Example of a Shell Script

```bash
#!/bin/bash
set -eu

timeout=120
retry_delay=1

query_with_timeout () {
    echo "Triggering with timeout"
    curl --retry $timeout --retry-connrefused --retry-delay $retry_delay
'google.com' &> /dev/null
}

print_online () {
    echo "we're online!"
}

query_with_timeout && print_online
```

What do you think is happening?

Ask the class to focus on the parts of this script they CAN understand based on their knowledge so far. Ask them to GUESS based on their observation and intuition what this script is doing.

```
Make the point that as engineers, our problem-solving approach should first
be to focus on the parts we think we CAN understand, rather than being
intimidated and overwhelmed by the parts we do not yet understand.

Note: /dev/null is a void where the output will just disappear, good for
capturing unwanted output without showing it on the screen.
```

How to do shell scripting?

We can save our script commands to a file, and run it from the ba**sh** shell.

To run a script, it must have execution permissions set.

```
chmod +x script.sh
```

A script can be run by using `./` in front of the filename in your shell terminal

```
./script.sh
```

The \`sh\` file extension indicates a ba**sh** script

```
Refer back to Unix Shell module and file permissions content that was
covered there.

chmod+x makes executable for the current user
```

Shebang

You will often see `#!/bin/bash` or `#!/usr/bin/bash` at the start of shell scripts.

This is the 'Shebang' - It tells the shell what interpreter to run the script with.

Without the shebang we would have to run our scripts explicitly with bash `bash my_script.sh`

With it we can simply run them with `./my_script.sh`

You can add `#!/usr/bin/env python3` to your python application to execute it without writing the python3 command.

Note that the path to python on your system may be different from the example above

---

Shebang Example

Notice how the shebang is always on the first line

```
#!/bin/bash
set -eu
name="John"
echo "Hi $name"
```

```
#!/usr/bin/env python3

import sys
import os
[...]
```

Parts of a Script: Options

Options that control how the shell will run the script.

Can be set by using **set** in your script. Common options you might use:

`-e`: Exit immediately if a command fails. **It's best practice to always fail a script the moment a command fails.**

`-u`: Treat unset variables and parameters as an error.

Parts of a Script: Variables

Similar to variables in Python!

Variables have a name and can be given a value with `=`

```
timeout=120
```

Reference a variable by using `$` in front of the variable name

```
$<variable name>
```

Variables inside a script should have lowercase names (to avoid accidentally overwriting global or environment variables)

When we assign a variable we must not leave any whitespace around the `=` - This is different to Python.

```
If we do then the shell will try to treat the variable name or value as a
command and run it - May demonstrate this to the class
```

Parts of a Script: Comments

Anything on a line after `#` will be commented out.

Don't forget to comment your scripts. It's good practice to write what the script does at the top after the shebang.

Try to include a help command, and explain how to invoke the script by giving an example

Comments `#` are different from the shebang `#!`

Parts of a Script: Arguments

Values can be passed into scripts from outside the script with the use of arguments.

Arguments are assigned an index number (e.g. 1, 2, 3) based on the order they appear.

Arguments can be accessed by using that index number with `$`

```
$<index of argument>
```

Arguments Example

Invoke the script:

```
./say-hi.sh "Jane Doe"
```

Assign a variable using the argument:

```
#!/bin/bash
set -eu

name=${1}
echo "Hi $name"
```

```
Hi Jane Doe
```

Note that when we use ./ to run the script then we must be in the same directory as the script - As we have seen previously `.` is an alias for the current directory.

Parts of a Script: Environment Variables

Environment variables are a different type of variable that is **inherited** inside a script from the outside **environment** in which that script runs.

We do not need to explicitly pass environment variables into a script like arguments, they are available automatically.

Environment variables are created by various sources:

- The operating system itself
- Your `.bashrc` where custom environment variables are set
- Other scripts or programs which may set new environment variables

---

Environment Variable Examples

Environment variables are UPPERCASE

$SHELL - What shell you are using.

$USER - Who you're logged in as.

$HOME - Path to your home directory.

$TEMP - Path to a suitable directory to write temporary files.

You can see other environment variables by using the `env` command.

You should not change the value of existing environment variables (without good reason) because it might break things that depend on those values.

Double vs Single Quotes:

Strings wrapped in single quotes are used exactly as they appear by the shell.

String wrapped in double quotes will have variable names replaced with the value of that variable.

```
#!/bin/bash
set -eu

name="John"
echo "Hi $name"  #=> Hi John
echo 'Hi $name'  #=> Hi $name
```

Parts of a Script: Conditionals

Very similar to Python!

The `if` condition is inside square brackets `[[ ]]` and must evaluate to `true` or `false`

An `if` condition is ended by `fi` (which is if backwards!)

```
#!/bin/bash
set -eu

echo -n "Enter a number: "
read var

if [[ $var -gt 10 ]]
then
    echo "The variable is greater than 10."
fi
```

Ask the class what they think the `-gt` operator is doing here based on what the script output is. (Greater Than comparison operator)

Numerical Conditional operators

These are used with number-type variables

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

String Conditional operators

These are used with string-type variables

- `=` or `==` : Strings are equal
- `!=` : Strings are not equal
- `-z` : String has zero length
- `-n` or no operator : String has non-zero length

File Conditional operators

These are used with files

- `-e` or `-a` : File exists
- `-f` : File exists and is regular file
- `-d` : File exists and is directory

- `-s` : File exists and is not empty
- `-r` : File is readable
- `-w` : File is writable
- `-x` : File is executable

```
if [[ -f "$filename" ]]; then
    echo "$filename exists!"
fi
```

Parts of a Script: Else Conditionals

```
#!/bin/bash
set -eu

echo -n "Enter a number: "
read var

if [[ $var -gt 10 ]]
then
    echo "The variable is greater than 10."
else
    echo "The variable is equal to or less than 10"
fi
```

Like in Python, we can add more 'else-if' conditions to our chain of conditional checks.

```
#!/bin/bash
set -eu

echo -n "Enter a number: "
read var

if [[ $var -gt 10 ]]
then
    echo "The variable is greater than 10."
elif [[ $var -eq 10 ]]
then
    echo "The variable is 10."
else
    echo "The variable is less than 10"
fi
```

Parts of a Script: Substitution

We can substitute function output for variables.

Any output value produced by a function inside `$( )` can be used as if it were a variable.

```
#!/bin/bash
set -eu

str="I love python"

echo $( echo $str | sed 's/python/bash/' )
```

What do you think is happening here?

`sed s/python/bash` replaces python with bash in the input it is given, which is piped in from the echo command

---

Exit codes

When any Unix application finishes running, it returns an `exit code`.

There are many exit codes, but 0 means success, and any other code (e.g. 1, 2, 3) is usually an error.

In `if` conditions, the exit code 0 evaluates to `true`, anything else is considered `false`.

```
#!/bin/bash
set -eu
if ! which python3;
then
    apt-get install python3
fi
```

Parts of a Script: For Loop

Can be defined as specific values or as a range.

```
#!/bin/bash
set -eu

for i in {1..5};
do
    echo "Welcome ${i}"
done
```

Must include a **done** to denote when the loop ends.

What do you think is happening here?

For Loop Practical Example

```
#!/bin/bash
set -eu

# Switch into every git repo directory under a path,
# pull the latest changes and switch back to
# the parent directory to repeat the process

reposPath="/home/linus/myrepos"
for repoDir in $(ls -d ${reposPath}/*/)
do
    cd ${repoDir}
    git pull
    cd ../
done
```

Parts of a Script: While Loop

Much like python, will loop until condition evaluates to false.

```
#!/bin/bash
set -eu

n=1
while [ $n -le 5 ]
do
    echo "Running $n times"
    n=$((n + 1))
done
```

What do you think is happening here?

While Loop Practical Example

```
#!/bin/bash
set -eu

# Read a list of URLs from an input file
# For each URL ping the url to check it is working

urlsFile=${1}
while IFS= read -r url
do
    echo -e "\n\n...CHECKING ${url} IS UP...\n\n"
    ping -c 3 ${url}
done < "${urlsFile}"
```

Tips

Don't forget to add execution rights to your shell scripts using `chmod +x <filename>`

Double quotes over single quotes

Scripting cheat sheet

Exercise

Write a script that will run your test suite and (if the tests pass!) do a git commit

See exercises handout for detailed requirements

Solution

```
#!/bin/bash
set -eu

project_dir="$HOME/mini-project"

if [[ $(pwd) != ${project_dir} ]];
then
    echo -e "Not in project directory.Changing directory to
${project_dir}\n"
    cd ${project_dir}
fi

if pytest;
then
    git commit -am "${1}"
fi
```

Learning Objectives Revisited

We should be able to:

- Describe what shell scripting is used for, and why we might choose it.
- Practice running scripts in a shell
- Know how Arguments and Variables work with scripts
- Describe which conditonals and loops can be used in shell scripts
- Write our own (simple!) shell script

Key Terms and Definitions

Shell Script: A computer program to automate repetitive tasks.

Shebang or Hashbang: This tells the shell what interpreter to use.

Shell options: Settings that change shell and/or script behaviour.

Environment Variables: Variables that our applications inherit from their parent scope.

Further Reading

- Advanced Bash-Scripting Guide
- Shell Scripting Crash Course - Beginner Level (Video)
- The Beginner's Guide to Shell Scripting: The Basics