# Lecture notes

## Topic:

Course:
Date:
Professor/Speaker:

| Questions | Notes |
|---|---|
| | |
| | |
| | |
| | |
| | |
| | |
| | |

## Summary

- < home< de-devops
- 
- **Shell Scripting with Bash**
- **Prerequisites**
- It is assumed you have some experience at using shell commands in a terminal.
- **What is a Shell?**
- A Unix shell is a command-line intepreter that provides a command-line user interface for Unix-like operating systems. It is both an interactive command language and a scripting language.
- The most popular shell is the Bourne-Again Shell or `bash` but others include:

    - `ksh` - Korn Shell
    - `fish` - Friendly Interactive Shell
- When you use commands in the terminal such as `ls`, `mkdir` or `touch` you are working with your computers shell.
- ## What is Shell Scripting?
- In it's simplest form a shell script is a file containing a series of commands. The shell reads this file and carries out the commands as if they have been entered directly on the command-line.
- ## Creating a Bash Script
- Let's take a look at how we can create a simple `Hello World` script and run it from the command-line.
- Common naming convention is to create a file that uses the `.sh` file extension (though it can be run perfectly fine without this).

```
#!/bin/bash

echo "Hello World!
```

1. Notice that the file starts with `#!/bin/bash`. This is known as a shebang. The shebang tells your shell which interpreter program to use to execute your script. It ensures that Bash is used to interpret the script, even if you run the file with a different shell!
2. It consists of a `#!` and the absolute path to the interpreter.
3. To find the path to your `bash` shell you can use the `which` command like so:

```
which bash
```

1. ### Why do we use Bash?
2. The advantages of Bash Scripts include:
    - Simple to create
    - Easy way to run multiple commands
    - Can be automated
    - Pre-installed on most Unix-like operating systems
    - Flexibility
3. ## Using a Bash Script
4. Unfortunately it's not as simple as just creating a file with a shebang and few commands. To understand this let's take a look a file permissions.

5. **File Permissions**
6. When you create a new file it will be created with a certain set of permissions. The easiest way to see these is to use `ls -l` which shows the contents of a directory with details about the permissions along with other information.
7. It should look something like this.

```
$ ls -l

total 0
drwxr-xr-x 2 dcrawley staff 64B 2 Nov 16:44 example-dir
-rw-r--r-- 1 dcrawley staff 0B 2 Nov 16:22 hello-world.sh
```

1. The permissions string is made up of 10 characters starting with a `d` for directories or a `-` for files.
2. The first three positions (after the `-` or `d`) indicate the owner's permissions.
   - `r` indicates the owner can read the file.
   - `w` indicates the owner can write to the file.
   - `x` indicates the owner can execute the file.
3. The next 3 positions designate the permission for the group (if you don't share a group space you need not be concerned with group permissions).
4. The last three positions are for the world/anyone.
5. **Running a Bash Script**
6. In order to run your script you will need execute permissions for it. In order to give yourself the correct permissions you can use the `chmod` command.
   - `chmod u+x FILE_PATH` - will give executable permissions to the file owner.
7. or
   - `chmod +x FILE_PATH` - will give executable permissions to everyone.
8. Once this is done you can run the file by entering the path to said file in your terminal. For example `./` hello-world.sh.
9. ❗ Make sure you include the `./` otherwise Bash will interpret the filename as a command and you will recieve an error like this:

```
zsh: command not found: hello-world.sh
```

## 1. Bash Syntax

2. **Variable Declaration**
3. Just like other programming languages you can make use of variables.

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

```
greeting="Hello 👋"

echo $greeting
```

1. Those variables can then be used by referencing them with the `$` symbol.
2. It can be tempting to declare your variables with a space around the `=` sign.

```
#!/bin/bash

greeting = "Hello 👋"
```

1. However this will cause an error as Bash interprets `greeting` as a command followed by 2 arguments.
2. If you see an error message like this then check your variable declarations!

```
./hello-world.sh: line 3: greeting: command not found
```

1. **Passing Arguments**
2. Arguments passed to a script or function aren't named like we're used to with JavaScript and Python but instead they are accessed based on their position.
3. For example here's a script that simply prints whatever arguments are passed to it.

```
#!/bin/bash

echo $1 $2
```

1. If we want to pass arguments the script above we just need to run it like any other (by typing out the path to the file) but then follow it with space seperated arguments like so:

```
$ ./echo-arguments.sh "Hello" "World!"

# Prints "Hello World!" to the terminal
```

1. **Command Substitution**
2. The output of a command can be saved to a variable, this is known as command substitution.
3. This is something we've seen before when testing Python code:

```
PYTHONPATH=$(pwd) pytest
```

1. In the above example the output of the `pwd` command is being saved in the `PYTHONPATH` variable (add subsequently used by the `pytest` command)
2. **Piping**
3. The output of one command can be passed as input to another using the pipe operator ( `|` )
4. Take this example:

```
ls -l | grep ".py"
```

1. This command redirects the output of running `ls` into the `grep` command (`grep` is a very useful command for pattern matching strings) which filters the list of files to find all the python files.
2. **Redirection**
3. You can redirect command input and output using redirection operators.
4. Unlike piping, which only allows passing output to another command, a redirection operator can be used to pass input to a command from a file/stream or send it's output to a file/stream.

```
# This sends the output of the "hello-world" script to a txt file
# Note this will create a new file if it doesn't exist and *overwrite* any existing
content
./hello-world.sh > "output.txt"

# This will *append* the scripts output to the end of the txt file instead of over-
writing it completely
./hello-world.sh >> "output.txt"

# This will *read* the "argument.txt" file and pass it in a input to the "echo-
arguments" script
./echo-arguments.sh < "arguments.txt"
```

1. **Other language features**
2. Bash also supports:
   - Control Flow - Conditional logic with `if` / `else` statements
   - Iteration - Looping with `for` / `while` loops
   - Functions
   - Logical Operators
   - Regex
3. **Errors**
4. Bash errors are interesting because a lot of the time they will not stop execution! However some errors such as syntax errors will stop execution.

5. Take this example:

```
#!/usr/bin/env bash

ls "fake directory"

echo "Did we reach this point?"
```

1. You might think that if the `ls` command fails then the execution of the script would stop. However this is the output:

```
ls: fake directory: No such file or directory
Did we reach this point?
```

1. As you can see the script continues executing as if nothing went wrong.
2. This is the default behaviour as a lot of the time when scripting we don't want to stop as soon as a error occurs, we just want to handle those errors.
3. We won't be going into error handling but if you want to read into it you could start here.
4. ## Debugging
5. When it comes to debugging there are a number of useful tools, here are just a few.
6. ### Echo
7. The `echo` command is very useful in a similar way to Python's `print` or JavaScripts `console.log`. You can use it to print messages or take a look at what is stored in a variable.
8. ### Man
9. There are numerous bash commands and most come with loads of options for different circumstances. You could look up the documentation for each one but there's an even easier way to do this with the `man` command.
10. `man` will print the documentation for any bash command straight to your terminal!

```
$ man ls
```

1. Will print something like this:

```
NAME top
ls - list directory contents
SYNOPSIS top
ls [OPTION]... [FILE]...
DESCRIPTION top
```

```
List information about the FILEs (the current directory by
default). Sort entries alphabetically if none of -cftuvSUX nor
--sort is specified.

Mandatory arguments to long options are mandatory for short
options too.

-a, --all
do not ignore entries starting with .

-A, --almost-all
do not list implied . and ..

--author
with -l, print the author of each file

...
```

1. **Set**
2. As we discovered previously Bash will only halt execution for things like syntax errors. It will not halt execution when a command fails. This is useful in a lot of cases but can cause issues if later commands rely on a previous one working.
3. Thankfully Bash has yet another helpful tool for this issue. The `set` command. We can use this command to tell bash to end execution on any error or to run the script in debug mode.
4. Let's take a look at how to exit when we have an error.

```
set -e

ls "fake directory"

echo "Did we reach this point?"

set +e
```

1. When we run this script we no longer reach the echo command. The `ls` command has failed and the script stops execution.

```
ls: fake directory: No such file or directory
```

1. `set -e` tells bash to exit on error. This is modifying the shells normal behaviour. Therefore is is good practice to undo this modification with `set +e` at the end of the file (or wherever we want to revert it). If the script is run with `source` then any changes will continue for your current session.

2. Another option is to use the `set -x` command to run a script in debug mode. This is useful when writing long scripts or using iteration. For this example let's look at a for loop to count from 0 to 5.

```
for ((i = 0; i < 5; i++)); do
echo "${i}"
done
```

1. Output:

```
0
1
2
3
4
```

1. If we want to find out why I'm not reaching number 5 we can use `set -x` like so:

```
set -x

for ((i = 0; i < 5; i++)); do
echo "${i}"
done

set +x
```

1. Which gives us the following output:

```
+ (( i = 0 ))
+ (( i < 5 ))
+ echo 0
0
+ (( i++ ))
+ (( i < 5 ))
+ echo 1
1
+ (( i++ ))
+ (( i < 5 ))
+ echo 2
2
+ (( i++ ))
+ (( i < 5 ))
+ echo 3
3
+ (( i++ ))
+ (( i < 5 ))
+ echo 4
```

```
4
+ (( i++ ))
+ (( i < 5 ))
+ set +x
```

1. This can be very helpful in seeing exactly where you may have gone wrong.
2. ## The PATH Variable
3. Unix systems have a `PATH` variable that specifies a set of colon (`:`) separated directories that contain executable programs.

```
$ echo $PATH

# You will see something like this though it may have different directories

/Users/dcrawley/.pyenv/shims:/Users/dcrawley/bin:/usr/local/bin:/Users/dcrawley/.local/b
in/usr/local/bin
```

1. Executable programs contained in these directories are able to be used without specifying the file path, this is the reason why commands like `ls` are globally available.
2. If you want to make a shell script globally available as a command you could move it into the `/usr/local/bin` directory or any others listed there. You could also add the scripts parent directory to the `PATH` variable though this could lead to a very messy PATH!
3. ## The Source command
4. The `source` command is built in to bash and other popular shells used in Unix-like operating systems. It is used to read and execute commands from the file given as an argument.
5. It is useful to load functions, variables and config files into your shell scripts.
6. You will have also seen it used to activate Pythons virtual environment - `source venv/bin/activate`
7. ## Further Reading
   - Bash Scripting Cheatsheat
   - Another Cheatsheet
   - Archive of man pages
   - GNU Bash manual