Session 1 Bonus: Shell scripting

In this bonus sheet we're going to go through the basics of shell scripting. Shell scripting involves writing a text file with the suffix `.sh` with a series of commands you would usually run on the terminal. Then when you run the shell script, each command is executed in order. A lot of these concepts are teasers for things we will cover in more depth when we start Python coding.

Exercise 1. Running a basic shell script

A shell script **must** start with a *shebang* which is

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

This tells the system to run the script using the default Bash shell — the program that processes commands in your terminal.

1. Use vim to create a shell script `test.sh`, and make sure the first line is the shebang above. On the next line, write a command from the last practical sheet (maybe `ls -l` for example). Save and close the file, then make sure the user has permissions to execute the file. You can then execute the file from the terminal with

   ```
   ./test.sh
   ```

   Note: The `./` means to run the script located in the current directory.
   Hopefully you observe that running this shell script results in the same as if the command had been ran on the terminal.

2. Now change your shell script to run a series of commands (e.g. make a directory, then move to the directory, then print the location and the contents).

3. One of the most basic terminal commands that we haven't covered is `echo`, a command that prints text or variables to the terminal. You can try it in the terminal by running

   ```
   echo This is some text!
   ```

   Create a shell script that prints the classic "Hello World!"

4. We can add variables into shell scripts by assigning them with a `=` and calling them with a `$`, e.g.

   ```
   #!/bin/bash
   name="Your Name Here"
   echo "Welcome to Python Bootcamp, $name!"
   ```

   Edit this code for your own name and run it to confirm it works as expected.

Exercise 2. Conditionals

We can write a script such that commands only run under certain conditions using `if`, `elif` and `else`

Bash if-elif-else structure

```
if [ condition1 ]; then
    # commands for condition1 true
elif [ condition2 ]; then
    # commands for condition2 true
else
    # commands if none of the above conditions are true
fi
```

Every conditional must start with an `if` which can be the only conditional if desired, if there is an alternative condition `else` can be used, and if there are more than two conditions as many `elif` (else if) statements can be placed between `if` and `else`.

1. Here is an example of a shell script using conditionals

   ```
   #!/bin/bash

   name="Natalie"

   if [ "$name" = "Natalie" ]; then
       echo "Hello Natalie!"
   else
       echo "You are not Natalie."
   fi
   ```

   Run this script, then alter the `name` argument to be your own name and run again.

2. Using the information that in bash `-gt` is greater than, `-lt` and `-eq` and is less than, make a bash script similar to the above that takes an argument `number` and tests if it is less than 5, greater than 5, or equal to 5. It should print to the terminal which one of these three options `number` is. Test this by changing `number` to be each of these options and running to check you get what you expect.

Exercise 3. Loops

Loops allow us to repeat commands multiple times. Two common types of loops in Bash are `for` and `while` loops.

Bash for loop structure

```
for var in list
do
    # commands using $var
done
```

The `for` loop iterates over each element in `list`, executing the commands in the `do` block for each element.

Bash while loop structure

```
while [ condition ]
do
    # commands to run while condition is true
done
```

The `while` loop repeats the commands as long as the `condition` remains true.

1. Run this script that uses a `for` loop to print numbers 1 to 10:

   ```
   #!/bin/bash

   for i in {1..10}
   do
       echo "Count: $i"
   done
   ```

   Run the script and observe the output. Modify the script to print "Halfway there!" only after 5 is printed.

2. Create another script that does the same thing as above but with a while loop.

3. While loops can be problematic as a bug can end in code that runs infinitely - luckily we can stop code on the terminal with `Ctrl + C`. Test this by running this script

   ```
   #!/bin/bash

   while true
   do
     echo "Looping forever..."
   done
   ```

Exercise 4. Passing Arguments to Scripts

Bash scripts can take input arguments from the command line. These arguments are accessed using special variables like `$1`, `$2`, etc., where `$1` is the first argument, `$2` the second, and so on. The variable `$@` represents all arguments, and `$#` gives the number of arguments.

Accessing script arguments

```
#!/bin/bash

echo "First argument: $1"
echo "Second argument: $2"
echo "All arguments: $@"
echo "Number of arguments: $#"
```

When you run this script with arguments, e.g.

```
./script.sh apple banana cherry
```

it will print the corresponding arguments.

1. Create a script called `greet.sh` that takes one argument — a name — and prints:

   ```
   Hello, [name]!
   ```

   If no name is given, it should print:

   ```
   Hello, stranger!
   ```

   Hint: use an `if` statement to check if `$1` is empty with condition `[ "$1" = "" ]`.

2. Modify your `greet.sh` script to accept any number of names and greet each one individually by looping over `$@`. Try running your script with different numbers of arguments to see how it behaves.