Introduction To Shell Scripting

```
module = Module(
    code="ELEE1147",
    name="Programming for Engineers",
    credits=15,
    module_leader="Seb Blair BEng(H) PGCAP MIET MIHEEM FHEA"
)
```

UNIVERSITY OF GREENWICH

Download as a PDF

Scripting

- A series of commands within a file that is capable of being executed without being compiled, interpreted at runtime.
- Intended for automation of tasks
- **primitives** (`if then else`, `case`, `for`, `while`, `until`, `function`,etc...)

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

:(){:|:&};:
```

⚠️ Do not do this.

Identifying a shell script

- naming convention -> `.sh`
- The first line in this file is the "**shebang**"/ **hashbang**" line.

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

- When you execute a file from the shell, the shell tries to run the file using the command specified on the shebang line.
- The `!` is called the "*bang*". The `#` is not called the "she", so sometimes the "*shebang*" line is also called the "*hashbang*".
- `#!` is encoded to the **bytes 23 21** which is the **magic number** of an executable script.
- A magic number is a sequence of bytes at the beginning of a file that allows to identify which is the type of a file, for example, a png file will always begin by the **bytes 89 50 4E 47**

More on `#!`

- The *shebang* line was invented because scripts are not compiled, so they are not executable files, but people still want to "*run*" them.

- The shebang line specifies exactly how to run a script.

  - In other words, this shebang line says that:

```
$ ./basics.py
```

  - the shell will actuall run `/usr/bin/env python basics.py`
  - We use `#!/usr/bin/env python`
  - `/usr/bin/env` is a utility that uses the user's `PATH` to run an application (in this case, python). Thus, it's more portable.

Task 1.

The `#!` tells to the kernel which interpreter is to be used to run the commands present in the file. If you run a script without specifying the interpreter, the shell will spawn another instance of itself and try to run the commands in the script.

```
$ nano script.sh
```

```
#! /usr/bin/env cat
VAR1=Hello
VAR2=World!
VAR3=Goodbye

echo ${VAR1} ${VAR2}
echo ${VAR3} ${VAR2}

history
```

```
$ chmod +x script.sh && ./script.sh
```

`chmod` vs `bash`

- `chmod` change file mode bits
  - `rwx rwx rwx` || `777`
  - `chmod +x` changes all modes to include executable
- `bash` command language interpreter that executes commands read from the standard input or from a file
- `bash` will interpret the contents of the file and run the lines as commands.
- `./script.sh` takes the `#!` and passes the script to the command

```
#! /usr/bin/env cat script.sh

#! /usr/bin/env bash script.sh
```

Note of file permissions

- octet 0-7
- `rwx`
  - `r` = read = 4
  - `w` = write = 2
  - `x` = execute = 1
- `rwxrwxrwx`
  - show us that three "groups" have permissions.
    - user, group and rest of the world

- `d` = directory
- `.` = file in-situ of its directory
- `l` = link to another location

Variables

- Bash does not have a **type system**, `int`, `char`, `var`..,etc
- Bash only saves them as a string
- We can declare variables in a Bash script. Unlike other programming languages,it can only save `string` values. Hence internally, Bash saves them as a `string`
- To declare a variable and assign with a value, use `VARIABLE_NAME=VALUE` expression (with no spaces in between).

```
GREETING=Hi
STATEMENT="my name is,"
INTERROGATIVEPRONOUN1=what?
INTERROGATIVEPRONOUN2=who?
NAME=${1:-"Slim Shady"}
CONFUSION=huh?
ALLITERATION=chka-chka
NUMBER=${:-default}
```

Calculations

- Arithmetic Expansion
  - `$((...))`
  - `VAR=$((expression))`

```
#! /usr/bin/env bash
echo $((x=4,y=5,z=x*y,u=z/2))
X=4
Y=5
Z=$((${X}*${Y}))
U=$((${Z}/2))
echo U=${U}, Z=${Z}
```

Output:

```
> U=10, Z=20
```

Task 2.

```
$ nano int-or-string.sh
```

```
$ chmod +x int-or-string.sh && ./int-or-string.sh
```

```
#! /usr/bin/env bash
A=2334                    # Integer... though still a string
echo "A = ${A} "          # a = 2335
A=$(( ${A} + 1 ))         # increment A by 1.
echo "A = ${A} "          # a = 2335, Integer, still.
echo                      # new empty line
```

Conditionals

- Spacing matters

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

if [[$1 -lt 10]];then # error
    echo you are an amazing programmer
fi

if [[ $1 -lt 10 ]];then
    echo well done...
fi

if [[ $1 -lt 10 ]];then
    echo $1 is less than 10
elif [[ $1 -gt 10 ]];then
    echo $1 is greated than 10

else
    echo $1 is equal to 10...
fi
```

For Loops

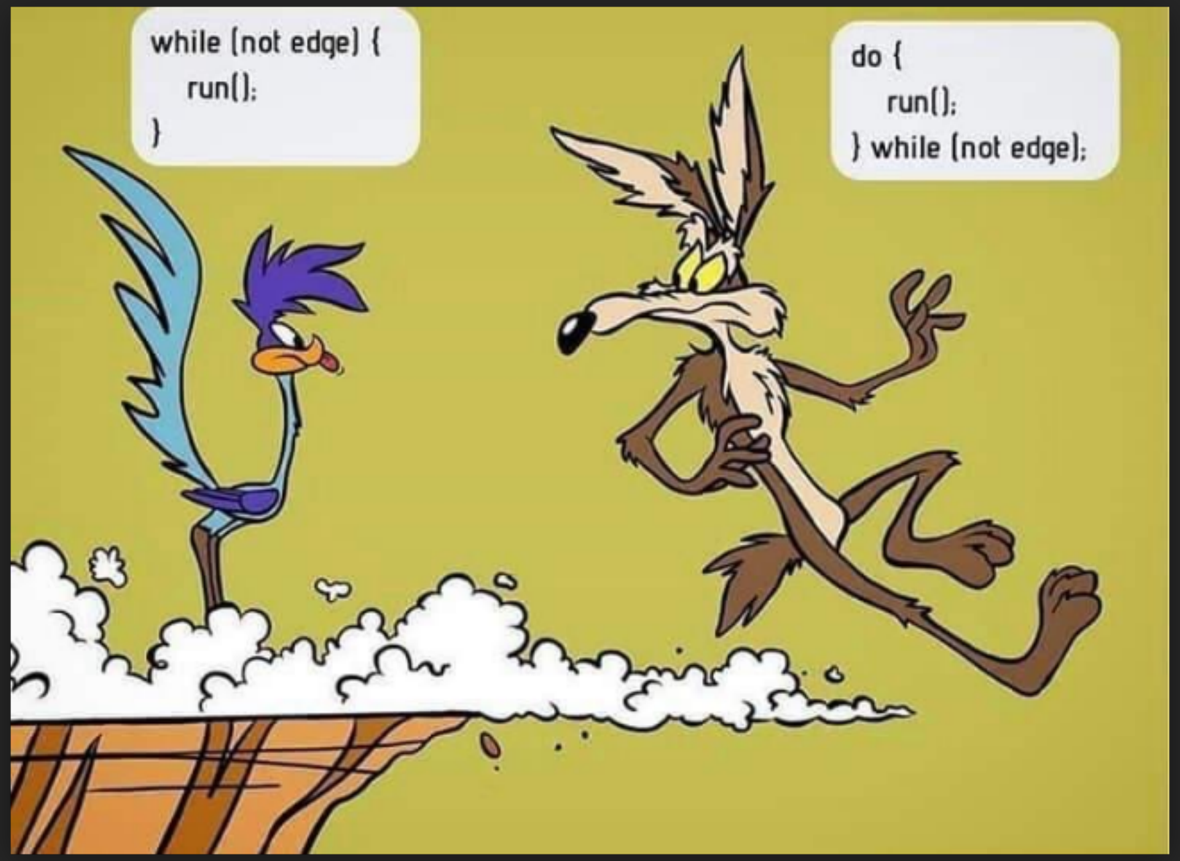
```
for a in 1 2 3 ; do
        touch foo_$a
done
```

```
for a in $( seq 1 10 ) ; do
        touch foo_$a
done
```

`while` , `until`

```
counter=1
while [ $counter -le 10 ]
do
echo $counter
((counter++))
done
```

```
counter=1
until [ $counter -gt 10 ]
do
echo $counter
((counter++))
done
```



Task 3.

```
#! /usr/bin/env bash
DIR="task5"

# if directory (-d) does not exist (!), then create it
if [[ ! -d ${DIR} ]]; then
  mkdir ${DIR} && echo "${DIR} created" # if successful printout created
fi
# a becomes 1 then 2, and 3 and this is appended to the word foo_ to
# create files in the directory that was created.
for a in 1 2 3 ; do
   touch ${DIR}/foo_$a
done
```

Flags

- Using flags is a common way of passing input to a script.
- When passing input to the script, there's a flag (usually a single letter) starting with a hyphen (`-`) before each argument.
- The `getopts` function reads the flags in the input, and `OPTARG` refers to the corresponding values:

```
while getopts u:a:f: flag
do
    case "${flag}" in
        u) username=${OPTARG};;
        a) age=${OPTARG};;
        f) fullname=${OPTARG};;
    esac
done
echo "Username: $username" echo "Age: $age" echo "Full Name: $fullname"
```

Shell Special Parameters

- `$!` is used to reference the PID of the most recently executed command in background.
- `$$` is used to reference the process ID of bash shell itself
- `$#` is quite a special bash parameter and it expands to a number of positional parameters in decimal.`
- `$0` bash parameter is used to reference the name of the shell or shell script.
- `$1` first supplied parameter, `$1...n`
- `$*` Expands to the the positional parameters starting from one.
- `"$*"` Does the same thing but creates spaces between each argument

Tasks 4.

```
$ ./parameters.sh -f 'Slim Shady' -a 25 -u Marshall
```

```
Username : Marshall
Age: 25
Full Name: Slim Shady
```

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

while getopts u:a:f: flag
do
    case "${flag}" in
        u) username=${OPTARG};;
        a) age=${OPTARG};;
        f) fullname=${OPTARG};;
    esac
done
echo "Username: $username" echo "Age: $age" echo "Full Name: $fullname"
```

Task 5: Reading from CLI

- Using the `stdin` stream by invocating `read`

```
echo -n "Enter your name:"
read NAME
echo "Your name is:" ${NAME}

read -p "Enter your name: " NAME
echo Your name is ${NAME}.

read  -t  5 -p "Enter your password: "$'\n' -s PASSWORD
echo ${PASSWORD}

read -a WORDS <<< "Hello world!"
echo ${WORDS[0]}
echo ${WORDS[1]}
```