Studio 1
Term 1 Assignment
Report

Custom Commands: Code, Rationales, Testing and Deficiencies:

Custom Command 1: tempcon.bash:

Pseudocode Screenshot:

```
IF [ INPUT != 1 ]
THEN
        DISPLAY "Error: Command needs input"
        DISPLAY usage
        EXIT 1
END IF

CASE [ currentInput ] IN
        -F
                PRINT "Enter fahrenheit"
                READ fahrenheit

                IF [ fahrenheit is numerical ]
                THEN
                        CALCULATE celsius from fahrenheit
                        CALCULATE kelvin from fahrenheit
                ELSE
                        PRINT "Error: Invalid input"
                        EXIT 2
                END IF
                PRINT "fahrenheit equals celsius and kelvin"
NEXT CASE
        -C
                PRINT "Enter celsius"
                READ celsius

                IF [ celsius is numerical ]
                THEN
                        CALCULATE fahrenheit from celsius
                        CALCULATE kelvin from celsius
                ELSE
                        PRINT "Error: Invalid input"
                        EXIT 3
                END IF
                PRINT "celsius equals fahrenheit and kelvin"
NEXT CASE
        -K
                PRINT "Enter kelvin"
                READ kelvin

                IF [ kelvin is numerical ]
                THEN
                        CALCULATE fahrenheit from kelvin
                        CALCULATE celsius from kelvin
                ELSE
                        PRINT "Error: Invalid input"
                        EXIT 4
                END IF
                PRINT "kelvin equals fahrenheit and celsius"
NEXT CASE
        *
                PRINT "Error: Invalid option"
                PRINT usage
                EXIT 5
END CASE
EXIT 0
```

Script Screenshot:

```
#!/bin/bash
#
# tempcon.bash - A command which converts between different temperature units
# Usage - tempcon.bash -[F|C|K]
#
# Author - Waqas Musharaf
# Date - 12.04.2015

if [ $# -ne 1 ] # If there is not exactly 1 argument, do the following:
then
        echo "Error: Command needs an option argument to convert" # Informs the user on what has gone wrong
        echo "Usage: tempcon.bash -[F|C|K]" # Informs the user on how to use the command
        echo "'F' indicates Fahrenheit, 'C' indicates Celsius and 'K' indicates Kelvin" # Informs the user on what the valid arguments are
        exit 1 # Exits safely, however with status 1, informing the user that something went wrong
fi # End of 'if' statement

option="${1}" # Renames the initial argument to 'option'
case ${option} in
        -F) # If the 'option' is '-F', do the following:

        echo "Input a numerical value of Fahrenheit to convert:"
        read fTemp # Stores the user input in the variable 'fTemp'

        if [[ "$fTemp" =~ ^-?[0-9]+(\.[0-9]+)?$ ]] # Checks the input against a Regular Expression, to ensure the input is numerical
        then
                cTemp="$(echo "scale=2;($fTemp-32)/1.8" | bc)" # Converts the input to Celsius, and stores the value in a variable, 'cTemp'
                kTemp="$(echo "scale=2;($fTemp+459.67)/1.8" | bc)" # Converts the input to Kelvin, and stores the value in a variable, 'kTemp'
        else # If the condition is not met, do the following:

                echo "Error: Non-numerical input detected" # Informs the user on what has gone wrong
                exit 2 # Exits safely, however with status 2, informing the user that something went wrong
        fi # End of 'if' statement

        echo "$fTemp Fahrenheit is equal to $cTemp Celsius and $kTemp Kelvin" # Outputs the conversion from Fahrenheit to Celsius and Kelvin
        ;; # Next case

        -C) # If the 'option' is '-C', do the following:

        echo "Input a numerical value of Celsius to convert:"
        read cTemp # Stores the user input in the variable 'cTemp'

        if [[ "$cTemp" =~ ^-?[0-9]+(\.[0-9]+)?$ ]] # Checks the input against a Regular Expression, to ensure the input is numerical
        then
                fTemp="$(echo "scale=2;($cTemp*(9/5))+32" | bc)" # Converts the input to Fahrenheit, and stores the value in a variable, 'fTemp'
                kTemp="$(echo "scale=2;$cTemp+273.15" | bc)" # Converts the input to Kelvin, and stores the value in a variable, 'kTemp'
        else # If the condition is not met, do the following:

                echo "Error: Non-numerical input detected" # Informs the user on what has gone wrong
                exit 3 # Exits safely, however with status 3, informing the user that something went wrong
        fi # End of 'if' statement

        echo "$cTemp Celsius is equal to $fTemp Fahrenheit and $kTemp Kelvin" # Outputs the conversion from Celsius to Fahrenheit and Kelvin
        ;; # Next case

        -K) # If the 'option' is '-K', do the following:

        echo "Input a numerical value of Kelvin to convert:"
        read kTemp # Stores the user input in the variable 'kTemp'

        if [[ "$kTemp" =~ ^-?[0-9]+(\.[0-9]+)?$ ]] # Checks the input against a Regular Expression, to ensure the input is numerical
        then
                fTemp="$(echo "scale=2;($kTemp*(9/5))-459.67" | bc)" # Converts the input to Fahrenheit, and stores the value in a variable, 'fTemp'
                cTemp="$(echo "scale=2;$kTemp-273.15" | bc)" # Converts the input Celsius, and stores the value in a variable, 'cTemp'
        else # If the condition is not met, do the following:

                echo "Error: Non-numerical input detected" # Informs the user on what has gone wrong
                exit 4 # Exits safely, however with status 4, informing the user that something went wrong
        fi # End of 'if' statement

        echo "$kTemp Kelvin is equal to $fTemp Fahrenheit and $cTemp Celsius" # Outputs the conversion from Kelvin to Fahrenheit and Celsius
        ;; # Next case

        *) # If the 'option' is none of the above, do the following:

        echo "Error: Invalid option argument, valid option arguments are listed below" # Informs the user on what has gone wrong
        echo "Usage: tempcon.bash -[F|C|K]" # Informs the user on how to use the command
        echo "'F' indicates Fahrenheit, 'C' indicates Celsius and 'K' indicates Kelvin" # Informs the user on what the valid arguments are
        exit 5 # Exits safely, however with status 5, informing the user that something went wrong
        ;; # Next case
esac # End of 'case' statement
exit 0 # Exits safely, with status 0, informing the user that everything went as it should
```

Rationale:

tempcon.bash is a command used to convert between three different units of temperature; Fahrenheit, Celsius and Kelvin. I believe this command is useful as all three of these units are used commonly and to convert between them manually can be difficult, especially if you are unaware of the formulae.

To use tempcon.bash, the user must enter tempcon.bash into their terminal, followed by a suffix of either –F, -C, or –K (depending on the unit they wish to convert from). Upon a valid input, the user is asked the value of their chosen unit to be converted. Assuming all inputs are valid, the command will then calculate and display the conversion from one unit into the remaining two.

Testing:

tempcon.bash requires an initial argument, either –F, -C, or –K as explained above. This is what happens when you attempt to run the command without an initial argument:

```
u1561634@ouranos:~/bin$ tempcon.bash
Error: Command needs an option argument to convert
Usage: tempcon.bash -[F|C|K]
'F' indicates Fahrenheit, 'C' indicates Celsius and 'K' indicates Kelvin
```

With an invalid initial argument:

```
u1561634@ouranos:~/bin$ tempcon.bash -D
Error: Invalid option argument, valid option arguments are listed below
Usage: tempcon.bash -[F|C|K]
'F' indicates Fahrenheit, 'C' indicates Celsius and 'K' indicates Kelvin
```

If a valid initial argument is submitted, the command asks for a value of the initial argument to convert:

```
u1561634@ouranos:~/bin$ tempcon.bash -F
Input a numerical value of Fahrenheit to convert:
```

However, it will only accept numerical arguments, including decimals and negative numbers. This is what happens if you input a non-numerical argument:

```
u1561634@ouranos:~/bin$ tempcon.bash -F
Input a numerical value of Fahrenheit to convert:
Test
Error: Non-numerical input detected
```

Or an argument with both numeric and non-numeric parts:

```
u1561634@ouranos:~/bin$ tempcon.bash -F
Input a numerical value of Fahrenheit to convert:
Test2
Error: Non-numerical input detected
```

Or no argument at all:

```
u1561634@ouranos:~$ tempcon.bash -F
Input a numerical value of Fahrenheit to convert:

Error: Non-numerical input detected
```

However, if the input is valid, the command will convert the inputted value of the inputted unit into the remaining two types. Here it is with an integer value input:

```
tempcon.bash -F
Input a numerical value of Fahrenheit to convert:
100
100 Fahrenheit is equal to 37.77 Celsius and 310.92 Kelvin
```

Or with a negative integer value:

```
u1561634@ouranos:~/bin$ tempcon.bash -F
Input a numerical value of Fahrenheit to convert:
-50
-50 Fahrenheit is equal to -45.55 Celsius and 227.59 Kelvin
```

Or with a decimal value (this time with the Celsius unit, to change it up):

```
tempcon.bash -C
Input a numerical value of Celsius to convert:
5.5
5.5 Celsius is equal to 41.90 Fahrenheit and 278.65 Kelvin
```

Or with a negative decimal value (this time with Kelvin):

```
u1561634@ouranos:~/bin$ tempcon.bash -K
Input a numerical value of Kelvin to convert:
-20.6
-20.6 Kelvin is equal to -496.75 Fahrenheit and -293.75 Celsius
```

Deficiencies:

tempcon.bash uses arbitrary-precision arithmetic through the 'bc' calculator language to convert between different temperature units. This can result in floating-point errors occasionally affecting the results of the calculations and inaccurate conversions being displayed. Due to this being a fault of floating-point arithmetic itself, it is unlikely that this issue could be fixed.

Custom Command 2: validEmail.bash:

Pseudocode Screenshot:

```
DECLARE regex

IF [ INPUT < 1 ]
THEN
        DISPLAY "Error: Command needs input"
        DISPLAY usage
        EXIT 1
END IF

WHILE [ INPUT > 0 ]
        IF [ currentInput conforms to regex ]
        THEN
                DISPLAY "currentInput is a valid email"
        ELSE
                DISPLAY "currentInput is not a valid email"
        END IF
DONE
EXIT 0
```

Script Screenshot:

```
#!/bin/bash
#
# validEmail.bash - A command which checks arguments against an email regular expression and returns validity
# Usage - validEmail.bash [argument1] [argument2] ...
#
# Author - Waqas Musharaf
# Date - 12.08.2015

regex="^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}$" # An email regular expression, used to check email validity

if [ $# -lt 1 ] # If the number of arguments is less than 1, do the following:
then
        echo "Error: Command needs arguments to validate" # Informs the user on what has gone wrong
        echo "Usage: validEmail.bash [argument1] [argument2] ..." # Informs the user on how to use the command
        exit 1 # Exits safely, however with status 1, informing the user that something went wrong
fi # End of 'if' statement

while [ $# -gt 0 ] # As long as there arguments, do the following:
do
        email="$1" # Rename the current variable to 'email'

        if [[ $email =~ $regex ]] # If the current email conforms with the regular expression, do the following:
        then
                echo "Email address '$email' is a VALID email address" # Inform the user that the current email is valid

        else # If the 'if' statement's conditions are not met, do the following:

                echo "Email address '$email' is NOT a valid email address" # Inform the user that the current email is not valid

        fi # End of 'if' statement
        shift # Shifts the current argument out of the 'array' of arguments

done # When the 'while' condition is no longer valid, the while loop is 'done'
exit 0 # Exits safely, with status 0, informing the user that everything went as it should
```

Rationale:

validEmail.bash is a command to check multiple arguments against a regular expression for email address format validity and return the results to the user. I believe this command is useful as there are many people unaware of what the correct email address format is and they could use this command to validate any addresses they are unsure of.

To use validEmail.bash, the user must enter validEmail.bash into their terminal, followed by one or more arguments to validate. Assuming there is at least one argument inputted, the command will validate each argument against the email address regular expression within the command and return the validity of each argument to the user.

Testing:

validEmail.bash requires at least one initial argument, as explained above. This is what happens when you attempt to run the command without an initial argument:

```
u1561634@ouranos:~$ validEmail.bash
Error: Command needs arguments to validate
Usage: validEmail.bash [argument1] [argument2] ...
```

If the command is run with at least one argument, it will always run successfully, since the entire function of the command is to validate input. This is what happens when one valid email address is inputted as the initial argument:

```
u1561634@ouranos:~$ validEmail.bash test_email@domain.com
Email address 'test_email@domain.com' is a VALID email address
```

Or with one invalid email address argument:

```
u1561634@ouranos:~$ validEmail.bash testemail.domain.com
Email address 'testemail.domain.com' is NOT a valid email address
```

Or with multiple valid email address arguments:

```
u1561634@ouranos:~$ validEmail.bash test_email@domain.com u1561634@unimail.hud.ac.uk blah@blah.co.uk
Email address 'test_email@domain.com' is a VALID email address
Email address 'u1561634@unimail.hud.ac.uk' is a VALID email address
Email address 'blah@blah.co.uk' is a VALID email address
```

Or with multiple invalid email address arguments:

```
u1561634@ouranos:~$ validEmail.bash test_email+domain.com u1561634@unimailhudacuk blah@blahcouk
Email address 'test_email+domain.com' is NOT a valid email address
Email address 'u1561634@unimailhudacuk' is NOT a valid email address
Email address 'blah@blahcouk' is NOT a valid email address
```

Or with a mixture of valid and invalid email address arguments:

```
u1561634@ouranos:~$ validEmail.bash should_be+valid@domain.com shouldnot_be@domaincom
Email address 'should_be+valid@domain.com' is a VALID email address
Email address 'shouldnot_be@domaincom' is NOT a valid email address
```

Or another mixture of valid and invalid email address arguments:

```
u1561634@ouranos:~$ validEmail.bash canIBreakThisCommand@maybe.com test@test.te.st chicken
Email address 'canIBreakThisCommand@maybe.com' is a VALID email address
Email address 'test@test.te.st' is a VALID email address
Email address 'chicken' is NOT a valid email address
```

Deficiencies:

Although validEmail.bash is a good command to validate email address formatting, it doesn't actually tell you if the email address exists; that is, can you send emails to this address? Another deficiency of this command is the regular expression it uses to check the arguments. There are many regular expressions used to validate email addresses, and they don't all accept/reject the same things. One could argue that another regular expression is more accurate at identifying valid email address formatting than the one I have used.

Main Commands: Code and Testing:

chmx.bash:

Script Screenshot:

```
#!/bin/bash
#
# chmx.bash - A command that takes filenames as its arguments and makes those files executable to all users
# Usage - chmx.bash [argument1] [argument2] ...
#
# Author - Waqas Musharaf
# Date - 11.30.2015

if [ $# -lt 1 ] # If there is less than 1 argument, do the following:
then
        echo "Error: Command needs file arguments to make exectuable" # Informs the user what has gone wrong
        echo "Usage: chmx.bash [argument1] [argument2] ..." # Informs the user how to use the command
        exit 1 # Exits safely, however with status 1, informing the user that something went wrong

fi # End of 'if' statement

while [ $# -gt 0 ] # As long as there are arguments, do the following:
do
        filename="$1" # Renames the current variable
        chmod u+x "$filename" # Grants users execute permissions
        chmod g+x "$filename" # Grants group execute permissions
        chmod o+x "$filename" # Grants others execute permissions
        shift # Shifts current argument out of the 'array' of arguments

done # When the 'while' condition is no longer valid, the while loop is 'done'
exit 0 # Exits safely, with status 0, informing the user that everything went as it should
```

Testing:

chmx.bash requires at least one initial argument, in order to change the permissions for said arguments. This is what happens when you attempt to run the command without an initial argument:

```
u1561634@ouranos:~/all$ chmx.bash
Error: Command needs file arguments to make exectuable
Usage: chmx.bash [argument1] [argument2] ...
```

Or with an incorrect argument (non-existent file):

```
u1561634@ouranos:~/all$ chmx.bash hello
chmod: cannot access âhelloâ: No such file or directory
chmod: cannot access âhelloâ: No such file or directory
chmod: cannot access âhelloâ: No such file or directory
```

Here is what happens when the command is used with one correct file argument:

```
u1561634@ouranos:~/all$ chmx.bash nusers
u1561634@ouranos:~/all$
```

As it can be seen above, nothing is returned when the command runs successfully. But when 'ls –l' is run, it can be seen that the command has given all users execute rights for the file 'nusers'. Below are screenshots of the 'ls –l' command results before and after 'chmx.bash nusers' is run:

```
---------- 1 u1561634 stud1516  179 Dec  7 15:34 nusers
```
(BEFORE)

```
---x--x--x 1 u1561634 stud1516  179 Dec  7 15:34 nusers
```
(AFTER)

Here is what happens when the command is used with multiple correct file arguments:

```
u1561634@ouranos:~/all$ chmx.bash useable useablev2 useablev3
u1561634@ouranos:~/all$
```

```
---------- 1 u1561634 stud1516  567 Nov 23 10:02 useable
---------- 1 u1561634 stud1516  615 Nov 23 10:30 useablev2
---------- 1 u1561634 stud1516  621 Nov 23 10:43 useablev3
```
(BEFORE)

```
---x--x--x 1 u1561634 stud1516  567 Nov 23 10:02 useable
---x--x--x 1 u1561634 stud1516  615 Nov 23 10:30 useablev2
---x--x--x 1 u1561634 stud1516  621 Nov 23 10:43 useablev3
```
(AFTER)

Here is what happens when the command is used with multiple incorrect file arguments:

```
u1561634@ouranos:~/all$ chmx.bash blah blahblah blahblahblah
chmod: cannot access âblahâ: No such file or directory
chmod: cannot access âblahâ: No such file or directory
chmod: cannot access âblahâ: No such file or directory
chmod: cannot access âblahblahâ: No such file or directory
chmod: cannot access âblahblahâ: No such file or directory
chmod: cannot access âblahblahâ: No such file or directory
chmod: cannot access âblahblahblahâ: No such file or directory
chmod: cannot access âblahblahblahâ: No such file or directory
chmod: cannot access âblahblahblahâ: No such file or directory
```

And finally, here is what happens when the command is used with a mixture of correct and incorrect file arguments:

```
u1561634@ouranos:~/all$ chmx.bash cls.bash compstring notAFile
chmod: cannot access ânotAFileâ: No such file or directory
chmod: cannot access ânotAFileâ: No such file or directory
chmod: cannot access ânotAFileâ: No such file or directory
```

The command has recognised that 'notAFile' does not exist, but has not commented on 'cls.bash' or 'compstring'. Looking at the before and after of the 'ls –l' command, it can be seen that the command has worked for the two files that existed and was unable to change permissions for the file that didn't exist as it didn't exist:

```
---------- 1 u1561634 stud1516  374 Dec  4 11:57 cls.bash
---------- 1 u1561634 stud1516 1317 Nov 16 13:49 compstring
```

(BEFORE)

```
---x--x--x 1 u1561634 stud1516  374 Dec  4 11:57 cls.bash
---x--x--x 1 u1561634 stud1516 1317 Nov 16 13:49 compstring
```

(AFTER)

del.bash:

Script Screenshot:

```
#!/bin/bash
#
# del.bash - A command similar to rm which copies files to a '.waste' directory.
# Usage - del.bash [argument1] [argument2] ...
#
# Author - Waqas Musharaf
# Date - 12.02.2015

if [ $# -lt 1 ] # If there is less than 1 argument, do the following
then
        echo "Error: Command needs file arguments to move to '.waste'" # Informs the user what has gone wrong
        echo "Usage: del.bash [argument1] [argument2] ..." # Informs the user how to use the command
        exit 1 # Exits safely, however with status 1, informing the user that something went wrong

fi # End of 'if' statement

while [ $# -gt 0 ] # As long as there are arguments, do the following:
do
        filename="$1" # Renames the current variable
        mv $filename ~/.waste # Copies the current variable to the '.waste' directory
        shift # Shifts current argument out of the 'array' of arguments

done # When the 'while' condition is no longer valid, the while loop is 'done'
exit 0 # Exits safely, with status 0, informing the user that everything went as should
```

Testing:

del.bash requires at least one initial argument, in order to move said arguments to the '.waste' folder. This is what happens when you attempt to run the command without an initial argument:

```
u1561634@ouranos:~/all$ del.bash
Error: Command needs file arguments to move to '.waste'
Usage: del.bash [argument1] [argument2] ...
```

Or with an incorrect argument (non-existent file):

```
u1561634@ouranos:~/all$ del.bash blah.blah
mv: cannot stat âblah.blahâ: No such file or directory
```

Here is what happens when the command is used with one correct file argument:

```
u1561634@ouranos:~/all$ del.bash test
u1561634@ouranos:~/all$
```

As it can be seen above, nothing is returned when the command runs successfully. But when the current directory is changed to '.waste' and 'ls' is run, it can be seen that 'del.bash' has moved the file 'test' to '.waste':

```
u1561634@ouranos:~/all$ cd
u1561634@ouranos:~$ cd .waste
u1561634@ouranos:~/.waste$ ls
test
```

Here is what happens when the command is used with multiple correct file arguments:

```
u1561634@ouranos:~/all$ del.bash test2 test3 test4
u1561634@ouranos:~/all$
```

```
u1561634@ouranos:~/all$ cd
u1561634@ouranos:~$ cd .waste
u1561634@ouranos:~/.waste$ ls
test2  test3  test4
```

Here is what happens when the command is used with multiple incorrect file arguments:

```
u1561634@ouranos:~/all$ del.bash 1 2 3 4 5
mv: cannot stat â1â: No such file or directory
mv: cannot stat â2â: No such file or directory
mv: cannot stat â3â: No such file or directory
mv: cannot stat â4â: No such file or directory
mv: cannot stat â5â: No such file or directory
```

And finally, here is what happens when the command is used with a mixture of correct and incorrect file arguments:

```
u1561634@ouranos:~/all$ del.bash testA testB testtest testC
mv: cannot stat âtesttestâ: No such file or directory
```

The command has recognised that 'testtest' does not exist, but has not commented on 'testA', 'testB' or 'testC'. If the directory is changed to '.waste' and the 'ls' command is run, it can be seen that the 'del.bash' command has moved 'testA', 'testB' and 'testC' to the '.waste' directory.

```
u1561634@ouranos:~/all$ cd
u1561634@ouranos:~$ cd .waste
u1561634@ouranos:~/.waste$ ls
testA  testB  testC
```

emptywaste.bash:

Script Screenshot:

```
#!/bin/bash
#
# emptywaste.bash - A command that deletes for good everything in the '.waste' directory
# Usage - emptywaste.bash
#
# Author - Waqas Musharaf
# Date - 12.02.2015

if [ $# -gt 0 ] # If there are more than 0 arguments, do the following:
then
        echo "Error: Command does not require arguments" # Informs the user what has gone wrong
        echo "Usage: emptywaste.bash" # Informs the user how to use the command
        exit 1 # Exits safely, however with status 1, informing the user that something went wrong

fi # End of 'if' statement

rm -rf /home/stud/u1561634/.waste/* # Removes all files in the '.waste' directory
# Note: This command will remove ALL the files in the '.waste' directory from ALL directories due to the extension '-f' (force)

exit 0 # Exits safely, with status 0, informing the user that everything went as it should
```

Testing:

emptywaste.bash does not require any initial arguments, as all it does is empty the '.waste' folder. This is what happens when you attempt to run the command with (an) initial argument(s):

```
u1561634@ouranos:~/all$ emptywaste.bash test
Error: Command does not require arguments
Usage: emptywaste.bash
u1561634@ouranos:~/all$ emptywaste.bash hello blah
Error: Command does not require arguments
Usage: emptywaste.bash
```

This is what happens when emptywaste.bash is run correctly, with no initial arguments:

```
u1561634@ouranos:~/all$ emptywaste.bash
u1561634@ouranos:~/all$
```

As it can be seen above, nothing is returned when the command runs successfully. But when the directory is changed to '.waste' and 'ls' is run, it can be seen that the '.waste' directory is empty, as a result of emptywaste.bash. Below is a screenshot of the 'ls' command of the '.waste' directory before and after 'emptywaste.bash' is run:

```
u1561634@ouranos:~/.waste$ ls
testA  testB  testC
```

(BEFORE)

```
u1561634@ouranos:~/all$ cd
u1561634@ouranos:~$ cd .waste
u1561634@ouranos:~/.waste$ ls
u1561634@ouranos:~/.waste$
```

(AFTER)

listwaste.bash:

Script Screenshot:

```
#!/bin/bash
#
# listwaste.bash - A command which lists the names of all the files in the '.waste' directory and their size in bytes
# Usage - listwaste.bash
#
# Author - Waqas Musharaf
# Date - 12.02.2015

if [ $# -gt 0 ] # If there are more than 0 arguments, do the following:
then
        echo "Error: Command does not require arguments" # Informs the user what has gone wrong
        echo "Usage: listwaste.bash" # Informs the user how to use the command
        exit 1 # Exits safely, however with status 1, informing the user that something went wrong

fi # End of 'if' statement

ls -l /home/stud/u1561634/.waste | awk '{print "Filename: "$9, "## File size: " $5, "bytes"}'
# Displays the contents of the directory '.waste' in long listing format
# "awk '{print ... $9, ... $5 ...}" ensures only the name and size of each file is displayed, other information is irrelevant
# Note: awk prints an extra line with empty fields, minor cosmetic bug

exit 0 # Exits safely, with status 0, informing the user that everything went as it should
```

Testing:

listwaste.bash does not require any initial arguments, as all it does is list the contents of the '.waste' folder, including the file names and sizes in bytes. This is what happens when you attempt to run the command with (an) initial argument(s):

```
u1561634@ouranos:~/all$ listwaste.bash test
Error: Command does not require arguments
Usage: listwaste.bash
u1561634@ouranos:~/all$ listwaste.bash test hello 123
Error: Command does not require arguments
Usage: listwaste.bash
```

This is what happens when listwaste.bash is run correctly, with no initial arguments:

```
u1561634@ouranos:~/all$ listwaste.bash
Filename:  ## File size:  bytes
Filename: test ## File size: 166 bytes
Filename: test2 ## File size: 40 bytes
Filename: test3 ## File size: 50 bytes
```

As it can be seen above, the listwaste.bash command lists the contents of the '.waste' directory, including the filenames and the file sizes in bytes. There is a minor bug with the code, as there is an extra line printed above the actual files. However, it does not affect the functionality of the command.

Here is another screenshot of the listwaste.bash command, but whilst waste is empty:

```
u1561634@ouranos:~/all$ listwaste.bash
Filename:  ## File size:  bytes
```

As it can be seen above, the '.waste' is now empty. The minor extra line bug is still there but it does not show any other file to be in the '.waste' folder.

lshead.bash:

Script Screenshot:

```
#!/bin/bash
#
# lshead.bash - Lists the first few lines of a every file in a directory specified. Has options to list the first 'x' lines or last 'x' lines of the files.
# Usage - lshead.bash [directory]
#
# Author - Waqas Musharaf
# Date - 11.30.2015

if [ $# -ne 1 ] # If there is not exactly 1 argument, do the following:
then
        echo "Error: Command needs a directory argument to search for files from" # Informs the user what has gone wrong
        echo "Usage: lshead.bash [argument]" # Informs the user how to use the command
        exit 1 # Exits safely, however with status 1, informing the user that something went wrong

fi # End of 'if' statement

dir="$1" # Renames the initial argument to 'dir' (directory)

echo "Would you like to list the start or the end of each file? Enter 'S' for start or 'E' for end"
# Asks which part of the files the user would like to search
read pos # Stores the input as a variable, 'pos'

echo "How many lines of each file would you like to list? Please enter an positive integer value"
# Asks how many lines of each file the user would like to list
read val # Stores the input as a variable, 'val'

if [ $val -gt 0 ] # If the variable 'val' is greater than 0, do the following:
then
        if [ $pos == "S" ] # If the variable 'pos' is 'S', do the following:
        then
                files=$(ls $dir) # Creates a variable, 'files', that contains a list of all files in 'dir'
                for cur in $files; # For each file ('cur') in 'files', do the following:
                do
                        echo "********** START OF FILE: $cur **********" # Outputs a line of asterisks and a file label for easy viewing
                        cat $dir/$cur | head -$val; # Outputs 'val' amount of lines of the file at 'dir/cur' starting from the head of the file
                        echo "********** END OF FILE: $cur **********" # Outputs a line of asterisks and a file label for easy viewing
                        echo # Inserts a linebreak for seperation of files
                done # Once all files in 'files' have been processed, the for loop is 'done'

        exit 0 # Exits safely, with status 0, informing the user that everything went as it should

        elif [ $pos == "E" ] # If the variable 'pos' is 'E', do the following:
        then
                files=$(ls $directory) # Creates a variable, 'files', that contains a list of all files in 'dir'
                for cur in $files; # For each file, ('cur') in 'files'
                do
                        echo "********** START OF FILE: $cur **********" # Outputs a line of asterisks and a file label for easy viewing
                        cat $dir/$cur | tail -$val; # Outputs 'val' amout of lines of the file at 'dir/cur' starting from the tail of the file
                        echo "********** END OF FILE: $cur **********" # Outputs a line of asterisks and a file label for easy viewing
                        echo # Inserts a linebreak for seperation of files
                done # Once all the files in 'files' have been processed, the for loop is 'done'

        else # If neither if statement's conditions are met, do the following:
                echo "Error: Invalid input, you must enter either 'S' or 'E', according to the part of the files to list, please retry"
                # Informs the user that they inputted an invalid section of files to list
                exit 2 # Exits safely, however with status 2, informing the user that something went wrong

        fi # End of 'if' statement
```

Testing:

lshead.bash requires one initial argument, a directory from which it will display files. This is what happens when you attempt to run the command without an initial argument:

```
u1561634@ouranos:~/all$ lshead.bash
Error: Command needs a directory argument to search for files from
Usage: lshead.bash [argument]
```

This is what happens when you attempt to run the command with more than one initial argument:

```
u1561634@ouranos:~/all$ lshead.bash /home/stud/u1561634/bin $PWD
Error: Command needs a directory argument to search for files from
Usage: lshead.bash [argument]
```

If the command detects just one initial argument, it progresses to ask the user if they would like to display the start or the end of the files in the argument directory…:

```
u1561634@ouranos:~/all$ lshead.bash /home/stud/u1561634/bin
Would you like to list the start or the end of each file? Enter 'S' for start or 'E' for end
```

… and how many lines they would like to list:

```
How many lines of each file would you like to list? Please enter an positive integer value
```

However, the program only accepts ‘S’ or ‘E’ as input for position to list from, and only a positive integer number for the number of lines they would like to list. This is what happens when the user inputs an invalid input for both variables:

```
u1561634@ouranos:~/all$ lshead.bash $PWD
Would you like to list the start or the end of each file? Enter 'S' for start or 'E' for end
X
How many lines of each file would you like to list? Please enter an positive integer value
Fourty-Two
./lshead.bash: line 27: [: Fourty-Two: integer expression expected
Error: Invalid input, you must enter a positive integer value of lines to list, please retry
```

Or an invalid input for the position variable but a valid input for the number of lines variable:

```
u1561634@ouranos:~/all$ lshead.bash $PWD
Would you like to list the start or the end of each file? Enter 'S' for start or 'E' for end
X
How many lines of each file would you like to list? Please enter an positive integer value
42
Error: Invalid input, you must enter either 'S' or 'E', according to the part of the files to list, please retry
```

Or a valid input for the position variable but an invalid input for the number of lines variable:

```
u1561634@ouranos:~/all$ lshead.bash $PWD
Would you like to list the start or the end of each file? Enter 'S' for start or 'E' for end
S
How many lines of each file would you like to list? Please enter an positive integer value
Fourty-Two
./lshead.bash: line 27: [: Fourty-Two: integer expression expected
Error: Invalid input, you must enter a positive integer value of lines to list, please retry
```

This is what happens when the position and number of lines inputs are both correct: (This particular test assumes the initial directory argument is valid, more on that later):

```
u1561634@ouranos:~/all$ lshead.bash /home/stud/u1561634/bin
Would you like to list the start or the end of each file? Enter 'S' for start or 'E' for end
S
How many lines of each file would you like to list? Please enter an positive integer value
5
********** START OF FILE: chmx.bash **********
#!/bin/bash
#
# chmx.bash - A command that takes filenames as its arguments and makes those files executable to all users
# Usage - chmx.bash [argument1] [argument2] ...
#
********** END OF FILE: chmx.bash **********

********** START OF FILE: del.bash **********
#!/bin/bash
#
# del.bash - A command similar to rm which copies files to a '.waste' directory.
# Usage - del.bash [argument1] [argument2] ...
#
********** END OF FILE: del.bash **********
```

As it can be seen above, the command has worked. It has listed the first 5 lines (as inputted), from the start of the files (as inputted) for each file in the directory inputted.

(Note that this is just a snippet of the result, the command will list all the files in the directory, the screenshot just shows the first two)

Here is another a screenshot of what happens when the position and number of lines inputs are both correct, but with different inputs this time. (Again, this test assumes the initial directory argument is valid):

```
u1561634@ouranos:~/.waste$ lshead.bash $PWD
Would you like to list the start or the end of each file? Enter 'S' for start or 'E' for end
E
How many lines of each file would you like to list? Please enter an positive integer value
4
********** START OF FILE: testfile **********
IS
A
TEST
FILE
********** END OF FILE: testfile **********

********** START OF FILE: testfile2 **********
HELLO
hello
HI
********** END OF FILE: testfile2 **********
```

As of yet, it has not been shown what happens if the initial argument (the directory to search) is incorrect. It should be assumed that the command will not work with an invalid initial argument, as there is no directory to list files from. This is what happens:

```
u1561634@ouranos:~/all$ lshead.bash notadirectory
Would you like to list the start or the end of each file? Enter 'S' for start or 'E' for end
S
How many lines of each file would you like to list? Please enter an positive integer value
5
ls: cannot access notadirectory: No such file or directory
```

As it can be seen above, the command recognises the input as a single argument at first, so it continues. The command asks for the position to list from and the number of lines as usual and in this example, valid inputs are inputted for both these variables. However, when it comes to the

stage to list the files from the directory, the command recognises that the directory argument does not exist and outputs an error message as it is unable to list any files.

wastesize.bash:

Script Screenshot:

```
#!/bin/bash
#
# wastesize.bash - A command which reports the number of files in the '.waste' directory
# Usage - wastesize.bash
#
# Author - Waqas Musharaf
# Date - 12.02.2015

if [ $# -gt 0 ] # If there are more than 0 arguments, do the following:
then
        echo "Error: Command does not require arguments" # Informs the user what has gone wrong
        echo "Usage: wastesize.bash" # Informs the user how to use the command
        exit 1 # Exits safely, however with status 1, informing the user that something went wrong

fi # End of 'if' statement

echo -n "Current number of files in '.waste' is " & ls /home/stud/u1561634/.waste | wc -l
# Informs the user of the exact number of files currently in the directory '.waste'
# The exact calculation the command makes is the word count (wc) of the lines (-l) in '.waste'
# 'echo -n' is used to ensure all the output is on one line

exit 0 # Exits safely, with status 0, informing the user that everything went as it should
```

Testing:

wastesize.bash does not require any initial arguments, as all it does is show the number of files in the '.waste' folder. This is what happens when you attempt to run the command with (an) initial argument(s):

```
u1561634@ouranos:~/bin$ wastesize.bash hello
Error: Command does not require arguments
Usage: wastesize.bash
```

This is what happens when wastesize.bash is run correctly, with no initial arguments:

```
u1561634@ouranos:~/bin$ wastesize.bash
Current number of files in '.waste' is 2
```

As it can be seen above, the wastesize.bash command lists the number of files in the '.waste' directory.

Here is another screenshot of the wastesize.bash command, but whilst waste is empty:

```
u1561634@ouranos:~/bin$ emptywaste.bash
u1561634@ouranos:~/bin$ wastesize.bash
Current number of files in '.waste' is 0
```

As it can be seen above, the '.waste' is now empty. The command shows that by outputting the number of files is 0.

DigitalDay.bash Command: Pseudocode, Code and Testing:

Pseudocode Screenshot:

```
IF [ INPUT < 1 ]
THEN
        DISPLAY "Error: Command needs input"
        DISPLAY usage
        EXIT 1
END IF

DECLARE validInput
DECLARE invalidInput
DECLARE totalInput

WHILE [ INPUT > 0 ]
        IF [ currentInput = -1 ]
        THEN
                totalInput = validInput+invalidInput
                DISPLAY "Control detected"
                DISPLAY validInput
                DISPLAY invalidInput
                DISPLAY totalInput
                EXIT 0

        ELSE IF [ currentInput = -help ]
        THEN
                DISPLAY help
                DISPLAY usage
                EXIT 0

        ELSE IF [ currentInput is not positive integer ]
        THEN
                DISPLAY "Error: Input not positive integer"
                INCREMENT invalidInput

        ELSE IF [ currentInput is positive integer ]
        THEN
                mod = currentInput MOD 7

                IF [ mod = 0 ]
                THEN PRINT "Sunday"
                INCREMENT validInput
                ELSE IF [ mod = 1 ]
                THEN PRINT "Monday"
                INCREMENT validInput
                ELSE IF [ mod = 2 ]
                THEN PRINT "Tuesday"
                INCREMENT validInput
                ELSE IF [ mod = 3 ]
                THEN PRINT "Wednesday"
                INCREMENT validInput
                ELSE IF [ mod = 4 ]
                THEN PRINT "Thursday"
                INCREMENT validInput
                ELSE IF [ mod = 5 ]
                THEN PRINT "Friday"
                INCREMENT validInput
                ELSE IF [ mod = 6 ]
                THEN PRINT "Saturday"
                INCREMENT validInput
                ELSE
                PRINT "Unexpected Error"
                INCREMENT invalidInput
                END IF
        ELSE
        PRINT "Unexpected Error"
        INCREMENT invalidInput
        END IF
DONE
EXIT 0
```

Script Screenshot:

```
#!/bin/bash
#
# digitalDay.bash - A command which takes any positive numeric argument, divides each positive numeric argument by 7, and takes the remainder of each.
#                   A day corresponding to each remainder is then printed for the user. The command stops when all arguments have been checked, or when
#                   the control flag (-1) detected. If the control flag is detected, the total number of valid and invalid inputs is printed.
#
# Usage - digitalDay.bash [argument1] [argument2] ...
#
# Author - Waqas Musharaf
# Date - 12.08.2015

if [ $# -lt 1 ] # If there is less than one argument, do the following:
then
        echo "Error: The command needs at least one positive numeric argument to run successfully" # Informs the user on what went wrong
        echo "Usage: digitalDay.bash [argument1] [argument2] ..." # Informs the user on how to use the command
        echo "You may enter '-help' as an argument for a help page, or '-1' to stop the program and check the number of valid and invalid inputs"
        exit 1 # Exits safely, however with status 1, informing the user that something went wrong

fi # End of 'if' statement

valid=0 # Sets the variable 'valid' to zero
invalid=0 # Sets the variable 'invalid' to zero

while [ $# -gt 0 ] # As long as there are arguments, do the following:
do
        arg="$1" # Renames the current variable to 'arg'

        if [ $arg == "-1" ] # If 'arg' is '-1', do the following:
        then
                total=$(($valid+$invalid)) # Calculate the total number of input arguments entered
                echo "Control flag (-1) detected" # Inform the user on what happened
                echo "Total valid inputs: $valid" # Inform the user on the total number of valid arguments inputted
                echo "Total invalid inputs: $invalid" # Inform the user on the total number of invalid arguments inputted
                echo "Total inputs: $total" # Inform the user on the total number of arguments inputted
                exit 0 # Exits safely, with status 0, informing the user that everything went as it should

        elif [ $arg == "-help" ] # If 'arg' is '-help', do the following:
        then
                echo "HELP: The command needs at least one positive argument to run successfully" # Informs the user on how to use the command
                echo "Usage: digitalDay.bash [argument1] [argument2] ..."
                echo "You may enter '-1' as an argument to stop the program and check the number of valid and invalid inputs"
                exit 0 # Exits safely, with status 0, informing the user that everything went as it should

        elif ! [[ $arg =~ ^[0-9]+$ ]] # If 'arg' does not conform to these rules (is not a positive integer), do the following:
        then
                echo "Error: Invalid argument, not a positive numeric argument" # Informs the user on what went wrong
                invalid=$(($invalid+1)) # Increments 'invalid' by one

        elif [[ $arg =~ ^[0-9]+$ ]] # If 'arg' conforms to these rules (is a positive integer), do the following:
        then
                mod=$(($arg%7)) # Assign 'mod' the remainder of 'arg' divided by 7

                if [ $mod == 0 ] # If 'mod' is equal to 0, do the following:
                then    echo "Sunday" # Echo the day assigned to 0
                        valid=$(($valid+1)) # Increments 'valid' by one
                elif [ $mod == 1 ] # If 'mod' is equal to 1, do the following:
                then    echo "Monday" # Echo the day assigned to 1
                        valid=$(($valid+1)) # Increments 'valid' by one
                elif [ $mod == 2 ] # If 'mod' is equal to 2, do the following:
                then    echo "Tuesday" # Echo the day assigned to 2
                        valid=$(($valid+1)) # Increments 'valid' by one
                elif [ $mod == 3 ] # If 'mod' is equal to 3, do the following:
                then    echo "Wednesday" # Echo the day assigned to 3
                        valid=$(($valid+1)) # Increments 'valid' by one
                elif [ $mod == 4 ] # If 'mod' is equal to 4, do the following:
                then    echo "Thursday" # Echo the day assigned to 4
                        valid=$(($valid+1)) # Increments 'valid' by one
                elif [ $mod == 5 ] # If 'mod' is equal to 5, do the following:
                then    echo "Friday" # Echo the day assigned to 5
                        valid=$(($valid+1)) # Increments 'valid' by one
                elif [ $mod == 6 ] # If 'mod' is equal to 6, do the following:
                then    echo "Saturday" # Echo the day assigned to 6
                        valid=$(($valid+1)) # Increments 'valid' by one

                else # If none of the 'if' conditions are met, do the following:
                        echo "Unexpected Error: Modulo of argument cannot be achieved" # Informs the user on what has gone wrong
                        invalid=$(($invalid+1)) # Increments 'invalid' by one

                fi # End of 'if' statement

        else # If none of the 'if' conditions are met, do the following:

                echo "Unexpected Error: Argument unreadable" # Informs the user on what has gone wrong
                invalid=$(($invalid+1)) # Increments 'invalid' by one

        fi # End of 'if' statement

shift # Shifts the current argument out of the 'array' of arguments
done # When the 'while' condition is not longer valid, the while loop is 'done'
exit 0 # Exits safely, with status 0, informing the user that everything went as it should
```

(Note: The code in the above image was changed slightly, to reword a few error messages slightly. The updated code is present in the digitalDay.bash file provided and is used in the testing to follow)

Testing:

digitalDay.bash requires at least one initial argument, a positive numeric value(s). This is what happens when you attempt to run the command without an initial argument:

```
u1561634@ouranos:~/all$ digitalDay.bash
Error: The command needs at least one positive numeric argument to run successfully
Usage: digitalDay.bash [argument1] [argument2] ...
You may enter '-help' as an argument for a help page, or '-1' to stop the program and check the number of valid and
invalid inputs
```

As it can be seen in the screenshot above, the command requires at least one positive numeric input to run successfully. This is what happens when the initial variable is a single non-numeric input:

```
u1561634@ouranos:~/all$ digitalDay.bash s
Error: Invalid argument, not a positive numeric argument
```

Or a single decimal input:

```
u1561634@ouranos:~/bin$ digitalDay.bash 1.5
Error: Invalid argument, not a positive numeric integer argument
```

Or a single negative input:

```
u1561634@ouranos:~/bin$ digitalDay.bash -2
Error: Invalid argument, not a positive numeric integer argument
```

Or multiple invalid inputs:

```
u1561634@ouranos:~/bin$ digitalDay.bash 4.5 -32 -9.2
Error: Invalid argument, not a positive numeric integer argument
Error: Invalid argument, not a positive numeric integer argument
Error: Invalid argument, not a positive numeric integer argument
```

Another example of multiple invalid inputs:

```
u1561634@ouranos:~/bin$ digitalDay.bash -123456789.4321 20a test
Error: Invalid argument, not a positive numeric integer argument
Error: Invalid argument, not a positive numeric integer argument
Error: Invalid argument, not a positive numeric integer argument
```

This is what happens when you run the command with a single valid initial argument:

```
u1561634@ouranos:~/bin$ digitalDay.bash 5
Friday
```

As it can be seen above, the command has worked. The input was a positive numeric argument and the day of the week, corresponding to modulo 7 of the input was displayed.

This is another example of a single valid initial argument:

```
u1561634@ouranos:~/bin$ digitalDay.bash 23
Tuesday
```

This is what happens when you run the command with multiple valid arguments:

```
u1561634@ouranos:~/bin$ digitalDay.bash 15 3 0 74 120 6
Monday
Wednesday
Sunday
Thursday
Monday
Saturday
```

Another example of what happens when the command is run with multiple valid arguments:

```
u1561634@ouranos:~/bin$ digitalDay.bash 42 94 1 1672326463
Sunday
Wednesday
Monday
Wednesday
```

Here is what happens when the command is run with a mixture of valid and invalid arguments:

```
u1561634@ouranos:~/bin$ digitalDay.bash 123 hello 45 89 02 bye
Thursday
Error: Invalid argument, not a positive numeric integer argument
Wednesday
Friday
Tuesday
Error: Invalid argument, not a positive numeric integer argument
```

Another example of what happens when the command is run with a mixture of valid and invalid arguments:

```
u1561634@ouranos:~/bin$ digitalDay.bash qwe 535 end 38
Error: Invalid argument, not a positive numeric integer argument
Wednesday
Error: Invalid argument, not a positive numeric integer argument
Wednesday
```

The command also has a few special initial variables that can be used. One of these is '–help', which shows a help statement, as follows:

```
u1561634@ouranos:~/bin$ digitalDay.bash -help
HELP: The command needs at least one positive numeric integer argument to run successfully
Usage: digitalDay.bash [argument1] [argument2] ...
You may enter '-1' as an argument to stop the program and check the number of valid and invalid inputs
```

Another one is '-1', which, when it is reached in the queue of initial variables, will halt the command (and any other variables) and display the number of valid inputs, invalid inputs and total inputs inputted in that instance of the command running. This is displayed below:

Using -1 as the sole variable:

```
u1561634@ouranos:~/bin$ digitalDay.bash -1
Control flag (-1) detected
Total valid inputs: 0
Total invalid inputs: 0
Total inputs: 0
```

Using -1 as the end variable (after a mixture of previous variables):

```
u1561634@ouranos:~/bin$ digitalDay.bash 213 f 324 djsf 234 dsh k 2 -1
Wednesday
Error: Invalid argument, not a positive numeric integer argument
Tuesday
Error: Invalid argument, not a positive numeric integer argument
Wednesday
Error: Invalid argument, not a positive numeric integer argument
Error: Invalid argument, not a positive numeric integer argument
Tuesday
Control flag (-1) detected
Total valid inputs: 4
Total invalid inputs: 4
Total inputs: 8
```

Another example of using -1 as the end variable:

```
u1561634@ouranos:~/bin$ digitalDay.bash 2 3 5 -1
Tuesday
Wednesday
Friday
Control flag (-1) detected
Total valid inputs: 3
Total invalid inputs: 0
Total inputs: 3
```

Using -1 when there are variables still remaining to be executed:

```
u1561634@ouranos:~/bin$ digitalDay.bash hello 2 4 5 -1 26 bye 8 ee
Error: Invalid argument, not a positive numeric integer argument
Tuesday
Thursday
Friday
Control flag (-1) detected
Total valid inputs: 3
Total invalid inputs: 1
Total inputs: 4
```

Another example of using -1 when there are variables still remaining to be executed:

```
u1561634@ouranos:~/bin$ digitalDay.bash -1 783 4 3 hi 82 902 var
Control flag (-1) detected
Total valid inputs: 0
Total invalid inputs: 0
Total inputs: 0
```

END OF DOCUMENT