UNIX SHELL PROGRAMMING
STANDARD INPUT/OUTPUT/ERROR AND I/O REDIRECTIONS

0 = standard input - where you read from when you use "scanf" or "gets" in C, "cin" in C++ or "input" in Python.
1 = standard output - where you write when you use "printf" in C, "cout" in C++, or "print" in Python.
2 = standard error - similar to the standard output, but conventionally used to display errors, in order to avoid mixing results with errors.

When you open a file in a program, you get back some kind of variable that allows you to operate on the file (FILE* from fopen). Whenever you start a program, it will have three files already open : 0, 1 and 2. The program treats the command line like a file: it writes to it and it reads from it.

I/O redirections

a. What if I want the output of a command to be stored in a file?
   ls -l /etc **>** output.txt
b. What if I want to add the output of another command to the same file?
   ps -ef **>>** output.txt
c. What if I want the standard output of a command to be sent to the tandard input of another command?
   ls **|** sort
d. What if I want the standard input to be taken from a file?
   sort **<** a.txt
e. Redirect the errors of a command to a file
   rm fileThatDoesNotExist 2 > output.err
f. Redirect both the standard and error output in the same file
   rm fileThatDoesNotExist > output.all 2>&1
   redirect standard output to output.all, and the error output to the same place where the standard output goes

/dev/null is the file that contains nothing, and everything that you write to it, disappears. Used mainly to hide program output (either standard or error)

COMMAND TRUTH VALUES

1. The truth value of a command execution is determined by its exit code. The rule is the opposite of the C convention. With **0** being **true** and anything else being **false**. The exit code is not the output of the command.
2. There are two commands **true** and **false** that simply return **0** or **1**.
3. Command **test** evaluate a expression and return it's status code.

Commands can be chained using logical operators **&&** and **||**.

a. true || echo This should not be displayed
b. false || echo This should be displayed
c. true && echo This should be displayed
d. false && echo This should not be displayed
e. grep -E -q "=" /etc/passwd || echo There are no equal signs in file /etc/passwd
f. test -f /etc/abc || echo File /etc/abc does not exist
g. test 1 -eq 2 || echo Not equal
h. test "asdf" == "qwer" || echo Not equal
i. ! test -z "abc" || echo Empty string

Test command conditional operators

a. String : **==**, **!=**, **-n**, **-z**

b. Integers : **-lt**, **-le**, **-eq**, **-ne**, **-ge**, **-gt**
c. File system : **-f**, **-d**, **-r**, **-w**, **-x**

SHELL VARIABLES AND EMBEDDED COMMANDS

1. Defined as A="Tom" or B=5
2. Embedded commands
   a. Delimited by ` (back-quote)
   b. Are replaced by the output of the command
   c. Store a command output in a variable N=`grep -E "/gr211/" /etc/passwd | wc -l`
3. Referred as **$A** or **${A}**
   a. echo $A is a human
   b. echo $Acat is a feline - doesn't work
   c. echo${A}cat is a feline
4. When used in strings delimited by **"**, variables and embedded commands will be replaced by their value. Strings delimited by **'** do not allow any substitutions in their content.
   a. echo "AA is a GPS navigator"
   b. echo "There are `grep "/gr211/" /etc/passwd | wc -l` students in group 211"

SHELL SCRIPTS

1. Any text file with exection permissions can be a script if it contains commands interpretable by the current shell.
2. Comments start with **#**.
3. Example :
   a. Create file a.sh with the content :

```
echo Hello World
```

   b. Give the script execution permissions using **chmod 700 a.sh**
   c. Execute the script using ./a.sh
4. Permissions
   a. Run **ls -l** and see the first 10 characters on each line
      i. The first character tells the file type: **-** is a regular file, **d** is a directory
      ii. Characters 2,3,4 shows the permissions for the owner of the file
      iii. Characters 5,6,7 shows the permissions for the group of the file
      iv. Characters 8,9,10 shows the permissions for everybody else
5. Creating a file with shell specification
   a. Create file a.sh with the content :

```
#!/bin/bash

echo Hello World
```

      The #! specifies that the script should be interpreted by Bash
   b. Give the script execution perissions using **chmod 700 a.sh**
   c. Execute the script using ./a.sh
6. Special variables
   a. **$0** - The name of the command
   b. **$1** - **$9** - Command line arguments
   c. **$*** or **$@** - All the arguments together
   d. **$#** - Number of command line arguments
   e. **$?** - Exit code of the previous command

```
#!/bin/bash

echo Command: $0
echo First four args: $1 $2 $3 $4
echo All args: $@
echo Arg count: $#

true
echo Command true exited with code $?

false
echo Command false exited with code $?
```

UNIX SHELL FOR LOOP

1. Similar to the Python **foreach**
2. Basic example, showing **do** on the same line or on the next line.

```
#!/bin/bash

for A in a b c d; do
    echo Here is $A
done

for A in a b c d
do
    echo Here is $A
done
```

3. Iterating over the command line arguments.

```
#!/bin/bash

for A in $@; do
    echo Arg A: $A
done

for A; do
    echo Arg B: $A
done
```

The list of values through which for iterates can be explicitly as above or throught wildcards or embedded commands.

Filename wildcards (similar to regular expressions) :

Rules :

i. ***** - matches any sequence of characters, but not the first dot in a filename
ii. **?** - matches any single character, but not the first dot in a filename
iii. **[abc]** - matches any single character from the list
iv. **[!abc]** - matches any single character that is not from the list

Example with wildcard : Count all the lines of code in the C files in the directory given as command line argument, excluding lines that are empty or contain only spaces

```
#!/bin/bash

S=0
for F in $1/*.c; do
    N=`grep -E "[^ ]" $F | wc -l`
    S=`expr $S + $N`
done
echo $S
```

Example with embedded command : Count all the lines in the C files in the directory given as command line argument and its subdirectories, excluding lines that are empty or contain only spaces

```
#!/bin/bash

S=0
for F in `find $1 -type f -name "*.c"`; do
    N=`grep -E "[^ ]" $F | wc -l`
    S=`expr $S + $N`
done
echo $S
```

Filenames that contain spaces will cause problems here as well, we can solve this problem with find ... | **while read F**.

UNIX SHELL IF/ELIF/ELSE/FI STATEMENT

In Bash, every command returns an **exit status** (0 for success, non-zero for failure), which can be used as a condition in if statements. You can also combine commands using logical operators like &&, || and !.

Basic example which checks each argument and announces whether it is a file, or a directory, or a number, otherwise it states that it does not know what it is :

```
#!/bin/bash

for A in $@; do
    if test -f $A; then
        echo $A is a file
    elif test -d $A
    then
        echo $A is a dir
    elif echo $A | grep -E -q "^[0-9]+$"; then
        echo $A is a natural number
    else
        echo We do not know what $A is
    fi
done
```

To make the condition look a bit more natural, there is a second syntax, in which **[** is an alias of command **test** and **]** marks the end of the command **test**. Leave spaces around these brackets or there will be syntax errors.

The basic IF example above can be re-written as follows :

```
#!/bin/bash

for A in $@; do
    if [ -f $A ]; then
        echo $A is a file
    elif [ -d $A ]
    then
        echo $A is a dir
    elif echo $A | grep -E -q "^[0-9]+$"; then
        echo $A is a number
    else
        echo We do not know what $A is
    fi
done
```

UNIX SHELL WHILE STATEMENT

Basic example which reads the user input until input is stop :

```
#!/bin/bash

while true; do
    read X
    if test "$X" == "stop"; then
        break
    fi
done
```

More complex example which read the console input until the user provides a filename that exists and can be read :

```
#!/bin/bash

F=""
while [ -z "$F" ] || [ ! -f "$F" ] || [ ! -r "$F" ]; do
    read -p "Provide an existing and readable file path:" F
done
```