LPIC-1 102-500 – Lesson 2

105.2 Customize or write simple scripts

Creating scripts

- **$ cat > many.sh << EOF** # create a new script
  # file.
  cd \$1
  ls -la
  pwd
  EOF
- **$ source many.sh /etc** # use the **many.sh** file as command source.
- **$. many.sh /etc** # identical command to the above. The keyword **source** is replaced with "."
- **$ bash many.sh /etc** # the output of this command is the same as above but the commands are executed in a child shell.

Creating executable scripts

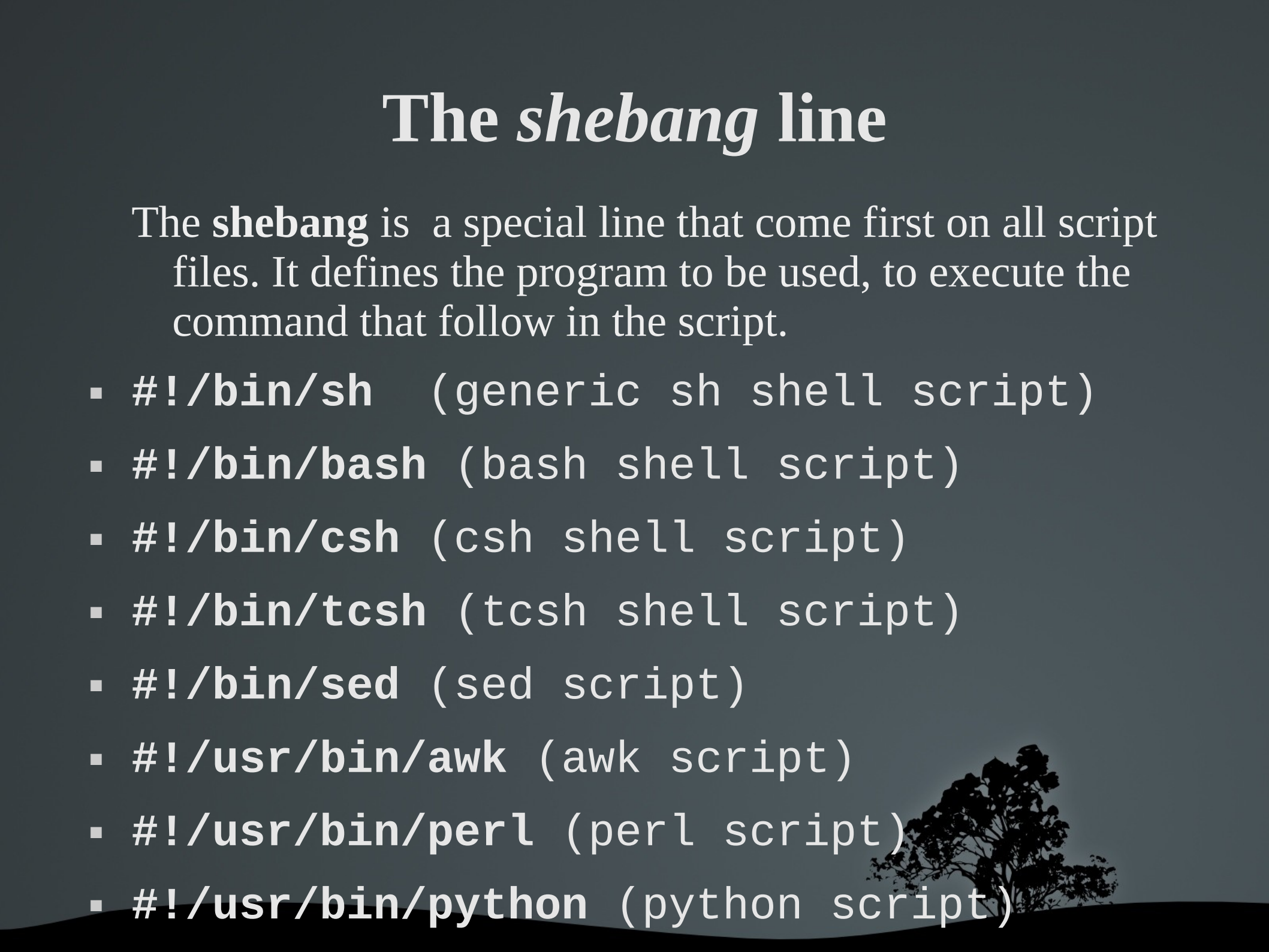
- **$ chmod a+x many.sh** # convert the **many.sh** script into an executable file.
- **$ ls -l many.sh** # verify.
  -rw**x**r-**x**r-**x** 1 theo theo 17 2011-12-18 10:24 many.sh
- **$ ./many.sh** # since **many.sh** is not included in **$PATH** it will have to be called explicitly with "./" or using the absolute path, **$HOME/many.sh** or **~/many.sh**. If it is simply invoked with its name on the working directory nothing is executed and an error is issued.

***Note**: applying **SUID** or **SGID** in scripts has no effect. This is a security measure.*

The *shebang* line

The **shebang** is a special line that come first on all script files. It defines the program to be used, to execute the command that follow in the script.

- **#!/bin/sh** (generic sh shell script)
- **#!/bin/bash** (bash shell script)
- **#!/bin/csh** (csh shell script)
- **#!/bin/tcsh** (tcsh shell script)
- **#!/bin/sed** (sed script)
- **#!/usr/bin/awk** (awk script)
- **#!/usr/bin/perl** (perl script)
- **#!/usr/bin/python** (python script)



Apply *shebang* in shell script

- **$ cat > many.sh << EOF** # create a shell script.
  #!/bin/sh
  # the **#!/bin/bash** shebang is another
  # possible option.
  cd \$1
  ls -la
  pwd
  EOF
- **$ chmod a+x many.sh**
- **$ ./many.sh /etc**

Command Substitution

- For command substitution we use the bash operators “``” or “**$()**”. The enclosed commands are executed in a child shell.
- `$ KERNEL_VER=`uname -r` #  the output of uname -r is passed as the value of the KERNEL_VER variable.`
- `$ grep -i linux $(find /usr/share/doc -name "*.txt") # the results of the find /usr/share/doc -name "*.txt" command are used as files to be searched by grep.`

Sending email from shell

- **$ echo "Universe Collapsed\!" | mail -s "Universe failed" root**
  # send email message to the root user with subject **"Universe failed"** and body **"Universe Collapse\!"**.
- **$ cat /var/log/messages | mail -s "Logs" user@example.com**
  # send email message to user **user@example.com** with subject **"Logs"** and body the contents of **/var/log/messages**.
- **$ mail -s "File systems" user@example.com < /etc/fstab** # send email message to **user@example.com** with subject **"File systems"** and get content from **/etc/fstab**.
- **$ mail -s "Test mail" root@server.int << EOF** # another example # using **"<<"**.

  > This is a test
  > We are the best
  > EOF

Input data with `read`

- **$ vi user.sh** # press "i" for insert mode.

```
#!/bin/bash
echo "User Name: "
read USERN
echo "Shell: "
read SHELLU
echo "User Name = $USERN, Shell = SHELLU"
exit 0
```

- **$ chmod +x user.sh** # make script, executable.
- **$ ./user.sh** # invoke script.

Check exit status with `test` or `[`

- The **test** or **[** commands are **bash** builtins but also executable files in **$PATH**.
- **$ test -e /etc/fstab** # check if file exists. exit status is "**0**" if it exists and "**1**" if absent.
- **$ [-e /etc/fstab]** # this command is identical to the command above. The bracket "**[**" is just another name for **test** with the only difference that it has to terminated with "**]**".
  Both brackets must be separated from the rest of text by space!
- **$ test -x /bin/ls** # check if file exists and it is executable.
- **$ [-s ~/.bash_profile]** # check if file exists and is not empty.

Check exit status with `test` or `[`

- **$ test "$HISTSIZE" -eq 1000** # check if the **HISTSIZE** variable equals to 1000. It is recommended for variables to be enclosed in double quotes: "".
- **$ ["$EDITOR"]** # = **[-n "$EDITOR"]**. check if the **$EDITOR** variable is set.
- **$ [-x /bin/ip -o -x /sbin/ip]** # logical OR. Check if files **/bin/ip** or **/sbin/ip** exist and they are executable.
- **$ ["$CONT" = "yes" -a -f /usr/lib/libtest.so]** # logical AND. Check if **CONT** exists and equals to**"yes"** and the regular **/usr/lib/libtest.so** file exists.

Options of `test` or `[`

- **-e file** # check if **file** exist
- **-f file** # check if **file** exist as normal file
- **-d dir** # check it the **dir** directory exists
- **-L file** # check if the symlink **file** exists
- **-r file** # check if **file** exists and is readable
- **-w file** # check if **file** exists and is writeable
- **-x file** # check if **file** exists and is executable
- **-s file** # check if **file** exists and is not empty
- **file1 -ot file2** # check if **file1** is older than **file2**
- **file1 -nt file2** # check if **file1** is newer than **file2**

Options of `test` or `[`

- **-n string** # check if the length of **string** is non-zero
- **-z string** # check if the length of **string** equals zero
- **string1 = string2** # check if the two strings are identical
- **string1 != string2** # check if the two strings are different
- **arg1 -eq arg2** # check if **arg1** is arithmetically equal to **arg2**
- **arg1 -ne arg2** # check if **arg1** is arithmetically not equal with **arg2**
- **arg1 -lt arg2** # check if **arg1** is less than **arg2**
- **arg1 -le arg2** # check if **arg1** is less or equal to **arg2**
- **arg1 -gt arg2** # check if **arg1** is greater than **arg2**
- **arg1 -ge arg2** # check if **arg1** is greater or equal to **arg2**

Options of `test` or `[`

- `! expr # check if expression expr is false`
- `expr1 -a expr2 # logical AND between expr1 and expr2`
- `expr1 -o expr2 # logical OR between expr1 and expr2`

*Note: for more information look at the **test** documentation with:*

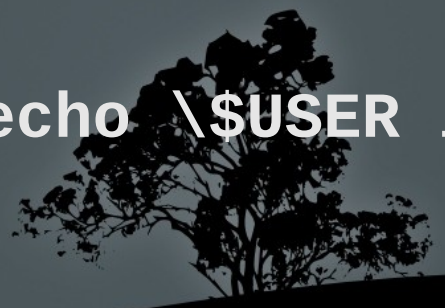
`info coreutils 'test invocation'`

Conditionals with `if`

- The **if** builtin is used for executing commands, conditionally.

- ```
  $ if [ -z "$USER" ] # = if test -z "$USER"
    then
       echo \$USER is not defined!
       exit 1
    elif [ "$USER" = root ]
       then
       echo 'Warning\! You are root!'
    else
       echo "\$USER is $USER"
    fi
  ```

- ```
  $ if [ "$USER" = user ] ; then echo \$USER is
    user ; fi
  ```



Conditionals with `if`

- The **if** command can be combined with any other command like,for example, **grep**. It can be used interactively from the shell or used in a script.
-
```
$ if grep tobedeleted /tmp/dummy.file
> then
> rm -f /tmp/dummy.file
> elif [ "$?" = 1 ]
> then
> echo "dummy.file is not to be
deleted\!"
> else
> echo "Error in grep"
> fi
```

Print sequences with `seq`

- **$ seq 1 10** # prints all numbers from 1 to 10 in separate lined each.
- **$ seq 1 2 10** # prints all numbers from 1 to 10 in steps of **2** (1, 3, 5, 7, 9)
- **$ seq 2 2 10 | xargs** # prints 2, 4, 6, 8, 10. The output will be in a single line because is piped to **xargs**.
- **$ seq 5 5 105** # five, ten,fifteen, …, 100, 105.

Create loops with `for`

- In its basic form, **for** sets a variable (**PET** in this case) which takes values from a list (**dog cat iguana turtle**).
- ```
  $ for PET in dog cat iguana turtle
  do
    echo "Pet is $PET"
  done
  ```


  ```

Create loops with `for`

- ```
  for FILE in `ls /etc` # use of /etc contents
                        # as a list.
  do
     echo "File is $FILE"
  done
  ```
- ```
  SUM=0
  for I in $(seq 1 30) # = for I in {1..30}, for
                       # ((I = 1 ; I <= 30 ; I++))
  do
     SUM=`expr $I + $SUM`
     if [ "$I" -eq 30 ]
        then echo "Sum is $SUM"
     fi
  done
  ```

Create loops with `for`

- **$ for FILE in *** # select all files/directories in
  # the working directory.

```
do
   echo "$FILE is in the current directory"
done
```

- **$ for FILE in *.txt** # select all ***.txt** files in
  # the working directory.

```
do
   echo "$FILE is a text file, in pwd"
done
```

Create loops with `while`

- The **while** builtin is used to check a condition at the beginning of the loop. The iterations persist until the condition is false.
- ```
  VAR=0
  LIMIT=30
  while [ "$VAR" -lt "$LIMIT" ]
  do
     echo "\$VAR = $VAR"
     VAR=`expr $VAR + 1`
  done
  ```


  ```

Create loops with `while`

```
while [ "$VAR" != "end" ] # this loop will
                          # accept values and
                          # print them until
                          # someone enters
                          # "end".
do
   echo "Input VAR: (end to exit) "
   read VAR
   echo "\$VAR = $VAR"
done
```

Create loops with `until`

- The **until** builtin is used, contrary to **while**, to check if a condition at the beginning of the loop is false and iterations persist until the condition is true.
- **until ["$VAR" = "end"]** `# this loop will`
  `# accept values and`
  `# print them until`
  `# someone enters`
  `# "end".`

```
do
    echo "Input VAR: (end to exit) "
    read VAR
    echo "\$VAR = $VAR"
done
```

License