Shells and Shell scripting

What is a Script?

- A **shell script** is a list of commands in a computer program that is run by the Unix shell which is a command line interpreter.
- The various dialects of **shell scripts** are considered to be **scripting** languages. Typical **operations** performed by **shell scripts** include file manipulation, program execution, and printing text.
- Shell scripts have several required constructs that tell the shell environment what to do and when to do it.
- The shell is a real programming language, complete with variables, control structures, and so forth. No matter how complicated a script gets, it is still just a list of commands executed sequentially.
- Virtually any command which can be executed at the unix prompt can be executed within a shell script.

Shell script structure

- **Structure**
  - Each line of the script file is a single command. There are two exceptions to this statement. The first is when a line ends with a backslash "\\", since the backslash removes the significance of the newline character as a command terminator. Therefore, a backslash at the end of a line is the indication of command continuation on the following line, just as it is at the prompt.
  - The second exception is when the "<<" I/O redirection symbol is used in a script file.
- **Comments**
  - Comments in C shell scripts begin at a pound sign "#" and continue to the end of the line. Since most Unix system support several shells, it is recommended that each shell script start with a comment indicating which shell the script was written for.
- **Output**
  - Text can be sent to standard output from within a shell script by using the c-shell command 'echo'. This command will print the value of its arguments on standard output appending a newline. The -n option to echo will suppress the newline at the end of the text.

Example - Create script

SOFTENGER
Strive to Excel

- The following script will print out the name and contents of the current working directory.

```
# print the name and contents of the current
# working directory
echo "This is from a shell script:"
pwd  # print the directory name
ls  # print the directory contents
```

- The following script will edit a C source code file named "count-c", compile and link it producing an executable version named "count", and finally run count.

```
# edit compile link, and run
vi count.c
cc count.c -o count
count
```

Running script

- **Direct Execution**
  - The simplest way to execute a shell script is to used chmod to make the text file executable. The script can then be executed by typing its name if it is in the current working directory or is in the path for that shell.
  - For example if the script named prg.sh is in the directory /home/trainee/samplescripts
  - Then the command sequence should be

```
unix> cd /home/trainee/samplescripts
unix> chmod +x ptg1.sh
unix> pwd
/usr/sample/script
```

Redirection of Standard output/input

- Put output of one command in to file. There are three main redirection symbols >,>>,<
  - > Redirector Symbol
    - Syntax: Linux-command > filename
    - To output Linux-commands result to file. Note that If file already exist, it will be overwritten else new file is created.
    - For e.g. To send output of ls command give **$ ls > myfiles**
    - Now if 'myfiles' file exist in your current directory it will be overwritten without any type of warning.
  - >> Redirector Symbol
    - Syntax: Linux-command >> filename
    - To output Linux-commands result to END of file. Note that If file exist , it will be opened and new information / data will be written to END of file, without losing previous information/data, And if file is not exist, then new file is created.
    - For e.g. To send output of date command to already exist file give $ date >> myfiles
  - < Redirector Symbol
    - Syntax: Linux-command < filename
    - To take input to Linux-command from file instead of key-board. For e.g. To take input for cat command give
    - $ cat < myfiles

Internal command and Variables

- The command echo is an internal command, it is executed by the shell itself. Its meaning is “show on the screen the literal which follows” (similar to printf()).
- What is the literal “$?” ?
  - Each literal starting with **$ represents an environment variable, i.e. a variable belonging to the process, which can be** read and written.
- **$?** is a special shell variable which expands to “the exit status of the last command executed”.
- Variables are symbolic names that represent values stored in memory
  - Two types of variables:
    - Environment variables hold information about your login session
    - Shell variables are created at the command prompt or in shell scripts and are used to temporarily store information

Shell Variables

- Shell Variables are variables that you can define and manipulate for use with program commands in a shell
- Observe basic guidelines for handling and naming shell variables
  - I recommend that you use all UPPERCASE characters when naming your variables
- Variables are handled differently depending on the syntax Type:
  - echo $USERNAME
  - echo “$USERNAME”
  - echo ’$USERNAME’
  - echo `$USERNAME`

Shell Operators

- Bash shell operators are in four groups:
  - Defining operators
  - Evaluating operators
  - Arithmetic operators
  - Redirection operators
  - Relational operators

Defining Operators

- Used to assign a value to a variable
- Most common is = (equal sign)
- Use quotation marks with strings
- Back-quote says execute the command inside the back-quotes and store the result in the variable

Evaluating Operators

- Used for determining the contents of a variable
- **echo $variablename** will show the value of variablename
- Double quotes can be used, but not single quotes

Arithmetic Operators

SOFTENGER
Strive to Excel

- Examples of Shell Arithmetic Operators:

Operator	Operation
+, -, *, /	addition, subtraction, multiply, divide
var++	Increase the variable var by 1
var--	Decrease the variable var by 1
%	Modulus (Return the remainder after division)

- Regular mathematical precedence rules apply to arithmetic operators

let is a built-in function of Bash that allows us to do simple arithmetic. It follows the basic format:

```
#!/bin/bash
# Basic arithmetic using let
let a=5+4
echo $a # 9
```

expr is similar to let except instead of saving the result to a variable it instead prints the answer. Unlike **let** you don't need to enclose the expression in quotes. You also must have spaces between the items of the expression. It is also common to use **expr** within command substitution to save the output to a variable.

```
#!/bin/bash
expr 5 + 4
expr "5 + 4"
expr 5+4
```

Relational Operators

- These operators do not work for string values unless their value is numeric. all the conditional expressions should be placed inside square braces with spaces around them. For example, **[$a <= $b]** is correct whereas, **[$a <= $b]** is incorrect.

Operator	Description	Example
-eq	Checks if the value of two operands are equal or not; if yes, then the condition becomes true.	[$a -eq $b] is not true.
-ne	Checks if the value of two operands are equal or not; if values are not equal, then the condition becomes true.	[$a -ne $b] is true.
-gt	Checks if the value of left operand is greater than the value of right operand; if yes, then the condition becomes true.	[$a -gt $b] is not true.
-lt	Checks if the value of left operand is less than the value of right operand; if yes, then the condition becomes true.	[$a -lt $b] is true.
-ge	Checks if the value of left operand is greater than or equal to the value of right operand; if yes, then the condition becomes true.	[$a -ge $b] is not true.
-le	Checks if the value of left operand is less than or equal to the value of right operand; if yes, then the condition becomes true.	[$a -le $b] is true.

Stdin, stdout, stderr and Redirection Operators

SOFTENGER
Strive to Excel

- Each process is characterized by three files which are automatically open when the process is started:
  - STDIN: standard input, the file for data input. Usually the keyboard.
  - STDOUT: standard output, the file for data output. Usually the screen.
  - STDERR: standard error, the file for error messages output. Usually the screen.
- They are associated respectively to file descriptors “0”, “1” and “2” of the process and are propagated from the parent process to the child.

Redirection Operators

SOFTENGER
Strive to Excel

- stdin, stdout and stderr can be **redirected to other files, in a** shell command, using the characters **<, >** and **>>**.
  - For e.g. To send output of ls command give **$ ls > myfiles**
  - To send output of date command to already exist file give **$ date >> myfiles**
  - To take input for cat command give **$ cat < myfiles**
- If you want to suppress the output (or the error messages) of a command, you can perform redirection to the special file
  - **/dev/null e.g. $ ls /> //dev/null**

Running Script

- **Non-script Execution**
  - There are several ways to execute commands within shells or at the command prompt.
  - **Sequential Commands**
    - A series of independent commands can be grouped together as a single command by separating them by semicolons ";".
    - Example : pwd; ls; date; who
  - **Pipes**
    - Pipes "|" also cause the execution of multiple processes from a single command line. With a pipe, the standard output of the process before the pipe is redirected to the standard input of the process after the pipe. This is also covered below, in I/O Redirection.
    - unix> /home/trainee/samplescripts |ls –l|wc –l

Grave Accents

- Grave accents "`" (on the same key as the tilde "~") will force inline execution of the enclosed command. Like the quotation marks """ and the apostrophes "'", grave accents must be used in pairs. Grave accents are used where a command "is not expected".

```
unix> currentDir=`pwd`
```

- It is getting the current directory using the **pwd** command, the simplest usage. Just execute the command and use the output to set the variable.

```
Unix>lineCount=`wc -l $fileName | cut -c1-8`
```

- This is getting the count of lines in a file. Use the **wc** command (word count) with the **-l** flag (lines) then *pipe* this to the **cut** command and use the **-c** flag (characters) to cut out the 1st to 8th character (the *number*) from the output. Use this to set the variable. The **cut** is required because **wc** insists on including the *filename* in the output string. Also beware of the *number* part, as it is not an *integer* but a *string of numbers*. If you want a number, then use the **expr** command to add a zero to the result which has the effect of converting the type. Although I said earlier that shell variables are un-typed, the **test** command gets upset when presented with a string and is then told to evaluate it as greater or less-than another number. The variables maybe typeless, however their contents are typed unless converted.

Grave Accents

Unix>**thirdPath=`echo $PATH | cut -f3-3 -d:`**

- Next is getting the third string from a complex string list. The PATH *environment variable* (See - Startup & Environment) is a list of search paths for the UNIX system to use when searching for command locations within the directory structure. Each path is separated from its neighbor by a colon thus: [/bin:/usr/bin:/usr/lib:/usr/etc]. This command *pipeline* echoes the path string into **cut** using the **-f** flag (field) to select the third field. The **-d** flag specifies the delimiter character for the fields as a *colon* in this case. Once again the result of the executed pipeline is used to set a variable.

Unix>**lastWord=`head -$lineCount $fileName | tail -1 | tr '''/' | basename`**

- Lastly finding the last word on a particular line. Use the **head** command with the *line_count* variable to echo out the top *line_count* lines of a file. Pipe this through the **tail** command and only pass the last one line. Having got the line you want, pipe this through **tr** (transform) and change each *blank* into a *forward-slash* (a UNIX directory separator). Then pipe this through **basename** which always strips away the full path leaving just the filename. This leaves you with just the last word, however long the line was, and however many words there were on the line, which can then be used to set our variable.

Grave Accent Exercise

- accept filename, print size of file, inode number of the file.

```
#!/bin/bash
echo “enter file name to print size
   and inode number: “
read fname
fsize=`ls –l $fname |awk ‘{print
   $5}’`
Finode=`ls –i $fname`
echo “size of $fname is $fsize and
   inode of $fname is $finode”
```

accept directory name, and print name of file with the largest size

```
#!/bin/bash
      echo “enter dir name to print name of
      the file with the largest size: “
      read dname
      fsize=`ls –lSr $dname | tail -1`
      echo “largest size file in $dname is
      $fsize”
```

conditional expression

SOFTENGER
Strive to Excel

- The double ampersand “**&&” is indeed a conditional expression** which includes also the double pipe “||” to indicate the “else” part.
- The complete syntax is: **command && then-part || else-part**
- The double ampersand “**&&” can be used to separate different** commands typed in a single line, but a command is executed **only if the previous one succeeded, i.e. its exit status is zero.**
- This is an initial form of **“if-then” conditional expression**
- The && serves as a logical AND (requiring all conditions to be true) operation, while the || provides a logical OR (requiring only one to be true).

conditional expression

- This particular script runs through the tests twice, but only to demonstrate the two “flavors” of the brackets that can be used. Note that && doesn’t work inside square brackets unless they’re doubled.

```
#!/bin/bash
echo $1 $2
 if
   [ $1 == "a" ] && [ $2 == "b" ] then
   echo c
fi
if
   [[ $1 == "a" && $2 == "b" ]] then
   echo c
fi
```

This script will output a "c" twice if the first argument is "a" or if the second is "b". It will do the same thing if the first two arguments are "a" and "b".

conditional expression

- The double brackets allow you to put the && and || between the tests themselves. This is a newer syntax and may not be supported by every shell.

```
#!/bin/bash
echo $1 $2
if [ $1 == "a" ] || [ $2 == "b" ] then
   echo c
fi
if [[ $1 == "a" || $2 == "b" ]] then
   echo c
fi
```

Quoting and escaping

- Whatever is between single quotes remains unchanged by the shell.
- If you enclose something in double quotes then concatenated blanks are preserved (as in the literal case with the single quotes) but variables are substituted by their values and filenames are expanded from their wild-cards to full filenames and or paths.

Example quoting

```
my_name=“Ram Sharma" # Set a variable
echo "$my_name" # Will output – Ram Sharma
echo '$my_name' # Will output - $my_name
```

Scripting and Magic Numbers

```
#!/bin/sh
ls . | grep Shell
```

- What is the meaning of the first line?
  - In general, the first two (or four) bytes of a file make the so-called **magic number, which is used to univocally identify the file type.**
  - The exec system function reads the magic number and, if it knows the type, executes the file accordingly.
  - 7F 45 4C 46, a ELF (executable and linkable format) file
  - CA FE BA BE, a Java class file
  - ’M’ ’Z’, a Windows EXE file
  - ’#’, ’!’, an interpreted file, the pathname which follows is the interpreter

Scripting and Arguments

- Script to search for the word not in the current dir but in a given dir

```
#!/bin/sh
ls $1 | grep Shell
```

- **$1, $2, $3, ... are special variables which expand to the relevant positional parameter given in the command line.**
- **$0 is the name of the command itself.**
- They play the same role of **argv of a C program.**
- The variable **$# expands to the number of parameters.**

The “if” conditional expression

```
if command then
   # do something
fi
else
   ... else part
Fi
```

OR

```
if [ something ]; then
   echo "Something"
elif [ something_else ]; then
echo "Something else"
else echo "None of the above"
fi
```

Note that fi is if backwards!. Also, be aware of the syntax - the "if [...]" and the "then" commands must be on different lines. Alternatively, the semicolon ";" can separate them

The “if” conditional expression

- Example

```
#!/bin/sh
a=10 b=20
if [ $a == $b ] then
        echo "a is equal to b"
else
        echo "a is not equal to b"
fi
```

Upon execution, you will receive the following result –

```
a is not equal to b
```

Iteration: the “for” construct

- The shell provides an internal command to perform iterations
- Iteration is always performed through a list of space-separated literals
- As in similar constructs, an iteration variable must be specified.
- The syntax of **for is:**

```
for iteration-variable in list of literals
do
... body of the iteration
done
```

‘for’ loop example

- The following script iterates over numbers from 1 to 10 and prints the iteration value.

```
#!/bin/sh
for i in 1 2 3 4 5 6 7 8 9 10
do
echo $i
Done
```

The following script iterates over numbers from 1 to 10 and prints “Hello” for numbers less than or equal to 5 and “World” for numbers greater than 5.

```
#!/bin/sh
for i in 1 2 3 4 5 6 7 8 9 10
do
    if [ $i -le 5 ] then
    echo Hello
    else
    echo World
    fi
done
```

Executing Mathematical Operations

- The **let internal command can execute mathematical** operations on variables
- It is present only in **bash and not in other Bourne shells** (sh, dash, etc.)

```
 #!/bin/bash
for i in 1 2 3 4 5 6 7 8 9 10
do
  let result = $i * 2
  echo $result
done
```

Executing Mathematical Operations

- In any case, for mathematical and other operations, there is the **expr program**
- It executes the mathematical operation given as argument
- and prints the result It also can perform operations on strings, such as computing the length, extracting a substring, etc.
- If it is enclosed into a reverse-quoted string, the output can be gathered into another variable

The following example prints the value of the iteration variable times 2 (quoting the “*” with backslash is necessary since the basic meaning of “*” is “all files”):

```
 #!/bin/sh
for i in 1 2 3 4 5 6 7 8 9 10
do
        result=‘expr $i \* 2‘
        echo $result
done
```

Generating Numerical Sequence

- The examples given uses a sequence of numbers in a **for** statement, which is given manually
- By using the **seq program we can automatically generate a** certain numeric sequence
- Syntax is:

  seq start end

  seq start increment end

Example:

```
#!/bin/sh
for i in ‘seq 1 10‘
do
        result=‘expr $i \* 2‘
        echo $result
done
```

Iteration: the “while” construct

- The shell provides also a **while construct**
- The condition tested has the same semantics of the **if: it** runs a command and test the exit value
- The syntax of **while is:**

```
while command
do
... body of the iteration
done
```

write a script which iterates on the arguments given and prints each argument We use the internal command **shift which performs a** “parameter shifting”, i.e. removes the first parameter from the list and “shifts left” the other; it also decrements the variable **$# “number of parameters”.**

```
 #!/bin/sh
while [ "$#" -gt "0" ]
do
    echo $1
    shift
done
```

Conditionals: the “case” construct

Like any structured programming language, the shell also

has a **case construct**

The syntax of **case is:**

case expr in first-case**)**

body of first case**;;**

second-case**)**

body of second case**;;**

.... ***)**

body of default case**;;**

esac

```
#!/bin/sh FRUIT=“Mango"
case "$FRUIT" in
    "apple") echo "Apple pie is quite tasty." ;;
    "banana") echo "I like banana." ;;
    "kiwi") echo “India is famous for Mango." ;;
Esac
```

Upon execution of above code, you will receive the following result –

India is famous for Mango

Exercise 1

- write a script which performs a recursive copy of all the files of a **source dir into a destination dir**
- For each file:
  - if it is regular, it has to be copied directly;
  - if it is a directory, it has be created (if not exists) or re-created (if exists)
- To write the script, we can use the "**find dir" program,** which prints all the files of dir scanning recursively also the
- sub-dirs.

Exercise 1 -

```
#!/bin/bash
if [ "$#" -ne "2" ] then
   echo usage: $0 source-dir dest-dir
exit 1
fi
cd $1
   for file in ‘find .‘
do
   if [ "$file" == "." ] then
continue
fi
```

```
if [ -d $file ] then
            # the file is a directory
if [ -e $2/$file ] then
            rm -rf $2/$file
fi
            mkdir $2/$file
else
# the file is a regular file?
            cp $file $2/$file
fi
done
```

Exercise 2

- check if the dest dir is not empty; in this case do not perform the copy unless option “-w” is specified
- **check if the dest dir is not empty; in this case, ask to** overwrite the content unless option “-w” is specified which, instead, overwrites the content by default

Exercise 2

```
#!/bin/bash
if [ "$1" == "-w" ]; then
        FORCE_OVERWRITE=y
        shift
else
        FORCE_OVERWRITE=n
fi
if [ "$#" -ne "2" ]; then
        echo usage: $0 source-dir dest-dir
        exit 1
fi
cd $1
for file in ‘find .‘; do
        if [ "$file" == "." ] ; then
continue
fi
if [ -d $file ] ; then
        # the file is a directory
if [ -e $2/$file ] ; then
if [ "$FORCE_OVERWRITE" == "n" ] ; then
        echo Destination directory $2/$file is not empty
exit 2
else
        rm -rf $2/$file
fi
fi
        mkdir $2/$file
else
        # the file is a regular file?
        cp $file $2/$file
fi
done
```

Exercise 3

- Write a shell script to accept number from user and print if it is odd number or even number

```
#!/bin/bash
echo -n "Enter number : " read n
rem=$(( $n % 2 ))
if [ $rem -eq 0 ] then
  echo "$n is even number"
else
  echo "$n is odd number"
fi
```

Exercise 4

- Write a shell script to find all prime numbers in between 1 to 100.

```
#!/bin/bash
for ((i=1; i<=100; i++))
do output=$(( $i % 2))
        if [ $output -ne 0 ] then
                echo "We got odd
numbers: $i"
        fi
done
```

Exercise 5

- Write a script that gives menu to run the following:
  - List of active users on the system
  - Process list for the system
  - Accept file name and print permission, owner and group details for the file
  - Date/time when the machine was last rebooted

Exercise 5

```
#!/bin/bash
while :
do
 clear
 echo "   M A I N - M E N U"
 echo "1. List of active users on the system"
 echo "2. Process list on the system"
 echo "3. Accept file name and print permission, owner and group details for the file"
 echo "4. Date/time when the machine was last rebooted"
 echo "5. Exit"
 echo -n "Please enter option [1 - 5]"
 read opt
case $opt in
  1) echo "************ List of active users on the system ************";
     who;;
 2) echo "*********** Process list on the system";
    who | more;;
3) echo "You are in $(pwd) directory";
    echo "Enter file name to print permission, owner and group details";
    read fname;
    ls -l $fname | awk '{print $1, $3, $4}';;
  5) echo "************ Date/time when the machine was last rebooted ************";
     last reboot;;
  5) echo "Bye $USER";
    exit 1;;
  *) echo "$opt is an invaild option. Please select option between 1-4 only";
    echo "Press [enter] key to continue. . .";
    read enterKey;;
esac
done
```

Functions in Shell scripting

- Functions enable you to break down the overall functionality of a script into smaller, logical subsections, which can then be called upon to perform their individual tasks when needed.
- Using functions to perform repetitive tasks is used to create **code reuse.**
- Creating Functions
  - To declare a function, simply use the following syntax –

    function_name () { list of commands }
  - The name of your function is **function_name**, and that's what you will use to call it from elsewhere in your scripts. The function name must be followed by parentheses, followed by a list of commands enclosed within braces.

Example of Function in shell script

```
#!/bin/sh
# Define your function here
Hello () { echo "Hello World" }
# Invoke your function
Hello
```

Upon execution, you will receive the following output –

```
$./test.sh Hello World
```

Pass Parameters to a Function

- You can define a function that will accept parameters while calling the function. These parameters would be represented by **$1**, **$2** and so on.
- Following is an example where we pass two parameters *Ram* and *Sharma* and then we capture and print these parameters in the function.
- Example

```
#!/bin/sh
# Define your function here
Hello () { echo "Hello World $1 $2" }
# Invoke your function Hello Ram Sharma
```

- Upon execution, you will receive the following result –

```
$./test.sh
Hello World Ram Sharma
```

Using functions

SOFTENGER
Strive to Excel

- Add two variables using function in shell script

```
#!/bin/bash
function add()
{
   sum=$(($1 + $2))
   echo "Sum = $sum"
}

a=10
b=20
#call the add function and pass the values
add $a $b
```

Returning Values from Functions

- you can return any value from your function using the **return** command and effect will be
  - not only to terminate execution of the function but also of the shell program that called the function. OR
  - just terminate execution of the function
  - Aside from creating functions and passing parameters to it, bash functions can pass the values of a function's local variable to the main routine by using the keyword *return*. The returned values are then stored to the default variable *$?* For instance, consider the following code:
- Example- function returns a value an addition of 3 numbers

```
#!/bin/bash
add(){
    sum=$(($1+$2))
    return $sum
}

read -p "Enter an integer: " int1
read -p "Enter an integer: " int2
add $int1 $int2
echo "The result of addition is: " $?
```

Recursive function

A recursive function is a function that calls itself from inside itself. This function is very useful when you need to call the function to do something again from inside of it.

```
#!/bin/bash
calc_factorial() {
if [ $1 -eq 1 ] then
    echo 1
else
    local var=$(( $1 - 1 ))
    local res=$(calc_factorial $var)
echo $(( $res * $1 ))
Fi }
read -p "Enter a number: " val
factorial=$(calc_factorial $val)
echo "The factorial of $val is: $factorial"
```

Assignments

1. Write a shell script to accept number from user and print if it is odd number or even number
2. Write a shell script to find all prime numbers in between 1 to 100.
3. Write a script that gives menu to run the following:
   - List of active users on the system
   - Process list for the system
   - Accept file name and print permission, owner and group details for the file
   - Date/time when the machine was last rebooted
4. Write a script that will extract patterns from /var/log/messages file where patterns contain the words
   - Error
   - Failed
   - Down
   - Up
   - Full
   - Unknown
   - Critical
   - Warning

Assignments

5. Write a script that shows the name of the file system that has less than 50% free space. Percentage of free should be a configurable parameter not hardcoded
6. Write a script that picks-up list of hosts from a file, connects to each of these hosts and run following commands on these hosts
    5. Uptime
    6. Uname -a
7. Write a script that accepts directory name as input and print file name where the file-modification time is older than 15 days
8. Write a script that accepts directory name as input and print file name where the file-modification time is lesser than 04 days
9. Write a script that accepts directory name as input and print file name where the size of file is lesser than 100Kb
10. Write a script that checks if following process are running on the machine.
    5. ora_pmon_test
    6. tnslsnr
    7. cron
    8. httpd
    9. sshd

Assignments

11. write a script that accepts a hostname / IP as input and checks if connectivity to that host exists
12. write a script that accepts a hostname / IP as input and draws the routing table to reach that host.
13. write a script to find files that have SUID bit set, GUID bit set, Sticky-bit set
14. from the file /var/log/httpd/ ssl_error_log find names of top-5 clients who are connecting to the server most frequently.
15. from the file /var/log/httpd/ ssl_access_log find names of top-5 clients who are connecting to the server most frequently
16. from the passwd file find the list of users, where home-directory does not exist
17. Accept a date from user. Then using the "last" command, print number of unique pairs of users and the IP's (from where they have connected to this machine) for the chosen "date"
18. Write a script that joins alternate lines and saves the output into a new file
19. Write a script that accepts a directory name, then prints
    11. Number of files in this directory
    12. sum of size of all files in this directory