```bash
#!/bin/bash
# Any lines starting with the # symbol are comments
# The very first line in a shell script starting with #! tells Linux which
# interpreter to use for running the script. For example, a Perl script would
# have #!/usr/bin/perl and Python scripts would have #!/usr/bin/python. You can
# determine the location of the interpreter using the 'which <lang>' command.

# Execution happens line by line. In the example below, the first command
# creates a file called temp, the second command prints the directory listing.
echo "Hello" > temp.txt
ls


# Variables are created simply by assignment
x=10
FILE_PATH=/home/rohit/Desktop/report.txt
# Rules:
# 1) Variable names cannot have spaces or special characters like *, ? etc.
# but underscores are allowed.
# 2) Names are case sensitive.
# 3) Try not to use any of the system defined variable names like PATH, SHELL,
# DISPLAY, PWD, PS1. You can get a list of these by the 'env' command.
# 4) Note that most system variables are uppercase, so sticking with lowercase
# names should be safe.
# 5) There should be no spaces around the = sign! x = 10 will not work!

# You can ask the user for input using the read command
echo Please enter a value for x:
read x

# To use the value of a variable, use a $ sign before its name. For example, to
# print a variable using the echo command
echo $x
y=$x
echo "The value of y is $y"
# Sometimes it is necessary to enclose the variable name in ${ }
echo "File backup path is ${FILE_PATH}_bak"   # this works properly
echo "File backup path is $FILE_PATH_bak"  # fails as FILE_PATH_bak not a var

# Math operations on variables need a special syntax $(( ))
x=5
y=20
z=$((x+y))
echo "z = x + y = $z"

# You can assign the output of a command to a variable using $( )
current_date=$(date)
echo "The date and time right now is $current_date"
# Another way to do the same thing:
echo "The date and time right now is $(date)"

# Script arguments - the blocks of text you may type after a command name - are
# available as $1, $2, $3, .. etc. Their count is stored in $#.
# E.g. script one two go
# will set $1=one, $2=two, $3=go, and $#=3


# If-Else conditional statements
# if condition
# then
#   statements
# else
#   statements
# fi

# The condition can be a program that returns success (0) or failure (non-0)
if mkdir data
```

```
then
  echo Success! The folder \'Data\' was successfully created.
else
  echo The command failed! Please make sure the folder \'data\' does not already exist.
  exit
fi

# Or the condition can be on the value of a variable
if [ $# -eq 2 ]
then
  echo Success! The arguments supplied are $1 and $2.
else
  echo You must specify exactly two arguments! Please re-run with two arguments.
  exit
fi

# The spaces in [ $# -eq 2 ] are very important. This means that
# [$# -eq 2] or [ $#-eq2 ] won't work.
# Other types of tests available are:
# Numerical tests where $a and $b are integers:
# [ $a -gt $b ]    means $a > $b
# [ $a -lt $b ]    means $a < $b
# [ $a -ge $b ]    means $a >= $b
# [ $a -le $b ]    means $a <= $b
# [ $a -eq $b ]    means $a == $b
# [ $a -ne $b ]    means $a != $b
# String tests where $a and $b are strings:
# [ $a = $b ]      means $a and $b are equal strings
# [ $a != $b ]     means $a and $b are unequal strings
# [ -z $a ]        length of string $a is zero
# [ -nz $a ]       length of string $a is non-zero
# File tests
# [ -d $a ]        the file $a exists and is a directory
# [ -f $a ]        the file $a exists and is a regular file

# for loops are written as
# for variable in list
# do
#   commands that use $variable
# done

# the list can be specified manually as
for i in 1 2 3 2 1
do
  echo $i
done

echo # print a blank line

# for longer sequences you can use {startno..endno}
for i in {1..10}
do
  echo $i
done

echo

# or it can be the list of files in a folder obtained through globbing:
# * represents all files in the current directory
# a* is all files starting with a
# *.txt is all files ending with .txt
# data/*.dat is all files in the folder 'data' ending with .dat
for filename in *.txt
do
  echo $filename
  # cat $filename # this will print the contents of the file
  # read  # pause
```

```
done

echo "The rest of the script is unsafe to run if you don't know what it's doing."
echo "This program will now exit."
exit;


# Globs cannot be used recursively. To search for files in all subdirectories,
# the find command can be used instead.
# find path_to_search -name 'filename or wildcard'
# -iname does a case-insensitive search
# The following command searches the current folder (.) and its subdirectories
# for all files ending matching the wildcard *.txt
find . -name '*.txt'

# You can run certain actions on the returned filenames. The "{}" represents
# the filename in the executed command. The exec option must be terminated by
# an escaped semicolon like \; or ';'.
find . -name '*.txt' -exec cat "{}" \;

# Be very careful when deleting files. It is recommended that deletion be done
# using the -delete option rather than -exec rm "{}" \;
# The following deletes all the temporary files ending in ~.
# NOTE: I have commented the lines below so that you do not accidentally delete
# your important files by running these commands. You run a delete command in
# the wrong folder and you have lost all the files you did not back up. There
# is no "Recycle Bin" for files deleted from the shell!
#    find . -name '*~' -delete

# Another precaution with find:
# When you don't need to search the whole subtree, use the -maxdepth N option
# to limit search to N levels. -maxdepth 1 only searches the current folder.
#    find . -maxdepth 1 -name '*~' -delete
# This is really equivalent to rm *~ but the find command has its uses.

# Example:
# This script will back up all files ending with .txt to the backup folder.
mkdir backup
find . -maxdepth 1 -name '*.txt' -exec cp "{}" backup/"{}" \;

# Special scripts:
# 1) ~/.bashrc - the script that is automatically executed when you open a shell
# 2) ~/.bash_history - records all your shell commands
```