Shell script Examples

1. To print a message

#!/bin/sh

echo "hello world"

1. the script below displays the current directory name using the **pwd** command, and then lists the directory contents using the **ls** command.

#!/bin/sh

# purpose: print out current directory name and contents

pwd

ls

3.#!/bin/sh

# name is a variable

name="fred"

echo "The name is $name"

1. The special variables $1-$9 correspond to the arguments passed to the script when it is invoked. For example, if we rewrite the script above as shown below, calling the script **name**, and then invoke the command **name Dave Smith**, the message "Your name is Dave Smith" will be printed out:

#!/bin/sh

echo "Your name is $1 $2"

1. The script below adds one to the number passed to it as an argument

#!/bin/sh

result=`expr $1 + 1`

echo "Result is $result"

**Quoting in Scripts**

Three different types of quotes are used, all of which have special meanings. We have already met two of these, and will now consider all three in detail.

Two types of quotes are basically designed to allow you to construct messages and strings. The simplest type of quotes are single quotes; anything between the two quote marks is treated as a simple string. The shell will not attempt to execute or otherwise interpret any words within the string.

The script below simply prints out the message: "your name is fred."

#!/bin/sh

echo 'Your name is fred'

#!/bin/sh

name=fred

echo 'Your name is $name'

However, this will **not** do what we want! It will actually output the message "Your name is $name", because anything between the quote marks is treated as literal text - and that includes $name.

shells also understand double quotes. The text between double quotes marks is also interpreted as literal text, except that any variables in it are interpreted. If we change the above script to use double quotes, then it will do what we want:

#!/bin/sh

name=fred

echo "Your name is $name"

The above script writes out the message: "Your name is fred." Double quotes are so useful that we normally use them rather than single quotes, which are only really needed on the rate occasions when you actually want to print out a message with variable names in it.

The third type of quotes are called back-quotes, and we have already seen them in action with the **expr** command. Back-quotes cause the Shell to treat whatever is between the quotes as a command, which is executed, then to substitute the output of the command in its place. This is the main way to get the results of commands into your script for further manipulation.

#!/bin/sh

today=date

echo "Today is $today"

The **date** command prints out today's date. The above script attempts to use it to print out today's date. However, it does not work! The message printed out is "Today is date". The reason for this is that the assignment **today=date** simply puts the string "date" into the variable today. What we actually want to do is to execute the **date** command, and place the *output* of that command into the **today** variable. We do this using back-quotes:

#/bin/sh

today=`date`

echo "Today is $today"

Back-quotes have innumerable uses. Here is another example. This uses the grep command to check whether a file includes the word "and."

#!/bin/sh

# Check for the word "and" in a file

result=`grep and $1`

if [ "$result" ]

then

echo "The file $1 includes the word and"

fi

**IF-THEN-ELSE-ENDIF**

1. , the shell below displays the contents of a file on the screen using **cat**, but lists the contents of a directory using **ls**.

#!/bin/sh

# show script

if [ -d $1 ]

then

ls $1

else

cat $1

fi

1. There are a number of conditions supported by shell scripts; for a complete list, use the on-line manual on the **test** command (**man test**). Some examples are: **-d** (is a directory?), **-f** (is a file?), **=** (are two strings the same?), **-r** (is string set?), **-eq** (are two numbers equal?), **-gt** (is first number greater than second?). You can also test whether a variable is set to anything, simply by enclosing it in quotes in the condition part of the if-statement.

#!/bin/sh

# Script to check that the user enters one argument, "fred"

if [ "$1" ]

then

echo "Found an argument to this script"

if [ $1 = "fred" ]

then

echo "The argument was fred!"

else

echo "The argument was not fred!"

fi

else

echo "This script needs one argument"

fi

1. The **case** statement

#!/bin/sh

case "$1" in

fred)

echo "Hi fred. Nice to see you"

;;

joe)

echo "Oh! Its you, is it, joe?"

;;

harry)

echo "Clear off!"

;;

\*)

echo "Who are you?"

;;

esac

The case-statement compares the string given to it (in this case "$1", the first argument passed to the script) with the various strings, each of which is followed by a closing bracket. Once a match is found, the statements up to the double semi-colon (;;) are executed, and the case-statement ends. The asterix \* character matches anything, so having this as the last case provides a default case handler (that is, what to do if none of the other cases are matched). The keywords are **case**, **in** and **esac** (end of case).

#!/bin/sh

# join command - joins two files together to create a third

# Three parameters must be passed: two to join, the third to create

# If $3 doesn't exist, then the user can't have given all three

if [ "$3" ]

then

# this cat command will write out $1 and $2; the > operator redirects

# the output into the file $3 (otherwise it would appear on the screen)

cat $1 $2 > $3

else

echo "Need three parameters: two input and one output. Sorry."

fi

#!/bin/sh

# An alternative version of the join command

# This time we check that $# is exactly three. $# is a special

# variable which indicates how many parameters were given to

# the script by the user.

if [ $# -eq 3 ]

then

cat $1 $2 > $3

else

echo "Need exactly three parameters, sorry."

fi

#!/bin/sh

# checks whether a named file exists in a special directory (stored in

# the dir variable). If it does, prints out the top of the file using

# the head command.

# N.B. establish your own dir directory if you copy this!

dir=/home/cs0ahu/safe

if [ -f $dir/$1 ]

then

head $dir/$1

fi

**LOOPING STATEMENTS**

Whereas conditional statements allow programs to make choices about what to do, looping commands support repetition. Many scripts are written precisely because some repetitious processing of many files is required, so looping commands are extremely important.

The simplest looping command is the **while** command. An example is given below:

#!/bin/sh

# Start at month 1

month=1

while [ $month -le 12 ]

do

# Print out the month number

echo "Month no. $month"

# Add one to the month number

month=`expr $month + 1`

done

echo "Finished"

Another Example

=============

#!/bin/sh

for name in fred joe harry

do

echo "Hello $name"

done

The for-loop is most successful when combined with the ability to use wildcards to match file names in the current directory

To list all the sub-directories, but not the files, in the current directory.

#!/bin/sh

for file in \*

do

if [ -d "$file" ]

then

echo "$file"

fi

done

we will check for certain age ranges. If you are less than 20 or greater than 50, you are out of the age range. If you are between 20 and 30 you are in your 20's and so on.

#!/bin/sh

# Prompt for a user name...

echo "Please enter your age:"

read AGE

if [ "$AGE" -lt 20 ] || [ "$AGE" -ge 50 ]; then

echo "Sorry, you are out of the age range."

elif [ "$AGE" -ge 20 ] && [ "$AGE" -lt 30 ]; then

echo "You are in your 20s"

elif [ "$AGE" -ge 30 ] && [ "$AGE" -lt 40 ]; then

echo "You are in your 30s"

elif [ "$AGE" -ge 40 ] && [ "$AGE" -lt 50 ]; then

echo "You are in your 40s"

fi

Here is a simple example that uses the while loop to display the numbers zero to nine:

#!/bin/sh

a=0

while [ $a -lt 10 ]

do

echo $a

a=`expr $a + 1`

done

Here is a simple example of loop nesting, let's add another countdown loop inside the loop that you used to count to nine:

#!/bin/sh

a=0

while [ "$a" -lt 10 ] # this is loop1

do

b="$a"

while [ "$b" -ge 0 ] # this is loop2

do

echo -n "$b "

b=`expr $b - 1`

done

echo

a=`expr $a + 1`

done

Here is a simple example that uses the while loop to display the numbers zero to nine:

#!/bin/sh

a=10

while [ $a -ge 10 ]

do

echo $a

a=`expr $a + 1`

done

This loop would continue forever because a is alway greater than or equal to 10 and it would never become less than 10. So this true example of infinite loop.

Here is a simple example which shows that loop would terminate as soon as a becomes 5:

#!/bin/sh

a=0

while [ $a -lt 10 ]

do

echo $a

if [ $a -eq 5 ]

then

break

fi

a=`expr $a + 1`

done

#!/bin/sh

for var1 in 1 2 3

do

for var2 in 0 5

do

if [ $var1 -eq 2 -a $var2 -eq 0 ]

then

break 2

else

echo "$var1 $var2"

fi

done

done

The following loop makes use of continue statement which returns from the continue statement and start processing next statement:

#!/bin/sh

NUMS="1 2 3 4 5 6 7"

for NUM in $NUMS

do

Q=`expr $NUM % 2`

if [ $Q -eq 0 ]

then

echo "Number is an even number!!"

continue

fi

echo "Found odd number"

done