## Bash shell programming

The \*\*shell\*\* is a command line interpreter which provides the user interface for terminal windows. It can also be used to run scripts, even in non-interactive sessions without a terminal window, as if the commands were being directly typed in.

```

#!/bin/bash

find /usr/lib -name "\*.c" -ls

```

The first line of the script, that starts with ``#!/bin/bash`` contains the full path of the command interpreter that is to be used on the file. The command interpreter is tasked with executing statements that follow it in the script. Commonly used interpreters include:

```

/usr/bin/perl

/bin/bash

/bin/csh

/bin/tcsh

/bin/ksh

/usr/bin/python

/bin/sh

```

Scripting is not only limited to shell interpreter. It can be used for Python scripts too.

```

# ll script

-rwxr--r--. 1 root root 55 Mar 3 15:22 script

# cat script

#!/usr/bin/python

print "Welcome to the Python script"

# ./script

Welcome to the Python script

```

Scripts can be interactive too.

```

# cat script.sh

#!/bin/bash

# Interactive reading of variables

echo "ENTER YOUR NAME"

read sname

# Display of variable values

echo "WELCOME "$sname"!"

# ./script.sh

ENTER YOUR NAME

Adriano

WELCOME Adriano!

```

All shell scripts generate a return value upon finishing execution. The value can be set with the ``exit`` statement. Return values permit a process to monitor the exit state of another process often in a parent-child relationship. This helps to determine how this process terminated and take any appropriate steps necessary, contingent on success or failure. By convention, success is returned as 0, and failure is returned as a non-zero value. The return value is always stored in the ``$?`` environment variable.

```

# cat names.txt

01 Mario Rossi

02 Antonio Esposito

03 Michele Laforca

04 Antonio Esposito

# echo $?

0

# cat names

cat: names: No such file or directory

# echo $?

1

```

### Basic syntax

Scripts require you to follow a standard language syntax. Rules delineate how to define variables and how to construct and format allowed statements, etc. The table lists some special character usages within bash scripts:

|Character|Description|

|---------|-----------|

|#|Used to add a comment, except when used as \#, or as #! when starting a script|

|\\|Used at the end of a line to indicate continuation on to the next line|

|;|Used to interpret what follows as a new command|

|$|Indicates what follows is a variable|

Sometimes you may want to group multiple commands on a single line. The semicolon character is used to separate these commands and execute them sequentially as if they had been typed on separate lines.

The three commands in the following example will all execute even if the ones preceding them fail:

```

$ make ; make install ; make clean

```

However, you may want to abort subsequent commands if one fails. You can do this using the and operator:

```

$ make && make install && make clean

```

If the first command fails the second one will never be executed. A final refinement is to use the or operator:

```

$ cat file1 || cat file2 || cat file3

```

In this case, you proceed until something succeeds and then you stop executing any further steps.

### Functions

A function is a code block that implements a set of operations. Functions are useful for executing procedures multiple times perhaps with varying input variables. Functions are also often called subroutines. Using functions in scripts requires two steps:

1. Declaring a function

2. Calling a function

The function declaration requires a name which is used to invoke it. The proper syntax is:

```

function\_name () {

command...

}

```

For example, the following function is named display:

```

display () {

echo "This is a sample function"

}

```

The function can be as long as desired and have many statements. Once defined, the function can be called later as many times as necessary. In the full example shown in the figure, we are also showing an often-used refinement: how to pass an argument to the function. The first, second, ..., n-th argument can be referred to as ``$1, $2, ..., $n``. The script name is referred as ``$0``. All parameters are referred as ``$\*`` and the total number of arguments is ``$#``.

```

# cat script.sh

#!/bin/bash

echo The name of this program is: $0

echo The first argument passed from the command line is: $1

echo The second argument passed from the command line is: $2

echo The third argument passed from the command line is: $3

echo All of the arguments passed from the command line are : $\*

echo All done with $0

exit 0

#

# ./script.sh A B C

The name of this program is: ./script.sh

The first argument passed from the command line is: A

The second argument passed from the command line is: B

The third argument passed from the command line is: C

All of the arguments passed from the command line are : A B C

All done with ./script.sh

```

### Command substitution

You may need to substitute the result of a command as a portion of another command. It can be done in two ways:

1. By enclosing the inner command with backticks (`)

2. By enclosing the inner command in $( )

No matter the method, the innermost command will be executed in a newly launched shell environment, and the standard output of the shell will be inserted where the command substitution was done. Virtually any command can be executed this way. Both of these methods enable command substitution; however, the second method allows command nesting.

```

# cat ./count.sh

#!/bin/bash

echo "The " $1 " contains " $(wc -l < $1) " lines."

echo $?

# ./count.sh /var/log/messages

The /var/log/messages contains 114 lines.

0

```

In the above example, the output of the inner command becomes the argument for the outer command.

### The if statement

Conditional decision making using an if statement, is a basic construct that any useful programming or scripting language must have. When an if statement is used, the ensuing actions depend on the evaluation of specified conditions such as:

\*. Numerical or string comparisons

\*. Return value of a command (0 for success)

\*. File existence or permissions

In compact form, the syntax of an if statement is:

```

if TEST-COMMANDS; then CONSEQUENT-COMMANDS; fi

```

A more general definition is:

```

if condition

then

statements

else

statements

fi

```

The following statement checks for a file argument, and if it is found, then it displays a message

```

#!/bin/bash

if [ -f $1 ]

then

echo "The " $1 " contains " $(wc -l < $1) " lines.";

echo $?

fi

# ./count.sh /etc/passwd

The /etc/passwd contains 35 lines.

0

```

Following options for file check

|Option|Action|

|------|------|

|-e file| Check if the file exists.|

|-d file| Check if the file is a directory.|

|-f file| Check if the file is a regular file.|

|-s file| Check if the file is of non-zero size.|

|-g file| Check if the file has sgid set.|

|-u file| Check if the file has suid set.|

|-r file| Check if the file is readable.|

|-w file| Check if the file is writable.|

|-x file| Check if the file is executable.|

You can use the if statement to compare strings. The syntax is as follows:

```

if [ string1 == string2 ]

then

ACTION

fi

```

Or to compare numbers, as follows:

```

if [ exp1 OPERATOR exp2 ]

then

ACTION

fi

```

The options for operators are:

Following options for file check

|Option|Action|

|------|------|

|-eq|Equal to|

|-ne|Not equal to|

|-gt|Greater than|

|-lt|Less than|

|-ge|Greater than or equal to|

|-le|Less than or equal to|