# Bash Shell Scripting

...

# Shell scripts - Introduction

A 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` is the first line of the script, contains the full path of the command interpreter (in this case `/bin/bash`) that is to be used on the file.

```
#!/bin/bash

echo "Hello World. Please find below the listing..."

ls -lisah
```

# Shell Sessions

Two kinds of Shell Sessions - a login shell session and a non-login shell session

Login shells read one or more of the following startup files

| File | Contents |
| --- | --- |
| `/etc/profile` | A global configuration script that applies to all users |
| `~/.bash_profile` | A user's personal startup file. Can be used to extend or override settings in the global configuration script |
| `~/.bash_login` | If ~/.bash_profile is not found, bash attempts to read this script |
| `~/.profile` | If neither ~/.bash_profile nor ~/.bash_login is found, bash attempts to read this file |

# Shell Sessions

Non-login shell sessions read the following startup files

| File | Contents |
| --- | --- |
| `/etc/bash.bashrc` | A global configuration script that applies to all users |
| `~/.bashrc` | A user's personal startup file. Can be used to extend or override settings in the global configuration script |

# Shell scripts - Introduction

`/bin/bash` is the most common shell on Linux

`/etc/shells` file has list of all the available shell or use `chsh -l`

```
[ec2-user@ip-172-31-56-194 init.d]$ cat /etc/shells
/bin/sh
/bin/bash
/sbin/nologin
/bin/dash
[ec2-user@ip-172-31-56-194 init.d]$ chsh -l
/bin/sh
/bin/bash
/sbin/nologin
/bin/dash
```

# Aliases

An alias is an easy way to create a new command which acts as an abbreviation for a longer one

```
$ alias name=value
```

name of the new command

text to be executed whenever name is entered on the command line

Aliases can be created directly at the command prompt; however they will only remain in effect during your current shell session

Specify aliases in `~/.bashrc` file to make them available across sessions

# Aliases

```
[ec2-user@ip-172-31-56-194 ~]$ alias l='ls -lisah'
[ec2-user@ip-172-31-56-194 ~]$ l
total 64K
19079 4.0K drwx------ 5 ec2-user ec2-user 4.0K Apr 29 22:26 .
   18 4.0K drwxr-xr-x 3 root     root     4.0K Apr 11 20:57 ..
19087 4.0K -rw------- 1 ec2-user ec2-user  329 Apr 24 01:21 .bash_history
19081 4.0K -rw-r--r-- 1 ec2-user ec2-user   18 Aug 15  2016 .bash_logout
19080 4.0K -rw-r--r-- 1 ec2-user ec2-user  193 Aug 15  2016 .bash_profile
19098 4.0K -rw-r--r-- 1 ec2-user ec2-user  181 Apr 29 22:26 .bashrc
19095 4.0K drwxrwxr-x 2 ec2-user ec2-user 4.0K Apr 29 18:38 bin
19097 4.0K -rwxr-xr-x 1 ec2-user ec2-user   78 Apr 29 18:37 hello-world.sh
19091 4.0K -rw-rw-r-- 1 ec2-user ec2-user   24 Apr 16 16:42 my-file-1.txt
19090 4.0K -rw-rw-r-- 1 ec2-user ec2-user   22 Apr 16 16:41 my-file.txt
19088    0 -rwxr---wx 1 ec2-user ec2-user    0 Apr 16 16:26 sample
19089 4.0K -rw-rw-r-- 1 ec2-user ec2-user  118 Apr 16 16:37 sample-out
19083 4.0K drwx------ 2 ec2-user ec2-user 4.0K Apr 11 20:57 .ssh
19093 4.0K drwxr-xr-x 2 ec2-user ec2-user 4.0K Apr 23 21:50 .vim
19096  12K -rw------- 1 ec2-user ec2-user 9.5K Apr 29 22:26 .viminfo
[ec2-user@ip-172-31-56-194 ~]$
```

```
# .bashrc

# Source global definitions
if [ -f /etc/bashrc ]; then
        . /etc/bashrc
fi

# User specific aliases and functions
alias l='ls -lisah'
alias today='date +"%A, %B %-d, %Y"'
~
```

# Storing & Execution

Store in any location that is part of `$PATH`

To make the script available to all users, store in `/usr/local/bin`

To make the script available just for you, store in `$HOME/bin`

Give the shell permission to execute the script -

```
chmod +x hello-world.sh
```

```
chmod 755 hello-world.sh
```

Use `./hello-world.sh` if the script is stored in a directory that is not part of the `PATH`

# Shell Script Examples

```
[ec2-user@ip-172-31-4-157 ~]$ cd /etc/init.d/
[ec2-user@ip-172-31-4-157 init.d]$ ls -l
[ec2-user@ip-172-31-4-157 init.d]$ vi netconsole
[ec2-user@ip-172-31-4-157 init.d]$ less crond
[ec2-user@ip-172-31-4-157 init.d]$ less iptables
```

```
#!/bin/bash
#
# netconsole    This loads the netconsole module with the configured parameters.
#
# chkconfig: - 11 89
# description: Initializes network console logging
# config: /etc/sysconfig/netconsole
#
# Copyright 2002 Red Hat, Inc.
#
# Based in part on a shell script by
# Andreas Dilger <adilger@turbolinux.com>  Sep 26, 2001

PATH=/sbin:/usr/sbin:$PATH
RETVAL=0
SERVER_ADDRESS_RESOLUTION=

# Check that networking is up.
. /etc/sysconfig/network

# Source function library.
. /etc/rc.d/init.d/functions

# Default values
LOCALPORT=6666
DEV=

SYSLOGADDR=
SYSLOGPORT=514
SYSLOGMACADDR=

Kernel=$(uname -r | cut -d. -f1-2)

usage ()
{
        echo $"Usage: $0 {start|stop|status|restart|condrestart}" 1>&2
        RETVAL=2
}
```

# Interactive Example

- Comments start with #
- Read for User Input and store in a variable
- Reference Variable in the script
- Display Output

```
#!/bin/bash

# Interactive reading of variables

echo "ENTER YOUR NAME"

read sname

echo "Hello" $sname "How are you doing!"
```

# Return Values

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.

As a script executes, one can check for a specific value or condition and return success or failure as the result. By convention, success is returned as 0, and failure is returned as a non-zero value.

# Basic Syntax

| 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 |
| `>` | Redirect Output |
| `>>` | Append Output |
| `<` | Redirect Input |
| `\|` | To pipe the result to next command |

# Putting Multiple Commands on a Single Line

The ; (semicolon) character is used to group multiple commands on a single line and execute them sequentially as if they had been typed on separate lines.

| Command | Description |
| --- | --- |
| `$ make ; make install ; make clean` | All three commands will execute even if the ones preceding them fail |
| `$ make && make install && make clean` | Using &&(and) operator - aborts subsequent commands if one fails |
| `$ cat file1 \|\| cat file2 \|\| cat file3` | Proceed until something succeeds and then you stop executing any further steps |

# Output Redirection

Most operating systems accept input from the keyboard and display the output on the terminal. However, in shell scripting you can send the output to a file. The process of diverting the output to a file is called output redirection.

The > character is used to write output to a file. The > character is used to write output to a file. Characters >> will append output to a file if it exists, and act just like > if the file does not already exist.

# Input Redirection

Just as the output can be redirected to a file, the input of a command can be read from a file. The process of reading input from a file is called input redirection and uses the < character.

```
#!/bin/bash

echo “Line count”

wc -l < /tmp/sample.txt
```

# Script Parameters

Users often need to pass parameter values to a script, such as a file name, date, etc. These values can be text or numbers. Within a script, the parameter or an argument is represented with a $ and a number.

| Parameter | Meaning |
|---|---|
| `$0` | Script name |
| `$1` | First parameter |
| `$2, $3, etc` | Second, third parameter, etc. |
| `$*` | All parameters |
| `$#` | Number of arguments |

# Script Parameters Example

```
#!/bin/bash
echo "The name of this program is: $0"
echo "Number of Arguments passed: $#"
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"
```

```
#!/bin/bash
cat <<- _EOF_
  The name of this program is: $0
  Number of Arguments passed: $#
  The first argument passed from the command line is: $1
  The second argument passed from the command line is: $2
  The third argument passed from the command line is: $3
  All of the arguments passed from the command line are : $*
  All done with $0
_EOF_
```

# Here Script

A here script (also sometimes called a here document) is an additional form of I/O redirection. It provides a way to include content that will be given to the standard input of a command.

```
command << token
content to be used as command's standard input
token
```

# Command Substitution

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

- By enclosing the inner command with backticks (`)
- 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 `$( )` method allows command nesting.

Example: **`$ cd /lib/modules/$(uname -r)/`**

# Environment Variables

Almost all scripts use variables containing a value, which can be used anywhere in the script. These variables can either be user or system defined. Some examples of standard environment variables are `HOME`, `PATH`, and `HOST`.

When referenced, environment variables must be prefixed with the `$` symbol as in `$HOME`.

- `$ echo $PATH`
- `$ PATH=$PATH:/home/ec2-user/`

No prefix is required when setting or modifying the variable value

`env`, `set`, or `printenv` commands can be used to view the list of environment variables.

# Exporting Environment Variables

By default, the variables created within a script are available only to the subsequent steps of that script. Any child processes (sub-shells) do not have automatic access to the values of these variables. To make them available to child processes, they must be promoted to environment variables using the `export` statement. While child processes are allowed to modify the value of exported variables, the parent will not see any changes; exported variables are not shared, but only copied.

```
export VAR=value
```

```
VAR=value ; export VAR
```

# Functions

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

```
function_name () {

   command...

}
```

The first argument can be referred to as `$1`, the second as `$2`, etc.

# if Statement

Conditional decision making using an if statement, is a basic construct. 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

```
if condition
then
  statements
else
  statements
fi
```

```
if condition; then
  statements
else
  statements
fi
```

# Testing for Files

bash provides a set of file conditionals, that can used with the if statement, including:

| Condition | Meaning |
|---|---|
| `-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 |
| `-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 view the full list of file conditions using the command **`man 1 test`**

To create a file with the contents of man

```
man 1 test | col -b > man-test.txt
```

# Testing Strings

Syntax

```
if [ string1 == string2 ] ; then
  ACTION
fi
```

# Numerical Tests

Syntax: `exp1 -op exp2`

Specially defined operators with the if statement to compare numbers.

| Operator | Meaning |
|---|---|
| `-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 |

# Arithmetic Expressions

Arithmetic expressions can be evaluated in the following three ways (spaces are important!):

- Using the expr utility: `expr` is a standard but somewhat deprecated program.
  - `expr 8 + 8`
  - `echo $(expr 8 + 8)`
- Using the `$((...))` syntax: This is the built-in shell format.
  - `echo $((x+1))`
- Using the built-in shell command let.
  - `let x=( 1 + 2 ); echo $x`

# Tracing

It is possible to have bash show  what it is doing when you run your script - Tracing

Tracing can be turned on by

- Adding -x to the first line of the script

```
#!/bin/bash -x
```

- Using set command

```
set -x    # turn on tracing

set +x    # turn off tracing
```