Study Guide

The System Administrator's Guide to Bash Scripting

Terry Cox
terry@linuxacademy.com
Aug 1, 2019

Contents

Terminal Shortcuts

Using the **up arrow** key

Up arrow: shows the previous command; you can cycle through all previous commands one by one by pressing it repeatedly.
Tab: Tab completion, used to automatically complete long strings, such as file names or locations
CTRL + A: Positions the cursor at the front of line
CTRL + E: Positions the cursor at the end of the line
CTRL + C: Cancels the current operation
CTRL + K: Deletes everything after the cursor location
CTRL + U: Deletes all text before the cursor location
CTRL + L: Clears the terminal
CTRL + R: Searches command history
CTRL + Z: Sends the current operation to the background

Special Files

`.` Current directory
`..` Parent directory
`../` Parent directory, including slash; used to navigate from the parent
`../../` The parent of the parent directory
`~/` The current user's home directory
`.hiddenfile` Files that start with a dot are hidden files. They are generally configuration files

Wildcards

`?` Represents any one character
`*` Represents any set of characters
`[xbc]` Represents any one of the characters listed within the bracket
`[a-z]` Represents any character between the defined range

History

View and edit commands previously run
`~/.bash_history`: Location of your Bash history
CTL + R: Searches the history of commands
`history`: Shows a list of commands that have been previously run in a numbered list
`!<history number>`: Shows specified command
`!14` for example shows the command numbered *14* listed by `history`
`!xy`: Runs the last command that starts with an *xy*
`!!`: Runs the last command
`fc`: Opens the previous command in a text editor; after closing the text editor, the modified command will be run in Bash

Screen

Screen is a manager for multiple terminal screens

`screen` Opens a new terminal window
`screen -S <name>` Opens a new screen with the specified name
`screen -list` Lists all active screens with numeric ID and name
`screen -d <screen ID or name>` Detaches from the specified screen and returns you to the original starting screen
`screen -r <screen ID or name>` Reattaches to the specified screen and opens it
`screen -x <screen ID or name>` Multi-display mode or screen sharing mode:
    Allows for multiple users to view and send input to the same screen
    Requires at least two different sessions, with both sessions attached to the screen
`screen <command(s)>` Executes a command in a new screen and closes when it is finished
`exit` Terminates an open screen and logs the user out if there are no other attached screens

Executing Scripts

`./program.ext` Executes program from current directory
`/path/to/program.ext` Executes program from any directory
`sh /path/to/program.ext` Runs `.bash` or `.sh` programs
`/bin/bash /path/to/program.ext` Same as above
`exec <command or path/to/program.ext>` Runs command or program as the terminal and exits the terminal once it is finished
`eval <command>` Evaluates the results of a command

I/O

I/O stands for Input / Output.

STDOUT: Standard output of command line programs
STDIN: The source of input(s) for a program
STDERR: Standard error output of a command line program

Redirection

These redirect the output or input of a command into files, devices, and the input of other commands.

`>` Redirects the standard output of a command into a file; replaces the contents of a file
`>>` Appends into the end of a file
`<` Imports the contents of a file into the command
`<<` Appends the contents of a file into the command
`2>` Redirects standard error of a command into a file
`2>>` Appends standard error of a command into the end of a file
`&>` Redirects standard error and standard output when redirecting text
`&>>` Appends standard error and standard output when redirecting text

Example: `cat < test.txt >> existingfile.txt`

Uses the contents of `test.txt` on the `cat` command, then appends the results to `existingfile.txt`

Piping

This processes commands on the output of other commands
Uses the standard output of a command prior to the pipe as the standard input for the command following the pipe

`<command1> | <command2>` Processes command2 based on the output command2

Think of it as command layering

Executing Commands on I/O

`xargs`: Reads items from the standard input and allows commands to be run on the items:

`<commands> | <xargs> <command>`

Example: `ls | grep test | xargs rm -fv`

Lists all items in the current directory, then filters the results for the string *test*, then performs a file removal with verbose output. This basically removes all files that have the string *test* in them.

Lists (for "One Liners")

In Bash, you can run multiple commands based on the following format: `<Command> <option> <Command>`

Options:

- `;` Run the following command even if the previous command fails or succeeds
- `&&` Run the following command only if the previous succeeds or has no errors
- `||` Run the following command only if the previous fails or results in error
- `&` Run the previous command in the background

Grouping Commands

Bash provides two ways to group a list of commands meant to be executed as a unit

`(list)` Parenthesis cause a subshell environment to be created:
Each of the commands in the list will be executed within that subshell
Because the list is executed within the subshell, variable assignments do not remain after the subshell completes `{ list; }` Curly braces cause the list to be executed in the current shell:
The semicolon at the end of the list is required and white space must be added before and after the list
Brace Expansion: Generates strings at the command line or in a shell script
Examples:
`{aa,bb,cc,dd} => aa bb cc dd`
`{0..12} => 0 1 2 3 4 5 6 7 8 9 10 11 12`
`{3..-2} => 3 2 1 0 -1 -2`
`{a..g} => a b c d e f g`
`{g..a} => g f e d c b a`
If the brace expansion has a prefix or suffix string, then those strings are included in the expansion:
`a{0..3}b => a0b a1b a2b a3b`
Example: `mkdir {dir1,dir2,dir3}`
Makes three folders: `dir1`, `dir2`, and `dir3`

Command Substitution

Inserts command output into another context

`` `Back Ticks` `` Input any bash command or set of commands

`$(Dollar Sign & Parenthesis)` Input any bash command or set of commands

Examples:

- `` `echo the current date is `date` `` Outputs the current date at the end of the string
- `file $(which login)` Outputs the file type of the located command file
- `echo "$(users | wc -w) users are logged in right now"` Outputs users are logged in right now

Jobs

Commands run from the terminal, whether in the foreground or in the background
In the terminal, while running a command, you can use **CTRL+Z** to stop, but not kill, a command/job. You can start it up again later, either in the foreground or background.
`jobs` Shows jobs and commands running in the background
`fg <job number>` Short for **Foreground**, and sends the specified job to the foreground of the terminal
`bg <job number>` Short for **Background**, and sends the specified job to the background of the terminal
`<command> &` Runs the command in the background, allowing you to run other commands while it processes
`nohup` Runs a command immune to hang-ups and allows a command to run even after a terminal is closed or the user who ran the command is logged out

Text Processing

"Double Quotation marks" Meta-characters enclosed within the quotes are treated literally with the exception of variables which have already been set.

Example: `name=Cameron ; echo "My name is $name"`

'single quotation marks' All meta-characters processed literally, with no variable processing

Scripts

Contain a series of commands
An interpreter executes commands in the script
Anything you can type at the command line, you can put in a script
Great for automating tasks

Basic Syntax

`#! /bin/bash`
`#` Commands
Shebang / HashBang: `#! /bin/bash`
Informs Linux which command line interpreter to use for the script. In this example, it's the Bourne Again Shell

Shell

Global Shell Configuration Files"

`/etc/profile`
`/etc/profile.d`
`/etc/bashrc`
`/etc/bash.bashrc`
`/etc/skel`
Contents of this directory are copied to new users directories when a new user is created

User Shell Configuration Files

`~/.bash_login` Executes whatever commands are within the file (`~/.bash_login`) when a user logs in
`~/.profile` User-specific Bash configuration file
`~/.bash_profile` User-specific Bash configuration file
`~/.bashrc` User-specific Bash configuration file that executes whatever commands are within the file (`~/.bash_login`) when a user logs in
`~/.bash_logout` Executes whatever commands are within the file (`~.bash_logout`) when a user logs out

Shell Variables

`set` Shows shell variables for the current instance of the running shell
Set your own shell variables: `EXAMPLE=VAR ; echo $EXAMPLE`
Creates the shell variable `EXAMPLE` and sets the value to `VAR`, then prints the variable's value
Remove shell variables: `unset EXAMPLE ; echo $EXAMPLE`
Removes the shell variable `EXAMPLE`; `echo` will show no display since `$EXAMPLE` is no longer set to any value

Environment Variables

`env` Shows all environment variables
`env | grep EXAMPLE` Prints current environment variables and then greps the result for the term *EXAMPLE*
`export EXAMPLE=VAR` Exports shell variable *EXAMPLE* to the environment variables
`EXAMPLE=VAR ; export EXAMPLE` Exports a previously-defined shell variable to the environment variables
After you log off, the environment variables you set will restore to default. To permanently set an environment variable, you must either edit the user configuration files or global configuration files for Bash.

Add to `.bashrc` (for user):

```
ABC="123"; export ABC
```

Add to `/etc/.bash.bashrc` (for system):

```
ABC="123"; export ABC
```

Common Environment Variables

`DISPLAY` X display name
`EDITOR` Name of default text editor
`HISTCONTROL` History command control options
`HOME` Path to home directory
`HOSTNAME` Current hostname
`MAIL` Holds the location of the user mail spools
`LD_LIBRARY_PATH` Directories to look for when searching for shared libraries
`PATH` Executable search path
`PS1` Current shell prompt
`PWD` Path to current working directory
`SHELL` Path to login shell
`TERM` Login terminal type
`USER / USERNAME` Current user's username
`VISUAL` Name of visual editor

Changing the Shell Prompt

Basic syntax: `PS1='\[ \] <end-of-prompt> '`
Prompt variables:
- `\h` hostname
- `\w` current working directory
- `\u` username
- `\@` 12 hour am/pm date
- `\t` 24 hour hh:mm:ss
- `\T` 12 hour hh:mm:ss
- `\j` Number of jobs running on the shell
- `\d` Date (day of week, month, day of month)
- `\H` Full hostname (hostname.domain.com)
- `\n` New line

Example: `PS1='[pwd]$ '`
Makes the shell prompt the path to current directory followed by the `$` sign
Color in the prompt, basic syntax: `\[\e[color\] <shell prompt> \[\e[m\]`
Color codes:
Reset:

```
Color_Off='\e[0m'
```

Regular Colors:

```
Black='\e[0;30m'        # Black
Red='\e[0;31m'          # Red
Green='\e[0;32m'        # Green
Yellow='\e[0;33m'       # Yellow
Blue='\e[0;34m'         # Blue
Purple='\e[0;35m'       # Purple
Cyan='\e[0;36m'         # Cyan
White='\e[0;37m'        # White
```

Bold:

```
BBlack='\e[1;30m'       # Black
```

```
BRed='\e[1;31m'         # Red
BGreen='\e[1;32m'       # Green
BYellow='\e[1;33m'      # Yellow
BBlue='\e[1;34m'        # Blue
BPurple='\e[1;35m'      # Purple
BCyan='\e[1;36m'        # Cyan
BWhite='\e[1;37m'       # White
```

Underline:

```
UBlack='\e[4;30m'       # Black
URed='\e[4;31m'         # Red
UGreen='\e[4;32m'       # Green
UYellow='\e[4;33m'      # Yellow
UBlue='\e[4;34m'        # Blue
UPurple='\e[4;35m'      # Purple
UCyan='\e[4;36m'        # Cyan
UWhite='\e[4;37m'       # White
```

Background:

```
On_Black='\e[40m'       # Black
On_Red='\e[41m'         # Red
On_Green='\e[42m'       # Green
On_Yellow='\e[43m'      # Yellow
On_Blue='\e[44m'        # Blue
On_Purple='\e[45m'      # Purple
On_Cyan='\e[46m'        # Cyan
On_White='\e[47m'       # White
```

High Intensity

```
IBlack='\e[0;90m'       # Black
IRed='\e[0;91m'         # Red
IGreen='\e[0;92m'       # Green
IYellow='\e[0;93m'      # Yellow
IBlue='\e[0;94m'        # Blue
```

```
IPurple='\e[0;95m'      # Purple
ICyan='\e[0;96m'        # Cyan
IWhite='\e[0;97m'       # White
```

Bold High Intensity

```
BIBlack='\e[1;90m'      # Black
BIRed='\e[1;91m'        # Red
BIGreen='\e[1;92m'      # Green
BIYellow='\e[1;93m'     # Yellow
BIBlue='\e[1;94m'       # Blue
BIPurple='\e[1;95m'     # Purple
BICyan='\e[1;96m'       # Cyan
BIWhite='\e[1;97m'      # White
```

High Intensity backgrounds

```
On_IBlack='\e[0;100m'   # Black
On_IRed='\e[0;101m'     # Red
On_IGreen='\e[0;102m'   # Green
On_IYellow='\e[0;103m' # Yellow
On_IBlue='\e[0;104m'    # Blue
On_IPurple='\e[0;105m' # Purple
On_ICyan='\e[0;106m'    # Cyan
On_IWhite='\e[0;107m'   # White
```

Aliases

Use them to set a string to use for another command:

`alias mycommand='<command>'` makes the string `mycommand` an alias for `command`

`alias <alias-name>` shows the command set for a certain alias

`unalias <alias-name>` removes an alias not set in the `.bashrc`

`~/.bashrc` is used to set predefined aliases

`mycommand='sh /path/to/file.sh'` makes a program executable from bash with aliases

If Statements

Basic Syntax

```
if [ condition ];
     then
          #commands to be run if true
     else
          #commands to be run if false
fi
```

Else If Syntax

When using else if within an if statement, you want to use `elif`

```
if [ condition ];
     then
          #commands to be run if true
     elif [ condition ];
     then
          #commands to be run if true
     else
          #commands to be run if false
fi
```

If Statement with Multiple Conditions

```
if [ condition ] OPERATOR [ condition ];
if [ condition ] || [ condition ];
if [ $g == 1 && $c == 123 ] || [ $g == 2 && $c == 456 ];
```

```
if [[ ( Condition ) OPERATOR ( Condition ) ]];
if [[ ( Condition ) || ( Condition ) ]];
if [[ ( $g == 1 && $c == 123 ) || ( $g == 2 && $c == 456 ) ]];
```

Case Statements

Case statements are used to check the value of a parameter and execute code depending on the value.

This is similar to the switch statement in other languages with some slight differences:

- Instead of the word `switch` , use the word `case`
- Where you would use `case`, instead list the pattern followed by a closing parenthesis
- To break the command chain, use `;;`

Basic Syntax

```
case "$VAR" in
      pattern_1 )
      # Commands to be executed
      ;;
      pattern_2 )
      # Commands to be executed
      ;;
      * )
      # Default
      ;;
esac
```

Operators

`<EXPRESSION1> && <EXPRESSION2>`: True if both expressions are true
`<EXPRESSION1> || <EXPRESSION2>`: True if at least one expression is true; do not use with `-o`
`<STRING> == <PATTERN>`: `<STRING>` is checked against the pattern `<PATTERN>`, and is true on a match
`<STRING> = <PATTERN>`: Equivalent to `==`
`<STRING> != <PATTERN>`: `<STRING>` is checked against the pattern `<PATTERN>` and is true if it does not match
`<STRING> =~ <ERE>`: `<STRING>` is checked against the extended regular expression `<ERE>` and is true on a match
`( <EXPRESSION> )`: Group expressions

File Tests

`-a <FILE>`: True if `<FILE>` exists, but may cause conflicts
`-e <FILE>`: True if `<FILE>` exists
`-f <FILE>`: True if `<FILE>` exists and is a regular file
`-d <FILE>`: True if `<FILE>` exists and is a directory
`-c <FILE>`: True if `<FILE>` exists and is a character special file
`-b <FILE>`: True if `<FILE>` exists and is a block special file
`-p <FILE>`: True if `<FILE>` exists and is a named pipe (FIFO)
`-S <FILE>`: True if `<FILE>` is a socket file
`-L <FILE>`: True if `<FILE>` exists and is a symbolic link
`-h <FILE>`: True if `<FILE>` exists and is a symbolic link
`-g <FILE>`: True if `<FILE>` exists and has sgid bit set
`-u <FILE>`: True if `<FILE>` exists and has suid bit set
`-r <FILE>`: True if `<FILE>` exists and is readable
`-w <FILE>`: True if `<FILE>` exists and is writable
`-x <FILE>`: True if `<FILE>` exists and is executable
`-s <FILE>`: True if `<FILE>` exists and has size bigger than 0
`-t <fd>`: True if file descriptor `<fd>` is open and refers to a terminal

`<FILE1> -nt <FILE2>`: True if `<FILE1>` is newer than `<FILE2>`
`<FILE1> -ot <FILE2>`: True if `<FILE1>` is older than `<FILE2>`
`<FILE1> -ef <FILE2>`: True if `<FILE1>` and `<FILE2>` refer to the same device and inode numbers

String Tests

`-z <STRING>`: True if `<STRING>` is empty
`-n <STRING>`: True if `<STRING>` is not empty, and is the default operation
`<STRING1> = <STRING2>`: True if the strings are equal
`<STRING1> != <STRING2>`: True if the strings are not equal
`<STRING1> < <STRING2>`: True if `<STRING1>` sorts before `<STRING2>` lexicographically
    Remember to escape (`\<`)
`<STRING1> > <STRING2>`: True if `<STRING1>` sorts after `<STRING2>` lexicographically
    Remember to escape (`\>`)

Arithmetic Tests

`<INTEGER1> -eq <INTEGER2>`: True if the integers are equal
`<INTEGER1> -ne <INTEGER2>`: True if the integers are ***not*** equal
`<INTEGER1> -le <INTEGER2>`: True if the first integer is less than or equal second one
`<INTEGER1> -ge <INTEGER2>`: True if the first integer is greater than or equal second one
`<INTEGER1> -lt <INTEGER2>`: True if the first integer is less than second one
`<INTEGER1> -gt <INTEGER2>`: True if the first integer is greater than second one

Misc Syntax

`<TEST1> -a <TEST2>`: True if `<TEST1>` and `<TEST2>` are true
    `-a` may also be used as a file test
`<TEST1> -o <TEST2>`: True if either `<TEST1>` or `<TEST2>` is true
`! <TEST>`: True if `<TEST>` is false

`( <TEST> )`: Group a test (for precedence)
In normal shell-usage, parentheses must be escaped
`\(` and `\)`
`-o <OPTION_NAME>`: True if the shell option `<OPTION_NAME>` is set
`-v <VARIABLENAME>`: True if the variable `<VARIABLENAME>` has been set
Use `var[n]` for array elements
`-R <VARIABLENAME>`: True if the variable `<VARIABLENAME>` has been set and is a `nameref` variable (since 4.3-alpha)

While Loop

Basic Syntax

```
while [ condition ] do
      #command(s)
      #increment
done
```

Example:

```
x=1
while [ $x -le 5 ]
do
  echo "Welcome $x times"
  x=$(( $x + 1 ))
done
```

The above loop will run a command while **x** is less than or equal to **5**
The last line adds 1 to **x** on each iteration

For Loop

Basic Syntax

```
for arg in [list]
do
     #command(s)
done
```

Any variable name can be used in place of arg
Brace-expanded {1..5} items can be used in place of [list]
During each pass through the loop, arg takes on the value of each successive variable in the list

Example:

```
for COLOR in red green blue do
     echo "COLOR: $COLOR"
done
```

Output:

```
# Color: red
# Color: green
# Color: blue
```

C-Like Syntax

```
for (( expression1; expression2; expression3 )) do
     # Command 1
     # Command 2
```

```
    # Command 3
done
```

Each expression in the for loop has a different purpose

`Expression1`: The first expression in the list is only checked the first time the for loop is ran. This is useful for setting the starting criteria of the loop.
`Expression2`: The second expression is the condition that will be evaluated at the start of each loop to see if it is true or false.
`Expression3`: The last expression is executed at the end of each loop. This comes in handy when we need to add a counter.

Example:

```
for (( SECONDS=1; SECONDS <= 60; SECONDS++ )) do
    echo $SECONDS
done
```

Will output all numbers **1** through **60**

Variables

Because everything in bash is case sensitive, it is best practice to make variables in ALL CAPS

Basic Syntax

Cannot start with a digit
Cannot contain symbols other than the underscore
No spaces between declaration and assignment
Declaration and assignment" `MY_VARIABLE="value"`
Calling variables: `$MY_VARIABLE`
Calling variables with text that precedes the variable: `echo "${MY_VARIABLE} some text"`
Assign a command output to a variable (two ways):

```
var1=$(command)
var1=`command`
```

For more information view the *Command Substitution* section, above

Booleans

Booleans are simple in Bash. Just declare a variable and assign it a true or false value

```
VAR_NAME=true
VAR_NAME=false
```

Boolean exit statuses: `0 = true` `1 = false`

Arrays

Basic Syntax

Cannot start with a digit
Cannot contain symbols other than the underscore
No spaces between declaration and assignment

Declaration

`ARRAY=()`: Declares an indexed array ARRAY and initializes it to be empty:
This can also be used to empty an existing array
`ARRAY[0]=`: Generally, sets the first element of an indexed array; if no array ARRAY existed before, it is created
`declare -a ARRAY`: Declares an indexed array `ARRAY`:
An existing array is not initialized
`declare -A ARRAY`: Declares an associative array `ARRAY`:
This is the one and only way to create associative arrays

Assignment

`ARRAY[N]=VALUE`: Sets the element N of the indexed array `ARRAY` to `VALUE`:
`N` can be any valid arithmetic expression
`ARRAY[STRING]=VALUE`: Sets the element indexed by `STRING` of the associated array `ARRAY`
`ARRAY=VALUE`: As above, if no index is given as a default, the zeroth element is set to `VALUE`
This is also true of associative arrays. There is no error if no key is specified, and the value is assigned to string index **0**
`ARRAY=(E1 E2 ...)`: Compound array assignment
Sets the whole array `ARRAY` to the given list of elements, indexed sequentially, starting at zero
The array is unset before assignment unless the `+=` operator is used
When the list is empty (`ARRAY= ()`), the array is set to an empty array

This method does not use explicit indexes and an associative array cannot be set like this
Clearing an associative array using `ARRAY=()` works
`ARRAY=([X]=E1 [Y]=E2 ...)`: Compound assignment for indexed arrays with index-value pairs declared individually (here, **X** and **Y**)
X and **Y** are arithmetic expressions
This syntax can be combined with the above
Elements declared without an explicitly-specified index are assigned sequentially starting at either the last element with an explicit index, or zero
`ARRAY=([S1]=E1 [S2]=E2 ...)`: Individual mass-setting for associative arrays
The named indexes (here, **S1** and **S2**) are strings.
`ARRAY+=(E1 E2 ...)`: Appends to `ARRAY`

Call Array Values

`${ARRAY[N]}`: Expands to the value of the index **N** in the indexed array `ARRAY`
If **N** is a negative number, it's treated as the offset from the maximum assigned index (can't be used for assignment), 1
`${ARRAY[S]}`: Expands to the value of the index **S** in the associative array `ARRAY`
`"${ARRAY[@]}"`, `${ARRAY[@]}`, `"${ARRAY[*]}"`, `${ARRAY[*]}`: Similar to mass-expanding positional parameters, this expands to all elements
If unquoted, both subscripts __*__ and __@__ expand to the same result
If quoted, **@__ expands to all elements individually quoted,** *__ expands to all elements quoted as a whole
`"${ARRAY[@]:N:M}"`, `${ARRAY[@]:N:M}`, `"${ARRAY[*]:N:M}"`, `${ARRAY[*]:N:M}`: Similar to what this syntax does for the characters of a single string, when doing substring expansion, this expands to **M** elements starting with element **N**. This way you can mass-expand individual indexes
The rules for quoting and the subscripts
__*__ and __@__ are the same as above for the other mass expansions

Positional Parameters

Used for passing arguments to your scripts at the command line
Positional parameters:
- `$0`: The first positional parameter, the script itself
- `$FUNCNAME`: The function name
  - Inside a function, `$0` is still the `$0` of the shell, not the function name
  - `$1 ... $9`: Argument list elements from **1** to **9**
  - `${10} ... ${N}`: Argument list elements beyond **9**
  - `$*`: All positional parameters except `$0`
  - `$@`: All positional parameters except `$0`
  - `$#`: Number of arguments, not counting `$0`

Basic Syntax

Example: `script.sh parameter1 parameter2 parameter3`
- `$0 = "script.sh"`
- `$1 = "parameter1"`
- `$2 = "parameter2"`
- `$3 = "parameter3"`

Example:

```
#! /bin/bash
echo $1
#This echos the first argument after the script name
echo -e "\n" #New Line
echo $2
#This echos the second argument after the script name
echo -e "\n" #New Line
```

```
echo $3
#This echos the third argument after the script name
echo -e "\n" #New Line
```

If run with the parameters **Tom Dick Harry**:

```
Tom
Dick
Harry
```

Example: `login.sh root 192.168.1.4`

Script:

```
#! /bin/bash
echo -e "Logging into host $2 with user \"${1}\" \n"
ssh -p 22 ${1}@${2}
```

Output:

`Logging into host 192.168.1.4 with user "root"`

Accept User Input

Sometimes you need to allow users running scripts to input custom data. This can be accomplished with the read command.

Basic Syntax

`read -p "Prompt" VARIABLE_TO_BE_SET`

Example:

```
#! /bin/bash
read -p "Type Your Username" USERNAME
echo -e "\n"
read -p "Type The IP Address" IPADDR
echo -e "Logging into host $IPADDR with user \"${USERNAME}\" \n"\
ssh -p 22 ${IPADDR}@${USERNAME}
```

To have formatted text at the command line, you need to know the escape sequences for `echo`
Escape sequences:

`echo -e " text <escape sequence> text`

`\a`: Alert (bell)
`\b`: Backspace
`\c`: Supress trailing newline
`\e`: Escape
`\f`: Form feed
`\n`: Newline
`\r`: Carriage return
`\v`: Vertical tab
`\\`: Backslash

Exit Statuses

This is the error status of a command. All commands return an exit status, allowing for granular control of your scripts, based on those statuses

In Bash, there are up to 255 exit statuses with 0 being the first

Exit status meanings: - `0`: Success - `1`: General Errors - `2`: Misuse of Shell Built-ins; syntax errors, missing keyword or command permission errors, etc - `Other`: Error

Global Variable

To reference the exit status of a script use $?
`$?`: Contains the return code of a previously executed command.
Exit statuses are numbered, so when you reference the variable `$?` , you get one of those numbers

Example:

```
#! /bin/bash
ls /path/does/not/exist
echo "$?"
## Output of (echo "$?") = 2
```

In Conditional Statements

In most cases, you use exit statuses within a conditional statement to perform an action based on whether your program is having errors or not.

Example:

```
#! bin/bash
HOST="google.com" ping c 1
$HOST
if [ "$?" eq "0"] then
     echo "$HOST is reachable" else
     echo "$HOST is unreachable"
fi
```

Because we're able to successfully ping google, our exit status would be **0**
We ask if our exit status is equal to **0** because if it is our output would be google.com is reachable

|| and && Operators

It may not be necessary to write out conditional statements with exit statuses. In Bash, there are two logical operators that can take the place of some conditional statements:

`command && command` - The second command will only run if the previous command succeeds
`command || command` - The second command will only run if the previous command fails

Custom Exit Statuses

There are conditions in which you may need to tell your program to halt its execution and return an exit status, whether Bash determines there is an error or not.

To tell bash to halt execution of a script and return an exit status, you would use the `exit` command.

Basic Syntax

```
exit <exit status number>
```

Example:

```
#! /bin/bash HOST="google.com" ping c 1
$HOST if ["$?" ne "0"] then
      echo "$HOST is unreachable"
      exit 1
fi
exit 0
```

This pings *google.com* with one packet, then it asks if the exit status is not equal to **0**
If exit status is not equal to **0** , then we exit with a status of **1**
If the exit status is **0**, then we simply exit with a status of **0**

Create a Function

Functions are blocks of reusable code; used when you need to do the same tasks multiple times.

Basic Syntax

```
myFunction {# Code Goes Here }
```

```
myFunction() {
    # Code Goes Here
}
```

Call a Function

Unlike other languages, calling a function in Bash does **not** entail using parentheses: - `myfunction parameter1 parameter2 parameter3`

Positional Parameters

In functions, it's possible to use positional parameters as arguments. To use positional parameters, you must first reference them within your function. Once defined, you can use your function with arguments that take on the place of the parameters:

Example:

```
function myfunction () {
     # echo -e "$1 \n"
     # echo -e "$2 \n"
     # echo -e "$3 \n"
```

```
    # echo -e "$4 \n"
}
```

```
myfunction John Mary Fred Susan
```

Output:

```
John
Mary
Fred
Susan
```

Return Codes

Each function has an exit status, and functions have their own method of dealing with exit statuses. Return codes are simply exit statuses for functions. By default, the return code of a function is simply the exit status of the last command executed within the function:

```
functionName() {
    # Code Goes Here
    return <Return Code>
}
```

Checklist

Does your script start with a shebang? - `#/bin/bash`
Does your script include a comment describing the purpose of the script? - `# This script creates a backup of every MySQL database on the system.`
Are the global variables declared at the top of your script, following the initial comments? `DEBUG=true` `HTML_DIR=/var/www`
Have you grouped all of your functions together following the global variables?
Do your functions use local variables? - `GREETING="Hello!"`
Does the main body of your shell script follow the functions?
Does your script exit with an explicit exit status? - `exit 0`

At the various exit points, are exit statuses explicitly used?

```
if [ ! d "$HTML_DIR" ]; then
    echo "$HTML_DIR does not exist. Exiting."
    exit 1
fi
```

Shell Script Template

```
#!/bin/bash
#
# Replace with the description and/or purpose of this shell script.
GLOBAL_VAR1="one"
GLOBAL_VAR2="two"
function function_one() {
local LOCAL_VAR1="one"
# Replace with function code.
}
# Main body of the shell script starts here.
#
# Replace with the main commands of your shell script.
# Exit with an explicit exit status.
exit 0
```

Syslog Standard

The syslog standard uses *facilities* and *severities* to categorize messages.
Facilities: `kern`, `user`, `mail`, `daemon`, `auth`, `local0 to local7`
Severities: `emerg`, `alert`, `crit`, `err`, `warning`, `notice`, `info`, `debug`
Log file locations: - `/var/log/messages` - `/var/log/syslog`

Log with Logger

By default, Logger creates *user.notice* messages

Basic Syntax

```
logger -p facility.severity "Message information"
logger -t tagname -p facility.severity "Message information"
```

Example: - `logger -p local10.info "Information: You are a pretty cool dude"` - `logger -t myscriptname -p local10.info "Swagnificent"`

Debugging

For detailed information regarding debugging tools for Bash, use the `help set` command.

X-Tracing and Print Debugging

X-tracing or print debugging is an option built into Bash that lets you display commands and their arguments as they are executed
Additionally, the values of variables and regex expansions will be shown.
To enable print debugging, place a `-x` after the hashbang: - `#!/bin/bash -x`
Or call it with `set`: - `set -x # Start debugging set +x # Stop debugging`

Exit on Error

Exit on error immediately halts the execution of code if any command within the script has a non-zero exit status
To enable exit on error, place a `-e` after the hashbang: - `#!/bin/bash -e`
Or call it with `set`: - `set -e # Start exit on error set +e # Stop exit on error`
Both the `-x` and `-e` options can be combined: `-xe`

Verbose Debugging

The `-v` option prints shell input lines as they are read
The verbose option is similar to x-tracing, but variables and regex are not expanded
To enable the verbose option, place a `-v` after the hashbang: - `#!/bin/bash -v`
Or call it with `set`: - `set -v # Start verbose debugging set +v # Stop verbose debugging`
Both the `-x` , `-e` and `-v` options can be combined: `-xev`

Manual Debugging

With manual debugging, we create our own debugging code. Normally, we create a special variable known as `DEBUG` to inform our script whether debugging is on or off:

```
#!/bin/bash DEBUG=true
if $DEBUG
     then
          echo "Debug Mode On." else
     echo "Debug Mode Off."
fi
```

```
$DEBUG && echo "DEBUG Mode is On"
$DEBUG || echo "DEBUG Mode is Off"
```