

Terminal Shortcuts

- **↑ Arrow** - Shows the previous command (You can cycle through all previous commands one by one using this)
- **[TAB]** - Tab Completion (Used to automatically complete long strings such as file names or locations)
- **CTL + A** - Positions the cursor at the FRONT of line
- **CTL + E** - Positions the cursor at the END of the line
- **CTL + C** - Cancels the current operation
- **CTL + K** - Delete everything after the cursor location
- **CTL + U** - Delete all text before the cursor location
- **CTL + L or clear** - Clears the terminal
- **CTL + R** - Search the history of commands
- **CRL + Z** - Sends the current operation to the background

Special Files:

- **.** - The Current Directory
- **..** - The Parent Directory
- **../** - The parent Directory with slash (Used to navigate from the parent)
- **../..** - The parent of the parent Directory
- **~/** - The current users home directory
- **●hiddenfile** - A file that starts with a dot is a hidden file. (They are normally configuration files)

Wildcards:

- **?** - Use to represent any ONE character
- ***** - Use to represent any SET of characters
- **[xbc]** - Use to represent any ONE of the characters listed within the bracket
- **[a-z]** - Use to represent any character between the range

History - View and edit commands previously ran

~/.bash_history - Location of your bash history

- **CTL + R** - Search the history of commands
- **history** - Shows a list of commands that have been ran previously in a numbered list
- **!<history number>** shows specified command
- **!xy** - Will run the last command that starts with an Xy

- **!!** - run the last command
- **fc** - opens the previous command in a text editor for editing. After closing the text editor the modified command will be ran in bash

Screen - Screen 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
(Will return you to the original starting screen)
- **screen -r <screen ID or name>** - Reattaches to the specified screen
(Will open the specified screen)
- **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 and that both sessions are 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
(Will log you out if there are no other attached screens)

Executing scripts

- **./program.ext** - Execute program from current directory
- **/path/to/program.ext** - Execute program from any directory
- **sh /path/to/program.ext** - Run .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>** - evaluate the results of a command

I/O - 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 - Redirect the output or input of a command into files, devices, and the input of other commands

- **>** - To redirect the standard output of a command into a file.
(Will replace the contents of a file
& any redirection that uses only 1 ">" will do this)
- **>>** - To append into the end of a file.
- **<** - To export the contents of a file into the the command.
- **<<** - To append the contents of a file into the command

- **2>** - To redirect standard error of a command into a file.
- **2>>** - To append standard error of a command into the end of a file.
- **&>** - To redirect standard error & standard output when redirecting text
- **&>>** - To append standard error & standard output when redirecting text

Examples

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

Piping - Processing commands on the output of other commands

Technically speaking, **piping** is using the standard output of a command prior to the pipe as the standard input for the command following the pipe

- `<command> | <command>` - Process a command on the output of the previous command

(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>`

The options you have are:

- **;** = Run the following command **Regardless** 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 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 "list" is required. White space must be added before and after the list.**

Brace Expansion - Bash brace expansion is used to generate strings at the command line or in a shell script

Some examples and what they expand to:

- {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`** - any bash command or set of commands
- **\$(Dollar Sign & Parenthesis)** - \$(any bash command or set of commands)

Examples:

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

Jobs - Commands ran from the terminal whether in the foreground or in the background

In the terminal, while running a command, you can use **CTRL+Z** to **send the command to the background**

- **jobs** - Shows jobs (or commands running in the background)
- **fg <job number>** - (**Foreground**) - Bring the specified job (command) to the foreground of the terminal
- **bg <job number>** - (**Background**) - Send the specified job (command) to the background of the terminal
- **<command> &** - Runs the command in the background (allowing you to run other commands while it is processed)
- **nohup** - Run a command immune to hangups (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 will be 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.
- Scripting is great for **automating tasks.**

Basic Syntax

#!/bin/bash

Commands

Shebang / HashBang - #! /bin/bash : Used to inform linux as to which command line interpreter is to be used for the script. In this case it is the **BOURNE AGAIN 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. 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**

Settings 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

Removing shell variables

- **unset EXAMPLE ; echo \$EXAMPLE** - Removes the shell variable "EXAMPLE" \ echo will show no display as \$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** - Export a previously defined shell variable to the environment variables

Remember, when you export a variable to environment variables, after you log off the environment variables you set will restore to default or be erased. **In order to permanently set an environment variables, you must either edit the user configuration files or global configuration files for bash.**

Permanently add environment variables for your users.

If you are using bash, ash, ksh or some other Bourne-style shell, you can add something like this

ABC="123" ; export ABC

in your **.bashrc** file (~/.bashrc). This is the default situation on most unix installations.

Permanently add environment variables on your system.

If you are using bash, ash, ksh or some other Bourne-style shell, you can add something like this

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

in your **.bash.bashrc** file (/etc/.bash.bashrc). This is the default situation on most unix installations.

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
- **LD_LIBRARY_PATH** = Directories to look for when searching for shared libraries.
- **MAIL** = Holds the location of the user mail spools
- **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="\[<shell-command,variable,or-anything>\] <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] \$ '** - Will make 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'          # Text Reset
```

```
# 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

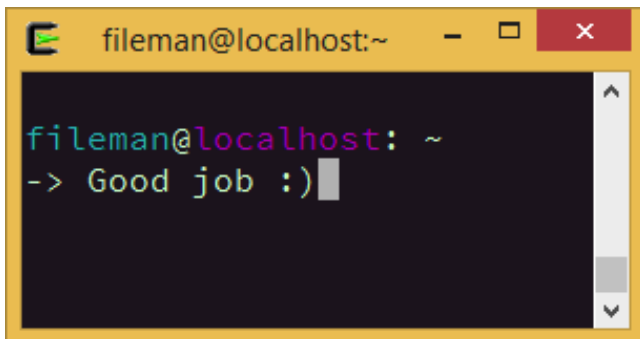
```

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

```

Example

- `export PS1='\n[\e[0;36m]\u[\e[m]\@[\e[0;35m]\h[\e[m]: \w\n-> '`



Aliases - Use to set a shortword for another command

alias mycommand='<command>' - will make the shortword "mycommand" an alias for <command>

To show the command set for a certain alias you would type -

- **alias <alias-name>**

To remove an alias not set in the .bashrc you would type -

- **unalias <alias-name>**

~/.bashrc - use to set predefined aliases

1. **vim ~/.bashrc**
2. Go to the bottom and create a new section called **#USER DEFINED ALIASES#**
3. Then type something like
4. **alias myalias='<command(s)>'**
5. Save and Exit the editor
6. type "**~/.bashrc**" without the quotes and hit enter to save changes

Making a program executable from bash with aliases - Make an "alias" that opens or runs an executable file in your .bashrc

alias mycommand='sh /path/to/file.sh'

If Statements

With IF statements, you open up with the word **if**, place your condition within brackets and close with **fi**

Basic Syntax

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

If Statement with Multiple Conditions

```
if [ condition ] OPERATOR [ condition ];
```

```
if [ condition ] || [ condition ];
```

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

Or

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

Basic Syntax


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

Operators are used within statements to check for certain conditions.

Operator	Description
(<EXPRESSION>)	Used to group expressions, to influence precedence of operators

<EXPRESSION1> && <EXPRESSION2>	<EXPRESSION1> and <EXPRESSION2> are TRUE (do not use - a!)
<EXPRESSION1> <EXPRESSION2>	TRUE if <EXPRESSION1> or <EXPRESSION2> is TRUE (do not use - o!)
<STRING> == <PATTERN>	<STRING> is checked against the pattern <PATTERN> - TRUE on a match
<STRING> = <PATTERN>	equivalent to the == operator
<STRING> != <PATTERN>	<STRING> is checked against the pattern <PATTERN> - TRUE on no match
<STRING> =~ <ERE>	<STRING> is checked against the extended regular expression <ERE> - TRUE on a match

File tests

Operator syntax	Description	
-a <FILE>	True if <FILE> exists.  (not recommended, may collide with - a for AND, see below)	
-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> exists and 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 (not empty).	
-t <fd>	True, if file descriptor <fd> is open and refers to a terminal.	
<FILE1> -nt <FILE2>	True, if <FILE1> is newer than <FILE2> (mtime).! 🚫	
<FILE1> -ot <FILE2>	True, if <FILE1> is older than <FILE2> (mtime).! 🚫	
<FILE1> -ef <FILE2>	True, if <FILE1> and <FILE2> refer to the same device and inode numbers .! 🚫	

String tests

Operator syntax	Description
-z <STRING>	True, if <STRING> is empty .
-n <STRING>	True, if <STRING> is not empty (this 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 (pure ASCII, not current locale!). Remember to escape! Use \ <
<STRING1> > <STRING2>	True if <STRING1> sorts after <STRING2> lexicographically (pure ASCII, not current locale!). Remember to escape! Use \ >

Arithmetic tests

Operator syntax	Description
<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

Operator syntax	Description
<TEST1> -a <TEST2>	True, if <TEST1> and <TEST2> are true (AND). Note that -a also may be used as a file test (see above)
<TEST1> -o <TEST2>	True, if either <TEST1> or <TEST2> is true (OR).
! <TEST>	True, if <TEST> is false (NOT).
(<TEST>)	Group a test (for precedence). Attention: In normal shell-usage, the "(" and ")" must be escaped; use "\" 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

Basic 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 a numbers 1 through 60

Variables

Because everything in bash is CASE SENSITIVE, it is best practice to make variables in ALL CAPS.

Basic Syntax

Rules

- 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

Using `$(command)` or ``command`` you can use a command in place of a variable value

For more information view the Command Substitution portion of bash basics.

```
VARIABLE_NAME=$(command)
```

or

```
VARIABLE_NAME=`command`
```

Example

```
SERVER_NAME=$(hostname)
```

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

Rules

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

Declaration

Syntax	Description
<code>ARRAY=()</code>	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

Syntax	Description
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 associative array ARRAY.
ARRAY=VALUE	As above. If no index is given, as a default the zeroth element is set to VALUE. Careful, this is even 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 will be set to an empty array. This method obviously does not use explicit indexes. An associative array can not be set like that! Clearing an associative array using ARRAY=() works.
ARRAY= ([X]=E1 [Y]=E2 ...)	Compound assignment for indexed arrays with index-value pairs declared individually (here for example 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 ...)	Append to ARRAY.

Calling Array Values

Syntax	Description
\${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

<code>\${ARRAY[S]}</code>	Expands to the value of the index S in the associative array ARRAY.
<code>"\${ARRAY[@]}"</code> <code>\${ARRAY[@]}</code> <code>"\${ARRAY[*]}"</code> <code>\${ARRAY[*]}</code>	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.
<code>"\${ARRAY[@]:N:M}"</code> <code>\${ARRAY[@]:N:M}</code> <code>"\${ARRAY[*]:N:M}"</code> <code>\${ARRAY[*]:N:M}</code>	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.

In order to pass arguments at the comand line or utilize interactive user input you want to know about **"Positional Parameters"** and the **"Read"** Command.

Positional Parameters

There are times when you need to pass pass arguments to your scripts at the command line.

These arguments are known as **positional parameters**

Parameter(s)	Description
<code>\$0</code>	the first positional parameter, the script itself .
<code>\$FUNCNAME</code>	the function name (<u>attention</u> : inside a function, \$0 is still the \$0 of the shell, not the function name)
<code>\$1 ... \$9</code>	the argument list elements from 1 to 9
<code>\${10} ... \${N}</code>	the argument list elements beyond 9
<code>\$*</code>	all positional parameters except \$0
<code>\$@</code>	all positional parameters except \$0

\$#

the number of arguments, not counting \$0

Basic Syntax

\$ script.sh parameter1 parameter2 parameter3

In the above example we have 4 parameters

- **\$0** = "script.sh"
- **\$1** = "parameter1"
- **\$2** = "parameter2"
- **\$3** = "parameter3"

The inside of our script may look something like this

```
#!/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
```

So if we ran our script from above with parameters listed below

myscript.sh Tom Dick Harry

We'd get this as our output

Tom

Dick

Harry

Here's a more practical example

Script with arguments root and IP passed

login.sh root 192.168.1.4

Our script

```
#!/bin/bash
```

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

Our output

Logging into host 192.168.1.4 with user "root"

Accepting User Input

Sometimes you will need to allow users running scripts to input custom data to have actions performed on them

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}
```

In order to have formatted text at the command line, you need to know the escape sequences for echo.

Basic Syntax

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

Sequence	Meaning
\a	Alert (bell).
\b	Backspace.
\c	Suppress trailing newline.
\e	Escape.
\f	Form feed.

\n	Newline.
\r	Carriage return.
\t	Horizontal tab.
\v	Vertical tab.
\\	Backslash.

When writing shell scripts, you want to account for conditions where possible errors could affect the functionality. In order to do this you must first understand Exit Statuses.

An **Exit Status** is simply the error status of a command. All commands return an exit status; this allows 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 Builtins
 - Syntax errors, Missing keyword or command, permission errors, ect.
- **Anything other than 0** = Error

Exit Status Global Variable

In order to reference the exit status of a script you need to use **\$?**

\$? - Contains the return code of a previously executed command.

As we mentioned before, exit statuses are numbered, so when you reference the variable **\$?**, you will get one of those numbers.

Example:

```
#!/bin/bash
ls /path/does/not/exist
echo "$?"
```

```
## Output of (echo "$?") = 2
```

Exit Statues in Conditional Statements

While its all fine and well to know what an exit status is, its more practical to do something with it.

In most cases you would find yourself using the exit status 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
## In our If statement we ask if our exit status is equal to 0
## Because it is our output would be "google.com is reachable"
```

|| and && Operators

In a lot of cases, it may not be completely 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 - When using this, the second command will only run if the previous command successful.

command || command - When using this, the second command will only run if the previous command fails.

Setting 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.

In order 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
## (Asking if the exit status is anything other than sucessful)
## 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
```

Functions are blocks of reusable code, use these within your scripts when you need to do the same tasks multiple times.

Creating a Function

Basic Syntax

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

Or

```
name() {
    # Code Goes Here.
}
```

Calling a Function

Unlike other language, when you call a function in bash you **DO NOT use parenthesis.**

Basic Syntax

```
myfunction parameter1 parameter2 parameter3 ect
```

Positional Parameters

In functions, its possible to use positional parameters as arguments for your functions.

To learn more about positional parameters, view the notes on [Passing Arguments and User Input Notes](#)

In order to use positional parameters, you must first reference them within your function.

After you've defined them 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"  
}
```

Calling A Function:

```
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.

Basic Syntax

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

Checklist

Does your script start with a shebang?

Example:

```
#!/bin/bash
```

Does your script include a comment describing the purpose of the script?

Example:

```
# 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 initialcomment(s)?

Example:

```
DEBUG=true
```

```
HTML_DIR=/var/www
```

Have you grouped all of your functions together following the global variables?

Do your functions use local variables?

Example:

```
local GREETING="Hello!"
```

Does the main body of your shell script follow the functions?

Does your script exit with an explicit exit status?

Example:

```
exit 0
```

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

Example:

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

Example Shell Script Template / Boilerplate:

```
#!/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 are configurable to:

- /var/log/**messages**
- /var/log/**syslog**

Logging with logger

- By default , 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"
```

General Debugging Information

For detailed information regarding debugging tools for bash, use the "help set" command.

```
help set | less
```

X Tracing & 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 an -x after the hashbang.

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

Can also be called with `set`

```
set -x # Start debugging
set +x # Stop debugging
```

Exit on Error

An **Exit on Error** option will immediately halt the execution of code if any command within the script has a non zero exit status.

In layman's terms this means that if at any point an error is reached during the scripts execution, it will immediately exit.

To enable Exit on Error, place an -e after the hashbang.

```
#!/bin/bash -e
```

Can also be called with `set`

```
set -e # Start exit on error
set +e # Stop exit on error
```

Both the -x and -e options can be combined, usually into -xe

Verbose Debugging

The **-v (Verbose)** option prints shell input lines as they are read.

*The **verbose** option is similar to **X tracing**, however variables and regex are **not expanded**.*

To enable the Verbose option, place an -v after the hashbang.

```
#!/bin/bash -v
```

Can also be called with `set`

```
set -v # Start verbose debugging
set +v # Stop verbose debugging
```

Both the -x, -e and -v options can be combined, usually into -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
```

```
# Or
```

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

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