Bash (Bourne Again Shell) is a shell language build on-top of the orignal Bourne Shell which was distributed with V7 Unix in 1979 and became the standard for writing shell scripts. Today it is primary to most Linux distributions, MacOS and it has even recently been enabled to run on Windows through something called WSL (Windows Subsystem for Linux).

This reference sheet should cover most syntax, tips and other tricks there is to Bash scripting. At this moment however (because the site was only recently released) I haven't gotten around to any explanations! But hopefully a lot of examples should be available in a few months as I'd have had the time to write explanations and script examples by then.

File Test Operators

Testing files in scripts is easy and straight forward. This is where shell scripting starts to show its glory! In Bash you can do file testing for permissions, size, date, filetype or existence.

Flag	Description
`-e`	File exists
`-a`	File exists (identical to -e but is deprecated and outdated)
`-f`	File is a regular file (not a directory or device file)
`-s`	file is not zero size
`-d`	file is a directory
`-b`	file is a block device
`-c`	file is a character device
`-p`	file is a pipe
`-h`	file is a symbolic link
`-L`	file is a symbolic link
`-S`	file is a socket
`-t`	file (descriptor) is associated with a terminal device; this test option may be used to check whether the stdin [-t 0] or stdout [-t 1] in a given script is a terminal

Flag	Description
`-r`	file has read permission (for the user running the test)
`-w`	file has write permission (for the user running the test)
`-x`	file has execute permission (for the user running the test)
`-g`	set-group-id (sgid) flag set on file or directory
`-u`	set-user-id (suid) flag set on file.
`-k`	sticky bit set.
`-O`	you are owner of file
`-G`	group-id of file same as yours
`-N`	file modified since it was last read
`f1 -nt f2`	file f1 is newer than f2
`f1 -ot f2`	file f1 is older than f2
`f1 -ef f2`	files f1 and f2 are hard links to the same file
`!`	Not – reverses the sense of the tests above (returns true if condition absent).

Job Identifiers

```
Job control allows you to selectively stop (suspend) the execution of
processes and continue their execution at a later point in time.
```

Notation	Description
`%N`	Job number [N]
`%S`	Invocation (command-line) of job begins with string S
`%?S`	Invocation (command-line) of job contains within it string S
`%%`	"current" job (last job stopped in foreground or started in background)
`%+`	"current" job (last job stopped in foreground or started in background)
`%-`	Last job
`%!`	Last background process

Signals

```
UNIX System V Signals.
```

Name	Number	Action	Description
SIGHUP	1	exit	Hangs up
SIGINT	2	exit	Interrupts.
SIGQUIT	3	core dump	Quits.
SIGILL	4	core dump	Illegal instruction.
SIGTRAP	5	core dump	Trace trap.
SIGIOT	6	core dump	IOT instruction.
SIGEMT	7	core dump	MT instruction.
SIGFPE	8	core dump	Floating point exception.
SIGKILL	9	exit	Kills (cannot be caught or ignored).
SIGBUS	10	core dump	Bus error.
SIGSEGV	11	core dump	Segmentation violation.
SIGSYS	12	core dump	Bad argument to system call.
SIGPIPE	13	exit	Writes on a pipe with no one to read it.
SIGALRM	14	exit	Alarm clock.
SIGTERM	15	exit	Software termination signal.

Integer Comparison Operators

How to compare integers or arithmetic expressions in shell scripts.

Flag	Description
-eq	is equal to
-ne	is not equal to
-gt	is greater than
-ge	is greater than or equal to
-lt	is less than
-le	is less than or equal to
<	is less than – place within double parentheses
<=	is less than or equal to (same rule as previous row)
>	is greater than (same rule as previous row)
>=	is greater than or equal to (same rule as previous row)

String Comparison Operators

String comparison in Bash.

Flag	Description
=	is equal to
==	same as above
!=	is not equal to
<	is less than ASCII alphabetical order
>	is greater than ASCII alphabetical order
-z	string is null (i.e. zero length)
-n	string is not null (i.e. !zero length)

Compound Operators

Useful for boolean expressions and is similar to && and ||. The compound operators work with the test command or may occur within single brackets [<expr>].

Flag	Description
-a	logical and
-o	logical or

List Constructs

Provides a means of processing commands consecutively and in effect is able to replace complex if/then/case structures.

Construct	Description
&&	and construct
||	or construct

Reserved Exit Codes

Useful for debugging a script. Exit takes integer args in the range 0-255.

Exit Code No.	Description
1	Catchall for general errors
2	Misuse of shell builtins
126	Command invoked cannot execute
127	Command not found
128	Invalid argument to exit
128+n	Fatal error signal "n"
130	Script terminated by Control-C

Sending Control Signals

> You can use these key-combinations to send signals

Key Combo	Description
Ctrl+C	The interrupt signal, sends SIGINT to the job running in the foreground.
Ctrl+Y	The delayed suspend character. Causes a running process to be stopped when it attempts to read input from the terminal. Control is returned to the shell, the user can foreground, background or kill the process. Delayed suspend is only available on operating systems supporting this feature.
Ctrl+Z	The suspend signal, sends a SIGTSTP to a running program, thus stopping it and returning control to the shell.

> Check your stty settings. Suspend and resume of output is usually disabled if you are using "modern" terminal emulations. The standard xterm supports Ctrl+S and Ctrl+Q by default.

File Types

> This is very different from Windows but straight forward once you get it. I'll expand this section soon with more context.

Symbol	Meaning
-	Regular file
d	Directory
l	(Symbolic) Link
c	Character device

Symbol	Meaning
s	Socket
p	Named pipe
b	Block device

Special Files

> Files that are read by the shell. Listed in order of their execution.

File	Info
/etc/profile	Executed automatically at login
~.bash_profile	Whichever is found first is executed at login.
~/.bash_login	
~.profile	
~/.bashrc	Is read by every nonlogin shell.

Permissions

> Now you may know what that arcane looking string rwxrwxrwx is when you invoke ls -l

Code	Description
s	setuid when in user column
s	setgid when in group column
t	sticky bit
0 -	The access right that is supposed to be on this place is not granted.
4 r	read access is granted to the user category defined in this place
2 w	write permission is granted to the user category defined in this place
1 x	execute permission is granted to the user category defined in this place
u	user permissions

Code	Description
g	group permissions
o	others permissions

String Manipulation

> Bash supports a surprisingly big number of string operations! Unfortunately, these tools lack a unified focus. Some are a subset of parameter substitution, and others fall under the functionality of the UNIX expr command. This results in inconsistent command syntax and overlap of functionality.

> MacOS built-in bash is from 2007 and won't support many of these.

Pattern	Description
`${#var}`	Find the length of the string
`${var%pattern}`	Remove from shortest rear (end) pattern
`${var%%pattern}`	Remove from longest rear (end) pattern
`${var:position}`	Extract substring from $var at $position
`${var:num1:num2}`	Substring
`${var#pattern}`	Remove from shortest front pattern
`${var##pattern}`	Remove from longest front pattern
`${var/pattern/string}`	Find and replace (only replace first occurrence)
`${var//pattern/string}`	Find and replace all occurrences
`${!prefix*}`	Expands to the names of variables whose names begin with prefix.
`${var,}` `${var,pattern}`	Convert first character to lowercase.
`${var,,}` `${var,,pattern}`	Convert all characters to lowercase.
`${var^}` `${var^pattern}`	Convert first character to uppercase.
`${var^^}` `${var^^pattern}`	Convert all character to uppercase.
`${string/substring/replacement}`	Replace first match of $substring with $replacement

Pattern	Description
`${string//substring/replacement}`	Replace all matches of $substring with $replacement
`${string/#substring/replacement}`	If $substring matches front end of $string, substitute $replacement for $substring
`${string/%substring/replacement}`	If $substring matches back end of $string, substitute $replacement for $substring
`expr match "$string" '$substring'`	Length of matching $substring* at beginning of $string
`expr "$string" : '$substring'`	Length of matching $substring* at beginning of $string
`expr index "$string" $substring`	Numerical position in $string of first character in $substring* that matches [0 if no match, first character counts as position 1]
`expr substr $string $position $length`	Extract $length characters from $string starting at $position [0 if no match, first character counts as position 1]
`expr match "$string" '\($substring\)'`	Extract $substring*, searching from beginning of $string
`expr "$string" : '\($substring\)'`	Extract $substring* , searching from beginning of $string
`expr match "$string" '.*\($substring\)'`	Extract $substring*, searching from end of $string
`expr "$string" : '.*\($substring\)'`	Extract $substring*, searching from end of $string

Quoting

> The following text shows characters or that need to be quoted if you want to use their literal symbols and not their special meaning.

Symbol	Literal Meaning
`;`	Command seperator
`&`	Background execution
`()`	Command grouping
`|`	Pipe
`< > &`	Redirection symbols
`* ? [ ] ~ + - @ !`	Filename metacharacters
`" ' \`	Used in quoting characters
`$`	Variable, command or arithmetic substituion

Symbol	Literal Meaning
#	Start a command that ends on a linebreak
space tab newline	Word seperators

Everything between "..." is taken literally, except $ (dollar) ` (backtick) and " (double-quotation).

Everything between '...' is taken literally, except ' (single-quote).

The characters following \ is taken literally.
Use \ to escape anything in "..." or '...'

Using $ before "..." or '...' causes some special behavior. $"..." is the same as "..." except that locale translation is done. Likewise, $'...' is similar to $'...' except that the quoted string is processed for escape sequences.

Command Parameters

Command parameters, also known as arguments, are used when invoking a Bash script.

Command	Description
$0	Name of the script itself
$1 … $9	Parameter 1 ... 9
${10}	Positional parameter 10
$*	Expands to the positional parameters, starting from one. When the expansion occurs within double quotes, it expands to a single word with the value of each parameter seperated by the first of the IFS environment variable
$-	Current options
$_	The underscore variable is set at shell startup and contains the absolute file name of the shell or script being executed as passed in the argument list. Subsequently, it expands to the last argument to the previous command, after expansion. It is also set to the full pathname of each command executed and placed in the environment exported to that command. When checking mail, this parameter holds the name of the mail file.
$$	Process id of the shell
$?	Exit status of the most recently executed command
$@	All arguments as separate words

Command	Description
`$#`	Number of arguments
`$!`	PID of most recently backgrounded process

History Expansion

> Enables use and manipulation of previous commands.

Command	Description
`!`	Starts a history substitution
`!!`	Refers to the last command.
`!n`	Refers to the <n>-th command line.
`!-n`	Refers to the current command line minus <n>.
`!string`	Refers to the most recent command starting with <string>
`!?string?`	Refers to the most recent command containing <string> (the ending ? is optional)
`^string1^string2^`	Quick substitution. Repeats the last command, replacing <string1> with <string2>.
`!#`	Refers to the entire command line typed so far.

Variable Operations

> Perform operations on variables.

Expression
`${parameter:-defaultValue}` Get default shell variables value
`${parameter:=defaultValue}` Set default shell variables value
`${parameter:?"Error Message"}` Display an error message if parameter is not set

Bash Globbing

> Bash cannot recognize RegEx but understand globbing. Globbing is done to filenames by the shell while RegEx is used for searching text.

Glob	Description
*	Matches zero or more occurences of a given pattern
?	Matches zero or one occurences of a given pattern
+	Matches one or more occurences of a given pattern
!	Negates any pattern matches — reverses the pattern so to speak

Character Classes In BRE

> A character class [:CharClass:] is a set of predefined patterns and comprpised of the following:

Character Class	Equivalent	Explanation
[:lower:]	[a-z]	Lowercase letters.
[:upper:]	[A-Z]	Uppercase letters.
[:alpha:]	[A-Za-z]	Alphabetic letters, both upper- and lowercase.
[:digit:]	[0-9]	Numbers 0-9.
[:alnum:]	[a-zA-Z0-9]	Alphanumeric: both letters (upper- + lowercase) and digits.
[:xdigit:]	[0-9A-Fa-f]	Hexadecimal digits.
[:space:]	[\t\n\r\f\v]	Whitespace. Spaces, tabs, newline and similar.
[:punct:]		Symbols (minus digits and letters).
[:print:]	[[:graph]]	Printable characters (spaces included).
[:blank:]	[\t]	Space and tab characters only.
[:graph:]	[^ [:cntrl:]]	Graphically printable characters excluding space.
[:cntrl:]		Control characters. Non-printable characters.

Regular Expressions

> Always use quotes in your RegEx to avoid globbing

Operator	Effect
.	Matches any single character.
?	The preceding item is optional and will be matched, at most, once.
*	The preceding item will be matched zero or more times.
+	The preceding item will be matched one or more times.

Operator	Effect
`{N}`	The preceding item is matched exactly N times.
`{N,}`	The preceding item is matched N or more times.
`{N,M}`	The preceding item is matched at least N times, but not more than M times.
`-`	Represents the range if it's not first or last in a list or the ending point of a range in a list.
`^`	Matches the empty string at the beginning of a line; also represents the characters not in the range of a list.
`$`	Matches the empty string at the end of a line.
`[aoeiAOEI]`	Matches any 1 character from the list.
`[^AOEIaoei]`	Matches any 1 character, not in the list!
`[a-f]`	Matches any 1 character in the range a-f

> In basic regular expressions the metacharacters "?", "+", "{", "|", "(", and ")" lose their special meaning; instead use the backslash versions "\?" ... "\)". Check in your system documentation whether commands using regular expressions support extended expressions.

Shell Builtins

> Shell builins are built into Bash are often very (if not extremely) fast compared to external programs. Some of the builtins are *inherited* from the Bourne Shell (sh) — these inherited commands will also work in the original Bourne Shell.

Builtin	Description
`:`	Equivalent to true.
`.`	Reads and executes commands from a designated file in the current shell.
`[`	Is a synonym for test but requires a final argument of].
`alias`	Defines an alias for the specified command.
`bg`	Resumes a job in background mode.
`bind`	Binds a keyboard sequence to a read line function or macro.
`break`	Exits from a for, while, select, or until loop.
`builtin`	Executes the specified shell built-in command.
`caller`	Returns the context of any active subroutine call
`case`	

Builtin	Description
`cd`	Changes the current directory to the specified directory.
`command`	Executes the specified command without the normal shell lookup.
`compgen`	Generates possible completion matches for the specified word.
`complete`	Displays how the specified words would be completed.
`comopt`	
`continue`	Resumes the next iteration of a for, while, select, or until loop.
`declare`	Declares a variable or variable type.
`dirs`	Displays a list of currently remembered directories.
`disown`	Removes the specified jobs from the jobs table for the process.
`echo`	Displays the specified string to STDOUT.
`enable`	Enables or disables the specified built-in shell command.
`eval`	Concatenates the specified arguments into a single command, and executes the command.
`exec`	Replaces the shell process with the specified command.
`exit`	Forces the shell to exit with the specified exit status.
`export`	Sets the specified variables to be available for child shell processes.
`fc`	Selects a list of commands from the history list.
`fg`	Resumes a job in foreground mode.
`getopts`	Parses the specified positional parameters.
`hash`	Finds and remembers the full pathname of the specified command.
`help`	Displays a help file.
`history`	Displays the command history.
`if`	Used for branching.

Builtin	Description
`jobs`	Lists active jobs.
`kill`	Sends a system signal to the specified process ID (PID).
`let`	Evaluates each argument in a mathematical expression.
`local`	Creates a limited-scope variable in a function.
`logout`	Exits a login shell.
`mapfile`	
`popd`	Removes entries from the directory stack.

Builtin	Description
`printf`	Displays text using formatted strings.
`pushd`	Adds a directory to the directory stack.
`pwd`	Displays the pathname of the current working directory.
`read`	Reads one line of data from STDIN, and assigns it to a variable.
`readonly`	Reads one line of data from STDIN, and assigns it to a variable that can't be changed.
`return`	Forces a function to exit with a value that can be retrieved by the calling script.
`set`	Sets and displays environment variable values and shell attributes.
`shift`	Rotates positional parameters down one position.
`shopt`	Toggles the values of variables controlling optional shell behavior.
`source`	Reads and executes commands from a designated file in the current shell.
`suspend`	Suspends the execution of the shell until a SIGCONT signal is received.
`test`	Returns an exit status of 0 or 1 based on the specified condition.
`times`	Displays the accumulated user and system shell time.
`trap`	Executes the specified command if the specified system signal is received.
`type`	Displays how the specified words would be interpreted if used as a command.
`typeset`	Declares a variable or variable type.
`ulimit`	Sets a limit on the specific resource for system users.
`umask`	Sets default permissions for newly created files and directories.
`unalias`	Removes specified alias.
`unset`	Removes the specified environment variable or shell attribute.
`until`	Loop that is very similar to the while-loop except that it executes until the test-command executes succesfully. As long as the test-command fails, the until-loop continues.
`wait`	Make the shell wait for a job to finish.
`while`	Waits for the specified process to complete, and returns the exit status.

Overview Of Bash Symbols

```
Here we have gathered a collection of all arcane syntax along with a brief
description. A bunch of these symbols are repeated from earlier but many are
new - this is a good starting point if you are new to the language.
```

Symbol	Quick Reference
`#`	used for comments
`$`	used for parameters and variables. Has a bunch of edge cases.
`( )`	is used for running commands in a subshell.
`$( )`	is used for saving output of commands that are send to run in a subshell.
`(( ))`	is used for arithmetic.
`$(( ))`	is used for retrieving the output of arithmetic expressions, either for usage with a command or to save the output in a variable.
`$[]`	deprecated integer expansion construct which is replaced by `$(( ))`. Evaluates integers between the square brackets
`[ ]`	is used for testing and is a built-in. Is useful in some cases for filename expansion and string manipulation.
`[[ ]]`	is used for testing. This is the one you should use unless you can think of a reason not to.
`<( )`	Used for process substitution and is similar to a pipe. Can be used whenever a command expects a file and you can use multiple at once.
`{ }`	is used for expansion of sequences
`${ }`	is used for variable interpolation and string manipulation.
`|`	is a pipe which is used for chaining commands together.
`<`	used for feeding input to commands from a file
`>`	used for sending output to a file and erasing any previous content in that file.

Symbol	Quick Reference
`||`	logical or
`&&`	logical and
`-`	used for option prefixes
`--`	used for the long-option prefixes
`&`	used to send a job to the background

`<<WORD`
`<<-WORD`
`<<'WORD'`
`<<-'WORD'`
	is used for *heredocs*

`<<<`	is used for *herestrings*
`>>`	is used to append output to a file.

Symbol	Quick Reference
' '	single quotes are used to preserve the literal value
" "	double quotes are used to preserve the literal value of all characters except $, ` ` and \
\	backslash is used to escape otherwise interpreted symbols/characters which has a special meaning
/	used for seperating the components of a filename
:	similar to a NOP – a do nothing operation. It is a shell builtin with an exit status of true
;	used to seperate commands intended to run sequentally.
,	used for linking together arithmetic operations. All are evalutated but only the last is returned
.	represents the current directory.
..	represents parent directory.
~	expands to home directory.
` `	is deprecated and should not be used. Read further in its respective section.

Flow Control

> Flow control structures in Bash are straight forward, albeit Bash is unforgiving if you get the syntax wrong.

> View examples on how to use control flow in bash.

Syntax Structure	Associated Keywords or Key Symbols	Description
If	if then fi	Test a condition.
If-else	if then else fi	Test a condition and use a fallback if the test fails.
If-elif-else	if then elif else fi	Provides additional testing plus a fallback if all tests fail. You may skip the elif conditions or add as many in-between as you like. Similarly, you may skip the else fallback.
For	for do done	Iterate over a sequence, a list or anything as far as the imagination goes.
While	while do done	While a condition is true - repeat until that condition is no longer true
Until	until do done	The inverse of the while loop - as long as the test-command fails, the until-loop continues.
Select	select in do done	Used for easy menu generation. Any statement within can be another select construct, thus enabling sub-menu creation.

Syntax Structure	Associated Keywords or Key Symbols	Description
Case	case) ;; esac	Alternative if-branching. Each case is an expression which matches a given pattern (i.e., a case).