

Bash

Notes for Professionals

Chapter 12: Arrays

Section 12.1: Array Assignments

List Assignment

If you are familiar with Perl, C, or Java, you might think that Bash would use commas to separate elements in an array, however this is not the case; instead, Bash uses spaces.

```
# Array in Perl
my @array = (1, 2, 3, 4);

# Array in Bash
array=(1 2 3 4)
```

Create an array with new elements:

```
array=(first element second element third element)
```

Subscript Assignment

Create an array with explicit element indices:

```
array[[1]]=fourth element [[4]]=fifth element
```

Assignment by Index

```
array[0]=first element
array[1]=second element
```

Assignment by name (associative array)

```
version = 4.0
declare -A array
array[first]=first element
array[second]=second element
```

Dynamic Assignment

Create an array from the output of other command, for example use `seq`:

```
array=( $(seq 1 10) )
```

Assignment from script's input arguments:

```
array=( "$@" )
```

Assignment within loops:

```
while read -r; do
    array+=("$REPLY") # Array append
    array[(i)]=${#array} # Increment index
    let i++
done < $(seq 1 10) # Command substitution
```

Bash Notes for Professionals

Chapter 21: Quoting

Section 21.1: Double quotes for variable and command substitution

Variable substitutions should only be used inside double quotes.

```
echo $var # prints 2
echo $(cat /dev/urandom) # prints 2, the list of files in the current directory, and 3
echo "${var}" # prints 2
```

Outside of double quotes, `$var` takes the value of `var`, splits it into whitespace-delimited parts, and interprets each part as a glob (wildcard) pattern. Unless you want this behavior, always put `$var` inside double quotes: `"$var"`.

The same applies to command substitutions: `$(mycommand)` is the output of `mycommand`. `$(mycommand)` is the result of splitting on the output.

```
echo "$var" # good
echo "$(mycommand)" # good
another=$(var) # also works, assignment is implicitly double-quoted
make -D "THING=$var" # good
make -D "THING=$(var)" # also good
```

Command substitutions get their own quoting contexts. Writing arbitrarily nested substitutions is easy because the parser will keep track of nesting depth instead of greedily searching for the first ``` character. The StackOverflow syntax highlighter parses this wrong, however. For example:

```
echo "formatted text: $(printf '%s = %s = %s\n' "$1" "$2")" # formatted text: a = b = 0000
```

Variable arguments to a command substitution should be double-quoted inside the expansions as well:

```
echo "$(mycommand "$arg1" "$arg2")"
```

Section 21.2: Difference between double quote and single quote

| Double quote | Single quote |
|---|---|
| Allows variable expansion | Prevents variable expansion |
| Allows history expansion if enabled | Prevents history expansion |
| Allows command substitution | Prevents command substitution |
| + and @ can have special meaning | + and @ are always literals |
| Can contain both single quote or double quote | Single quote is not allowed inside single quote |
| \$'...', ... | \$'...', ... |

Properties that are common to both:

- Prevents globbing
- Prevents word splitting

Examples:

Bash Notes for Professionals

Chapter 36: Internal variables

Section 36.1: Bash internal variables at a glance

| Variable | Details |
|-------------------------------------|---|
| <code>\$*</code> / <code>\$@</code> | Function/script positional parameters (arguments). Expand as follows: <ul style="list-style-type: none"><code>\$*</code> and <code>\$@</code> are the same as <code>\$1 \$2 ...</code> (show that it generally makes no sense to leave those unquoted)<code>\$*</code> is the same as <code>"\$1" "\$2" ...</code> (show that it generally makes no sense to leave those unquoted)<code>\$@</code> is the same as <code>"\$1" "\$2" ...</code> (show that it generally makes no sense to leave those unquoted) |
| <code>\$#</code> | Number of positional parameters passed to the script or function |
| <code>\$1</code> | Process ID of the last (right-most) for pipeline's command in the most recently job put into the background (note that it's not necessarily the same as the job's process group ID when job control is enabled) |
| <code>\$?</code> | Exit status of the last command |
| <code>\$n</code> | Positional parameters, where <code>n=1, 2, 3, ...</code> |
| <code>\$*</code> | Positional parameters (same as above), but <code>n</code> can be <code>> 9</code> |
| <code>\$*</code> | In scripts, path with which the script was invoked with <code>bash -e</code> "preserve" "exit" "set" "name" "args" : name (the first argument after the inline script), otherwise, the <code>arg[0]</code> that <code>bash</code> received |
| <code>\$IFS</code> | Last field of the last command |
| <code>\$PATH</code> | Internal field separator |
| <code>\$PWD</code> | PATH environment variable used to look-up executables |
| <code>\$PDIR</code> | Previous working directory |
| <code>\$PNAME</code> | Present working directory |
| <code>\$BASH_SOURCE</code> | Array of function names in the execution call stack |
| <code>\$BASH_ALIASES</code> | Array containing source paths for elements in <code>\$FUNCTIONS</code> array. Can be used to get the script path. |
| <code>\$BASH_REMATCH</code> | Associative array containing all currently defined aliases |
| <code>\$BASH_VERSION</code> | Array of matches from the last regex match |
| <code>\$BASH_VERSINFO</code> | Bash version string |
| <code>\$BASH</code> | Absolute path to the currently executing Bash shell itself (theoretically determined by <code>bash</code> itself on <code>arg[0]</code> and the value of <code>\$PATH</code> may be wrong in corner cases) |
| <code>\$UID</code> | Real (not effective if different) User ID of the process running <code>bash</code> |
| <code>\$PS1</code> | Primary command line prompt; see Using the PS* variables |
| <code>\$PS2</code> | Secondary command line prompt (used for additional input) |
| <code>\$PS3</code> | Tertiary command line prompt (used in select loop) |
| <code>\$PS4</code> | Quaternary command line prompt (used in select loop) |
| <code>\$RANDOM</code> | A pseudo random integer between 0 and 32767 |
| <code>\$REPLY</code> | Variable used by <code>read</code> by default (used to append info with verbose output) |
| <code>\$PIPESTATUS</code> | Array variable that holds the exit status values of each command in the most recently executed foreground pipeline. |

Bash Notes for Professionals

100+ pages
of professional hints and tricks

Contents

| | |
|---|----|
| About | 1 |
| Chapter 1: Getting started with Bash | 2 |
| Section 1.1: Hello World | 2 |
| Section 1.2: Hello World Using Variables | 4 |
| Section 1.3: Hello World with User Input | 4 |
| Section 1.4: Importance of Quoting in Strings | 5 |
| Section 1.5: Viewing information for Bash built-ins | 6 |
| Section 1.6: Hello World in "Debug" mode | 6 |
| Section 1.7: Handling Named Arguments | 7 |
| Chapter 2: Script shebang | 8 |
| Section 2.1: Env shebang | 8 |
| Section 2.2: Direct shebang | 8 |
| Section 2.3: Other shebangs | 8 |
| Chapter 3: Navigating directories | 10 |
| Section 3.1: Absolute vs relative directories | 10 |
| Section 3.2: Change to the last directory | 10 |
| Section 3.3: Change to the home directory | 10 |
| Section 3.4: Change to the Directory of the Script | 10 |
| Chapter 4: Listing Files | 12 |
| Section 4.1: List Files in a Long Listing Format | 12 |
| Section 4.2: List the Ten Most Recently Modified Files | 13 |
| Section 4.3: List All Files Including Dotfiles | 13 |
| Section 4.4: List Files Without Using `ls` | 13 |
| Section 4.5: List Files | 14 |
| Section 4.6: List Files in a Tree-Like Format | 14 |
| Section 4.7: List Files Sorted by Size | 14 |
| Chapter 5: Using cat | 16 |
| Section 5.1: Concatenate files | 16 |
| Section 5.2: Printing the Contents of a File | 16 |
| Section 5.3: Write to a file | 17 |
| Section 5.4: Show non printable characters | 17 |
| Section 5.5: Read from standard input | 18 |
| Section 5.6: Display line numbers with output | 18 |
| Section 5.7: Concatenate gzipped files | 18 |
| Chapter 6: Grep | 20 |
| Section 6.1: How to search a file for a pattern | 20 |
| Chapter 7: Aliasing | 21 |
| Section 7.1: Bypass an alias | 21 |
| Section 7.2: Create an Alias | 21 |
| Section 7.3: Remove an alias | 21 |
| Section 7.4: The BASH_ALIASES is an internal bash assoc array | 22 |
| Section 7.5: Expand alias | 22 |
| Section 7.6: List all Aliases | 22 |
| Chapter 8: Jobs and Processes | 23 |
| Section 8.1: Job handling | 23 |
| Section 8.2: Check which process running on specific port | 25 |

| | |
|---|-----------|
| Section 8.3: Disowning background job | 25 |
| Section 8.4: List Current Jobs | 25 |
| Section 8.5: Finding information about a running process | 25 |
| Section 8.6: List all processes | 26 |
| Chapter 9: Redirection | 27 |
| Section 9.1: Redirecting standard output | 27 |
| Section 9.2: Append vs Truncate | 27 |
| Section 9.3: Redirecting both STDOUT and STDERR | 28 |
| Section 9.4: Using named pipes | 28 |
| Section 9.5: Redirection to network addresses | 30 |
| Section 9.6: Print error messages to stderr | 30 |
| Section 9.7: Redirecting multiple commands to the same file | 31 |
| Section 9.8: Redirecting STDIN | 31 |
| Section 9.9: Redirecting STDERR | 32 |
| Section 9.10: STDIN, STDOUT and STDERR explained | 32 |
| Chapter 10: Control Structures | 34 |
| Section 10.1: Conditional execution of command lists | 34 |
| Section 10.2: If statement | 35 |
| Section 10.3: Looping over an array | 36 |
| Section 10.4: Using For Loop to List Iterate Over Numbers | 37 |
| Section 10.5: continue and break | 37 |
| Section 10.6: Loop break | 37 |
| Section 10.7: While Loop | 38 |
| Section 10.8: For Loop with C-style syntax | 39 |
| Section 10.9: Until Loop | 39 |
| Section 10.10: Switch statement with case | 39 |
| Section 10.11: For Loop without a list-of-words parameter | 40 |
| Chapter 11: true, false and : commands | 41 |
| Section 11.1: Infinite Loop | 41 |
| Section 11.2: Function Return | 41 |
| Section 11.3: Code that will always/never be executed | 41 |
| Chapter 12: Arrays | 42 |
| Section 12.1: Array Assignments | 42 |
| Section 12.2: Accessing Array Elements | 43 |
| Section 12.3: Array Modification | 43 |
| Section 12.4: Array Iteration | 44 |
| Section 12.5: Array Length | 45 |
| Section 12.6: Associative Arrays | 45 |
| Section 12.7: Looping through an array | 46 |
| Section 12.8: Destroy, Delete, or Unset an Array | 47 |
| Section 12.9: Array from string | 47 |
| Section 12.10: List of initialized indexes | 47 |
| Section 12.11: Reading an entire file into an array | 48 |
| Section 12.12: Array insert function | 48 |
| Chapter 13: Associative arrays | 50 |
| Section 13.1: Examining assoc arrays | 50 |
| Chapter 14: Functions | 52 |
| Section 14.1: Functions with arguments | 52 |
| Section 14.2: Simple Function | 53 |
| Section 14.3: Handling flags and optional parameters | 53 |

| | |
|--|-----------|
| Section 14.4: Print the function definition | 54 |
| Section 14.5: A function that accepts named parameters | 54 |
| Section 14.6: Return value from a function | 55 |
| Section 14.7: The exit code of a function is the exit code of its last command | 55 |
| Chapter 15: Bash Parameter Expansion | 57 |
| Section 15.1: Modifying the case of alphabetic characters | 57 |
| Section 15.2: Length of parameter | 57 |
| Section 15.3: Replace pattern in string | 58 |
| Section 15.4: Substrings and subarrays | 59 |
| Section 15.5: Delete a pattern from the beginning of a string | 60 |
| Section 15.6: Parameter indirection | 61 |
| Section 15.7: Parameter expansion and filenames | 61 |
| Section 15.8: Default value substitution | 62 |
| Section 15.9: Delete a pattern from the end of a string | 62 |
| Section 15.10: Munging during expansion | 63 |
| Section 15.11: Error if variable is empty or unset | 64 |
| Chapter 16: Copying (cp) | 65 |
| Section 16.1: Copy a single file | 65 |
| Section 16.2: Copy folders | 65 |
| Chapter 17: Find | 66 |
| Section 17.1: Searching for a file by name or extension | 66 |
| Section 17.2: Executing commands against a found file | 66 |
| Section 17.3: Finding file by access / modification time | 67 |
| Section 17.4: Finding files according to size | 68 |
| Section 17.5: Filter the path | 69 |
| Section 17.6: Finding files by type | 70 |
| Section 17.7: Finding files by specific extension | 70 |
| Chapter 18: Using sort | 71 |
| Section 18.1: Sort command output | 71 |
| Section 18.2: Make output unique | 71 |
| Section 18.3: Numeric sort | 71 |
| Section 18.4: Sort by keys | 72 |
| Chapter 19: Sourcing | 74 |
| Section 19.1: Sourcing a file | 74 |
| Section 19.2: Sourcing a virtual environment | 74 |
| Chapter 20: Here documents and here strings | 76 |
| Section 20.1: Execute command with here document | 76 |
| Section 20.2: Indenting here documents | 76 |
| Section 20.3: Create a file | 77 |
| Section 20.4: Here strings | 77 |
| Section 20.5: Run several commands with sudo | 78 |
| Section 20.6: Limit Strings | 78 |
| Chapter 21: Quoting | 80 |
| Section 21.1: Double quotes for variable and command substitution | 80 |
| Section 21.2: Difference between double quote and single quote | 80 |
| Section 21.3: Newlines and control characters | 81 |
| Section 21.4: Quoting literal text | 81 |
| Chapter 22: Conditional Expressions | 83 |
| Section 22.1: File type tests | 83 |

| | |
|---|------------|
| Section 22.2: String comparison and matching | 83 |
| Section 22.3: Test on exit status of a command | 85 |
| Section 22.4: One liner test | 85 |
| Section 22.5: File comparison | 85 |
| Section 22.6: File access tests | 86 |
| Section 22.7: Numerical comparisons | 86 |
| Chapter 23: Scripting with Parameters | 88 |
| Section 23.1: Multiple Parameter Parsing | 88 |
| Section 23.2: Argument parsing using a for loop | 89 |
| Section 23.3: Wrapper script | 89 |
| Section 23.4: Accessing Parameters | 90 |
| Section 23.5: Split string into an array in Bash | 91 |
| Chapter 24: Bash history substitutions | 92 |
| Section 24.1: Quick Reference | 92 |
| Section 24.2: Repeat previous command with sudo | 93 |
| Section 24.3: Search in the command history by pattern | 93 |
| Section 24.4: Switch to newly created directory with !#:N | 93 |
| Section 24.5: Using !\$ | 94 |
| Section 24.6: Repeat the previous command with a substitution | 94 |
| Chapter 25: Math | 95 |
| Section 25.1: Math using dc | 95 |
| Section 25.2: Math using bash capabilities | 96 |
| Section 25.3: Math using bc | 96 |
| Section 25.4: Math using expr | 97 |
| Chapter 26: Bash Arithmetic | 98 |
| Section 26.1: Simple arithmetic with (()) | 98 |
| Section 26.2: Arithmetic command | 98 |
| Section 26.3: Simple arithmetic with expr | 99 |
| Chapter 27: Scoping | 100 |
| Section 27.1: Dynamic scoping in action | 100 |
| Chapter 28: Process substitution | 101 |
| Section 28.1: Compare two files from the web | 101 |
| Section 28.2: Feed a while loop with the output of a command | 101 |
| Section 28.3: Concatenating files | 101 |
| Section 28.4: Stream a file through multiple programs at once | 101 |
| Section 28.5: With paste command | 102 |
| Section 28.6: To avoid usage of a sub-shell | 102 |
| Chapter 29: Programmable completion | 103 |
| Section 29.1: Simple completion using function | 103 |
| Section 29.2: Simple completion for options and filenames | 103 |
| Chapter 30: Customizing PS1 | 104 |
| Section 30.1: Colorize and customize terminal prompt | 104 |
| Section 30.2: Show git branch name in terminal prompt | 105 |
| Section 30.3: Show time in terminal prompt | 105 |
| Section 30.4: Show a git branch using PROMPT_COMMAND | 106 |
| Section 30.5: Change PS1 prompt | 106 |
| Section 30.6: Show previous command return status and time | 107 |
| Chapter 31: Brace Expansion | 109 |
| Section 31.1: Modifying filename extension | 109 |

| | |
|---|------------|
| Section 31.2: Create directories to group files by month and year | 109 |
| Section 31.3: Create a backup of dotfiles | 109 |
| Section 31.4: Use increments | 109 |
| Section 31.5: Using brace expansion to create lists | 109 |
| Section 31.6: Make Multiple Directories with Sub-Directories | 110 |
| Chapter 32: getopts : smart positional-parameter parsing | 111 |
| Section 32.1: pingnmap | 111 |
| Chapter 33: Debugging | 113 |
| Section 33.1: Checking the syntax of a script with "-n" | 113 |
| Section 33.2: Debugging using bashdb | 113 |
| Section 33.3: Debugging a bash script with "-x" | 113 |
| Chapter 34: Pattern matching and regular expressions | 115 |
| Section 34.1: Get captured groups from a regex match against a string | 115 |
| Section 34.2: Behaviour when a glob does not match anything | 115 |
| Section 34.3: Check if a string matches a regular expression | 116 |
| Section 34.4: Regex matching | 116 |
| Section 34.5: The * glob | 116 |
| Section 34.6: The ** glob | 117 |
| Section 34.7: The ? glob | 117 |
| Section 34.8: The [] glob | 118 |
| Section 34.9: Matching hidden files | 119 |
| Section 34.10: Case insensitive matching | 119 |
| Section 34.11: Extended globbing | 119 |
| Chapter 35: Change shell | 121 |
| Section 35.1: Find the current shell | 121 |
| Section 35.2: List available shells | 121 |
| Section 35.3: Change the shell | 121 |
| Chapter 36: Internal variables | 122 |
| Section 36.1: Bash internal variables at a glance | 122 |
| Section 36.2: \$@ | 123 |
| Section 36.3: \$# | 124 |
| Section 36.4: \$HISTSIZE | 124 |
| Section 36.5: \$FUNCNAME | 124 |
| Section 36.6: \$HOME | 124 |
| Section 36.7: \$IFS | 124 |
| Section 36.8: \$OLDPWD | 125 |
| Section 36.9: \$PWD | 125 |
| Section 36.10: \$1 \$2 \$3 etc.. | 125 |
| Section 36.11: \$* | 126 |
| Section 36.12: \$! | 126 |
| Section 36.13: \$? | 126 |
| Section 36.14: \$\$ | 126 |
| Section 36.15: \$RANDOM | 126 |
| Section 36.16: \$BASHPID | 127 |
| Section 36.17: \$BASH_ENV | 127 |
| Section 36.18: \$BASH_VERSION | 127 |
| Section 36.19: \$BASH_VERSION | 127 |
| Section 36.20: \$EDITOR | 127 |
| Section 36.21: \$HOSTNAME | 127 |
| Section 36.22: \$HOSTTYPE | 128 |

| | |
|--|-----|
| Section 36.23: \$MACHTYPE | 128 |
| Section 36.24: \$OSTYPE | 128 |
| Section 36.25: \$PATH | 128 |
| Section 36.26: \$PPID | 128 |
| Section 36.27: \$SECONDS | 128 |
| Section 36.28: \$SHELLOPTS | 129 |
| Section 36.29: \$ | 129 |
| Section 36.30: \$GROUPS | 129 |
| Section 36.31: \$LINENO | 129 |
| Section 36.32: \$SHLVL | 129 |
| Section 36.33: \$UID | 131 |
| Chapter 37: Job Control | 132 |
| Section 37.1: List background processes | 132 |
| Section 37.2: Bring a background process to the foreground | 132 |
| Section 37.3: Restart stopped background process | 132 |
| Section 37.4: Run command in background | 132 |
| Section 37.5: Stop a foreground process | 132 |
| Chapter 38: Case statement | 133 |
| Section 38.1: Simple case statement | 133 |
| Section 38.2: Case statement with fall through | 133 |
| Section 38.3: Fall through only if subsequent pattern(s) match | 133 |
| Chapter 39: Read a file (data stream, variable) line-by-line (and/or field-by-field)? | 135 |
| Section 39.1: Looping through a file line by line | 135 |
| Section 39.2: Looping through the output of a command field by field | 135 |
| Section 39.3: Read lines of a file into an array | 135 |
| Section 39.4: Read lines of a string into an array | 136 |
| Section 39.5: Looping through a string line by line | 136 |
| Section 39.6: Looping through the output of a command line by line | 136 |
| Section 39.7: Read a file field by field | 136 |
| Section 39.8: Read a string field by field | 137 |
| Section 39.9: Read fields of a file into an array | 137 |
| Section 39.10: Read fields of a string into an array | 137 |
| Section 39.11: Reads file (/etc/passwd) line by line and field by field | 138 |
| Chapter 40: File execution sequence | 140 |
| Section 40.1: .profile vs .bash_profile (and .bash_login) | 140 |
| Chapter 41: Splitting Files | 141 |
| Section 41.1: Split a file | 141 |
| Chapter 42: File Transfer using scp | 142 |
| Section 42.1: scp transferring file | 142 |
| Section 42.2: scp transferring multiple files | 142 |
| Section 42.3: Downloading file using scp | 142 |
| Chapter 43: Pipelines | 143 |
| Section 43.1: Using & | 143 |
| Section 43.2: Show all processes paginated | 144 |
| Section 43.3: Modify continuous output of a command | 144 |
| Chapter 44: Managing PATH environment variable | 145 |
| Section 44.1: Add a path to the PATH environment variable | 145 |
| Section 44.2: Remove a path from the PATH environment variable | 145 |
| Chapter 45: Word splitting | 147 |

| | |
|--|-----|
| Section 45.1: What, when and Why? | 147 |
| Section 45.2: Bad effects of word splitting | 147 |
| Section 45.3: Usefulness of word splitting | 148 |
| Section 45.4: Splitting by separator changes | 149 |
| Section 45.5: Splitting with IFS | 149 |
| Section 45.6: IFS & word splitting | 149 |
| Chapter 46: Avoiding date using printf | 151 |
| Section 46.1: Get the current date | 151 |
| Section 46.2: Set variable to current time | 151 |
| Chapter 47: Using "trap" to react to signals and system events | 152 |
| Section 47.1: Introduction: clean up temporary files | 152 |
| Section 47.2: Catching SIGINT or Ctrl+C | 152 |
| Section 47.3: Accumulate a list of trap work to run at exit | 153 |
| Section 47.4: Killing Child Processes on Exit | 153 |
| Section 47.5: react on change of terminals window size | 153 |
| Chapter 48: Chain of commands and operations | 155 |
| Section 48.1: Counting a text pattern occurrence | 155 |
| Section 48.2: transfer root cmd output to user file | 155 |
| Section 48.3: logical chaining of commands with && and | 155 |
| Section 48.4: serial chaining of commands with semicolon | 155 |
| Section 48.5: chaining commands with | 156 |
| Chapter 49: Type of Shells | 157 |
| Section 49.1: Start an interactive shell | 157 |
| Section 49.2: Detect type of shell | 157 |
| Section 49.3: Introduction to dot files | 157 |
| Chapter 50: Color script output (cross-platform) | 159 |
| Section 50.1: color-output.sh | 159 |
| Chapter 51: co-processes | 160 |
| Section 51.1: Hello World | 160 |
| Chapter 52: Typing variables | 161 |
| Section 52.1: declare weakly typed variables | 161 |
| Chapter 53: Jobs at specific times | 162 |
| Section 53.1: Execute job once at specific time | 162 |
| Section 53.2: Doing jobs at specified times repeatedly using systemd.timer | 162 |
| Chapter 54: Handling the system prompt | 164 |
| Section 54.1: Using the PROMPT_COMMAND environment variable | 164 |
| Section 54.2: Using PS2 | 165 |
| Section 54.3: Using PS3 | 165 |
| Section 54.4: Using PS4 | 165 |
| Section 54.5: Using PS1 | 166 |
| Chapter 55: The cut command | 167 |
| Section 55.1: Only one delimiter character | 167 |
| Section 55.2: Repeated delimiters are interpreted as empty fields | 167 |
| Section 55.3: No quoting | 167 |
| Section 55.4: Extracting, not manipulating | 167 |
| Chapter 56: Bash on Windows 10 | 169 |
| Section 56.1: Readme | 169 |
| Chapter 57: Cut Command | 170 |

| | |
|--|-----|
| Section 57.1: Show the first column of a file | 170 |
| Section 57.2: Show columns x to y of a file | 170 |
| Chapter 58: global and local variables | 171 |
| Section 58.1: Global variables | 171 |
| Section 58.2: Local variables | 171 |
| Section 58.3: Mixing the two together | 171 |
| Chapter 59: CGI Scripts | 173 |
| Section 59.1: Request Method: GET | 173 |
| Section 59.2: Request Method: POST /w JSON | 175 |
| Chapter 60: Select keyword | 177 |
| Section 60.1: Select keyword can be used for getting input argument in a menu format | 177 |
| Chapter 61: When to use eval | 178 |
| Section 61.1: Using Eval | 178 |
| Section 61.2: Using Eval with Getopt | 179 |
| Chapter 62: Networking With Bash | 180 |
| Section 62.1: Networking commands | 180 |
| Chapter 63: Parallel | 182 |
| Section 63.1: Parallelize repetitive tasks on list of files | 182 |
| Section 63.2: Parallelize STDIN | 183 |
| Chapter 64: Decoding URL | 184 |
| Section 64.1: Simple example | 184 |
| Section 64.2: Using printf to decode a string | 184 |
| Chapter 65: Design Patterns | 185 |
| Section 65.1: The Publish/Subscribe (Pub/Sub) Pattern | 185 |
| Chapter 66: Pitfalls | 187 |
| Section 66.1: Whitespace When Assigning Variables | 187 |
| Section 66.2: Failed commands do not stop script execution | 187 |
| Section 66.3: Missing The Last Line in a File | 187 |
| Appendix A: Keyboard shortcuts | 189 |
| Section A.1: Editing Shortcuts | 189 |
| Section A.2: Recall Shortcuts | 189 |
| Section A.3: Macros | 189 |
| Section A.4: Custome Key Bindings | 189 |
| Section A.5: Job Control | 190 |
| Credits | 191 |
| You may also like | 195 |

About

Please feel free to share this PDF with anyone for free,
latest version of this book can be downloaded from:
<https://goalkicker.com/BashBook>

This *Bash Notes for Professionals* book is compiled from [Stack Overflow Documentation](#), the content is written by the beautiful people at Stack Overflow. Text content is released under Creative Commons BY-SA, see credits at the end of this book whom contributed to the various chapters. Images may be copyright of their respective owners unless otherwise specified

This is an unofficial free book created for educational purposes and is not affiliated with official Bash group(s) or company(s) nor Stack Overflow. All trademarks and registered trademarks are the property of their respective company owners

The information presented in this book is not guaranteed to be correct nor accurate, use at your own risk

Please send feedback and corrections to web@petercv.com

Chapter 1: Getting started with Bash

Version Release Date

| | |
|-------|------------|
| 0.99 | 1989-06-08 |
| 1.01 | 1989-06-23 |
| 2.0 | 1996-12-31 |
| 2.02 | 1998-04-20 |
| 2.03 | 1999-02-19 |
| 2.04 | 2001-03-21 |
| 2.05b | 2002-07-17 |
| 3.0 | 2004-08-03 |
| 3.1 | 2005-12-08 |
| 3.2 | 2006-10-11 |
| 4.0 | 2009-02-20 |
| 4.1 | 2009-12-31 |
| 4.2 | 2011-02-13 |
| 4.3 | 2014-02-26 |
| 4.4 | 2016-09-15 |

Section 1.1: Hello World

Interactive Shell

The Bash shell is commonly used **interactively**: It lets you enter and edit commands, then executes them when you press the `Return` key. Many Unix-based and Unix-like operating systems use Bash as their default shell (notably Linux and macOS). The terminal automatically enters an interactive Bash shell process on startup.

Output Hello World by typing the following:

```
echo "Hello World"
#> Hello World # Output Example
```

Notes

- You can change the shell by just typing the name of the shell in terminal. For example: sh, **bash**, etc.
- **echo** is a Bash builtin command that writes the arguments it receives to the standard output. It appends a newline to the output, by default.

Non-Interactive Shell

The Bash shell can also be run **non-interactively** from a script, making the shell require no human interaction. Interactive behavior and scripted behavior should be identical – an important design consideration of Unix V7 Bourne shell and transitively Bash. Therefore anything that can be done at the command line can be put in a script file for reuse.

Follow these steps to create a Hello World script:

1. Create a new file called `hello-world.sh`

```
touch hello-world.sh
```

2. Make the script executable by running `chmod +x hello-world.sh`
3. Add this code:

```
#!/bin/bash
echo "Hello World"
```

Line 1: The first line of the script must start with the character sequence `#!/`, referred to as *shebang*¹. The shebang instructs the operating system to run `/bin/bash`, the Bash shell, passing it the script's path as an argument.

E.g. `/bin/bash hello-world.sh`

Line 2: Uses the `echo` command to write `Hello World` to the standard output.

4. Execute the `hello-world.sh` script from the command line using one of the following:

- `./hello-world.sh` – most commonly used, and recommended
- `/bin/bash hello-world.sh`
- `bash hello-world.sh` – assuming `/bin` is in your `$PATH`
- `sh hello-world.sh`

For real production use, you would omit the `.sh` extension (which is misleading anyway, since this is a Bash script, not a sh script) and perhaps move the file to a directory within your `$PATH` so that it is available to you regardless of your current working directory, just like a system command such as `cat` or `ls`.

Common mistakes include:

1. Forgetting to apply execute permission on the file, i.e., `chmod +x hello-world.sh`, resulting in the output of `./hello-world.sh: Permission denied`.
2. Editing the script on Windows, which produces incorrect line ending characters that Bash cannot handle.

A common symptom is : `command not found` where the carriage return has forced the cursor to the beginning of line, overwriting the text before the colon in the error message.

The script can be fixed using the `dos2unix` program.

An example use: `dos2unix hello-world.sh`

dos2unix edits the file inline.

3. Using `sh ./hello-world.sh`, not realizing that `bash` and `sh` are distinct shells with distinct features (though since Bash is backwards-compatible, the opposite mistake is harmless).

Anyway, simply relying on the script's shebang line is vastly preferable to explicitly writing `bash` or `sh` (or `python` or `perl` or `awk` or `ruby` or...) before each script's file name.

A common shebang line to use in order to make your script more portable is to use `#!/usr/bin/env bash` instead of hard-coding a path to Bash. That way, `/usr/bin/env` has to exist, but beyond that point, `bash` just

needs to be on your PATH. On many systems, `/bin/bash` doesn't exist, and you should use `/usr/local/bin/bash` or some other absolute path; this change avoids having to figure out the details of that.

¹ Also referred to as *sha-bang*, *hashbang*, *pound-bang*, *hash-pling*.

Section 1.2: Hello World Using Variables

Create a new file called `hello.sh` with the following content and give it executable permissions with `chmod +x hello.sh`.

Execute/Run via: `./hello.sh`

```
#!/usr/bin/env bash

# Note that spaces cannot be used around the `=` assignment operator
whom_variable="World"

# Use printf to safely output the data
printf "Hello, %s\n" "$whom_variable"
#> Hello, World
```

This will print `Hello, World` to standard output when executed.

To tell bash where the script is you need to be very specific, by pointing it to the containing directory, normally with `./` if it is your working directory, where `.` is an alias to the current directory. If you do not specify the directory, `bash` tries to locate the script in one of the directories contained in the `$PATH` environment variable.

The following code accepts an argument `$1`, which is the first command line argument, and outputs it in a formatted string, following `Hello, .`

Execute/Run via: `./hello.sh World`

```
#!/usr/bin/env bash
printf "Hello, %s\n" "$1"
#> Hello, World
```

It is important to note that `$1` has to be quoted in double quote, not single quote. `"$1"` expands to the first command line argument, as desired, while `'$1'` evaluates to literal string `$1`.

Security Note:

Read [Security implications of forgetting to quote a variable in bash shells](#) to understand the importance of placing the variable text within double quotes.

Section 1.3: Hello World with User Input

The following will prompt a user for input, and then store that input as a string (text) in a variable. The variable is then used to give a message to the user.

```
#!/usr/bin/env bash
echo "Who are you?"
read name
echo "Hello, $name."
```

The command **read** here reads one line of data from standard input into the variable **name**. This is then referenced using **\$name** and printed to standard out using **echo**.

Example output:

```
$ ./hello_world.sh
Who are you?
Matt
Hello, Matt.
```

Here the user entered the name "Matt", and this code was used to say Hello, Matt..

And if you want to append something to the variable value while printing it, use curly brackets around the variable name as shown in the following example:

```
#!/usr/bin/env bash
echo "What are you doing?"
read action
echo "You are ${action}ing."
```

Example output:

```
$ ./hello_world.sh
What are you doing?
Sleep
You are Sleeping.
```

Here when user enters an action, "ing" is appended to that action while printing.

Section 1.4: Importance of Quoting in Strings

Quoting is important for string expansion in bash. With these, you can control how the bash parses and expands your strings.

There are two types of quoting:

- **Weak:** uses double quotes: "
- **Strong:** uses single quotes: '

If you want to bash to expand your argument, you can use **Weak Quoting**:

```
#!/usr/bin/env bash
world="World"
echo "Hello $world"
#> Hello World
```

If you don't want to bash to expand your argument, you can use **Strong Quoting**:

```
#!/usr/bin/env bash
world="World"
echo 'Hello $world'
```



```
#> Hello $world
```

You can also use escape to prevent expansion:

```
#!/usr/bin/env bash
world="World"
echo "Hello \ $world"
#> Hello $world
```

For more detailed information other than beginner details, you can continue to read it [here](#).

Section 1.5: Viewing information for Bash built-ins

```
help <command>
```

This will display the Bash help (manual) page for the specified built-in.

For example, `help unset` will show:

```
unset: unset [-f] [-v] [-n] [name ...]
Unset values and attributes of shell variables and functions.

For each NAME, remove the corresponding variable or function.

Options:
-f      treat each NAME as a shell function
-v      treat each NAME as a shell variable
-n      treat each NAME as a name reference and unset the variable itself
        rather than the variable it references

Without options, unset first tries to unset a variable, and if that fails,
tries to unset a function.

Some variables cannot be unset; also see `readonly'.

Exit Status:
Returns success unless an invalid option is given or a NAME is read-only.
```

To see a list of all built-ins with a short description, use

```
help -d
```

Section 1.6: Hello World in "Debug" mode

```
$ cat hello.sh
#!/bin/bash
echo "Hello World"
$ bash -x hello.sh
+ echo Hello World
Hello World
```

The `-x` argument enables you to walk through each line in the script. One good example is [here](#):

```
$ cat hello.sh
#!/bin/bash
echo "Hello World\n"
```

```
adding_string_to_number="s"
v=$(expr 5 + $adding_string_to_number)
```

```
$ ./hello.sh
Hello World
```

```
expr: non-integer argument
```

The above prompted error is not enough to trace the script; however, using the following way gives you a better sense where to look for the error in the script.

```
$ bash -x hello.sh
+ echo Hello World\n
Hello World
```

```
+ adding_string_to_number=s
+ expr 5 + s
expr: non-integer argument
+ v=
```

Section 1.7: Handling Named Arguments

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

```
deploy=false
uglify=false
```

```
while (( $# > 1 )); do case $1 in
    --deploy) deploy="$2";;
    --uglify) uglify="$2";;
    *) break;
    esac; shift 2
done
```

```
$deploy && echo "will deploy... deploy = $deploy"
$uglify && echo "will uglify... uglify = $uglify"
```

```
# how to run
# chmod +x script.sh
# ./script.sh --deploy true --uglify false
```

Chapter 2: Script shebang

Section 2.1: Env shebang

To execute a script file with the **bash** executable found in the PATH environment variable by using the executable **env**, the **first line** of a script file must indicate the absolute path to the **env** executable with the argument **bash**:

```
#!/usr/bin/env bash
```

The **env** path in the shebang is resolved and used only if a script is directly launch like this:

```
script.sh
```

The script must have execution permission.

The shebang is ignored when a **bash** interpreter is explicitly indicated to execute a script:

```
bash script.sh
```

Section 2.2: Direct shebang

To execute a script file with the **bash** interpreter, the **first line** of a script file must indicate the absolute path to the **bash** executable to use:

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

The **bash** path in the shebang is resolved and used only if a script is directly launch like this:

```
./script.sh
```

The script must have execution permission.

The shebang is ignored when a **bash** interpreter is explicitly indicated to execute a script:

```
bash script.sh
```

Section 2.3: Other shebangs

There are two kinds of programs the kernel knows of. A binary program is identified by it's ELF (ExtenableLoadableFormat) header, which is usually produced by a compiler. The second one are scripts of any kind.

If a file starts in the very first line with the sequence **#!** then the next string has to be a pathname of an interpreter. If the kernel reads this line, it calls the interpreter named by this pathname and gives all of the following words in this line as arguments to the interpreter. If there is no file named "something" or "wrong":

```
#!/bin/bash something wrong  
echo "This line never gets printed"
```

bash tries to execute its argument "something wrong" which doesn't exist. The name of the script file is added too. To see this clearly use an **echo** shebang:

```
#"/bin/echo something wrong
# and now call this script named "thisscript" like so:
# thisscript one two
# the output will be:
something wrong ./thisscript one two
```

Some programs like **awk** use this technique to run longer scripts residing in a disk file.

Chapter 3: Navigating directories

Section 3.1: Absolute vs relative directories

To change to an absolutely specified directory, use the entire name, starting with a slash /, thus:

```
cd /home/username/project/abc
```

If you want to change to a directory near your current one, you can specify a relative location. For example, if you are already in `/home/username/project`, you can enter the subdirectory `abc` thus:

```
cd abc
```

If you want to go to the directory above the current directory, you can use the alias `..`. For example, if you were in `/home/username/project/abc` and wanted to go to `/home/username/project`, then you would do the following:

```
cd ..
```

This may also be called going "up" a directory.

Section 3.2: Change to the last directory

For the current shell, this takes you to the previous directory that you were in, no matter where it was.

```
cd -
```

Doing it multiple times effectively "toggles" you being in the current directory or the previous one.

Section 3.3: Change to the home directory

The default directory is the home directory (`$HOME`, typically `/home/username`), so `cd` without any directory takes you there

```
cd
```

Or you could be more explicit:

```
cd $HOME
```

A shortcut for the home directory is `~`, so that could be used as well.

```
cd ~
```

Section 3.4: Change to the Directory of the Script

In general, there are two types of Bash **scripts**:

1. System tools which operate from the current working directory
2. Project tools which modify files relative to their own place in the files system

For the second type of scripts, it is useful to change to the directory where the script is stored. This can be done with the following command:

```
cd "$(dirname "$(readlink -f "$0")")"
```

This command runs 3 commands:

1. `readlink -f "$0"` determines the path to the current script (\$0)
2. `dirname` converts the path to script to the path to its directory
3. `cd` changes the current work directory to the directory it receives from `dirname`