Bash Cheatsheet

A comprehensive cheat sheet for Bash scripting, covering essential syntax, commands, and best practices.

Getting Started with Bash

Basic Script Structure

Every Bash script starts with a shebang line, followed by commands.

```
#!/bin/bash

VAR="world"
echo "Hello $VAR!" # => Hello world!
```

To execute the script:

```
$ bash hello.sh
```

Variables

Defining variables:

```
NAME="John"
```

Accessing variables:

```
echo ${NAME}    # => John
echo $NAME      # => John
echo "$NAME"    # => John
echo '$NAME'    # => $NAME
echo "${NAME}!" # => John!
```

Important Note:

```
NAME = "John"   # => Error (about space)
```

Comments

Inline comments:

```
# This is an inline Bash comment.
```

Multi-line comments:

```
: '
This is a
very neat comment
in bash
'
```

Parameter Expansion and Arrays

Arrays

Defining arrays:

```
Fruits=('Apple' 'Banana' 'Orange')

Fruits[0]="Apple"
Fruits[1]="Banana"
Fruits[2]="Orange"

ARRAY1=(foo{1..2}) # => foo1 foo2
ARRAY2=({A..D})    # => A B C D
```

Indexing:

```
${Fruits[0]}    # First element
${Fruits[-1]}   # Last element
${Fruits[*]}    # All elements
${Fruits[@]}    # All elements
${#Fruits[@]}   # Number of all
${#Fruits}      # Length of 1st
${#Fruits[3]}  # Length of nth
${Fruits[@]:3:2} # Range
${!Fruits[@]}   # Keys of all
```

```
for i in "${!Fruits[@]}"; do
  printf "%s\t%s\n" "$i" "${Fruits[$i]}"
done
```

Array as arguments

```
function extract()
{
    local -n myarray=$1
    local idx=$2
    echo "${myarray[$idx]}"
}
Fruits=('Apple' 'Banana' 'Orange')
extract Fruits 2     # => Orangle
```

Bash Parameter Expansions

${FOO%suffix}	Remove suffix
${FOO#prefix}	Remove prefix
${FOO%%suffix}	Remove long suffix
${FOO##prefix}	Remove long prefix
${FOO/from/to}	Replace first match
${FOO//from/to}	Replace all
${FOO/%from/to}	Replace suffix
${FOO/#from/to}	Replace prefix

Array Operations

`Fruits=()`	Initialize an empty array.
`Fruits=("Apple" "Banana" "Cherry")`	Initialize an array with values.
`Fruits[0]="Apple"`	Assign value to array element at index 0.
`Fruits=("${Fruits[@]}" "Watermelon")`	Append “Watermelon” to the `Fruits` array.
`Fruits+=("Watermelon")`	Another way to append “Watermelon” to the `Fruits` array.
`Fruits=( ${Fruits[@]/Ap*/} )`	Remove elements matching the regex `Ap*` from the array (e.g., “Apple”).
`unset Fruits[2]`	Remove the element at index 2 from the `Fruits` array.
`Fruits=("${Fruits[@]}")`	Duplicate the `Fruits` array (creates a copy).
`Fruits=("${Fruits[@]}" "${Veggies[@]}")`	Concatenate the `Fruits` and `Veggies` arrays.
`lines=($(cat "logfile"))`	Read lines from “logfile” into the `lines` array.
`echo "${#Fruits[@]}"`	Get the number of elements in the `Fruits` array.

Arguments and Functions

Bash Functions

Defining a function

```
myfunc() {
  echo "hello $1"
}

# Alternate syntax:
function myfunc() {
  echo "hello $1"
}
```

Calling a function

```
myfunc "John"
```

Returning values from a function

```
myfunc() {
  local myresult='some value'
  echo $myresult
}

result="$(myfunc)"
```

Returning status codes

```
myfunc() {
  return 1
}

if myfunc; then
  echo "success"
else
  echo "failure"
fi
```

Local variables

Use `local` to declare variables within a function's scope.

```
myfunc() {
  local myvar="function scope"
  echo $myvar
}
```

Passing arguments

Arguments are accessed via `$1`, `$2`, etc. `$0` is the script name (or shell name, if interactive).

```
myfunc() {
  echo "First arg: $1, Second arg: $2"
}
myfunc "arg1" "arg2"
```

Accessing all arguments

`$@` expands to all arguments passed to the script or function.

```
myfunc() {
  echo "All args: $@"
}
myfunc "a" "b" "c"
```

Number of arguments

`$#` expands to the number of arguments passed to the script or function.

```
myfunc() {
  echo "Number of args: $#"
}
myfunc "x" "y"
```

Shift arguments

The `shift` command renumbers the arguments, effectively discarding `$1` and moving `$2` to `$1`, etc.

```
myfunc() {
  echo "First arg: $1"
  shift
  echo "New first arg: $1"
}
myfunc "a" "b"
```

Function scope

Functions are executed in the same shell environment as the caller. Variables declared outside the function are accessible inside unless shadowed by a `local` variable.

Recursion

Bash functions can be recursive, but recursion depth is limited.

```
countdown() {
  echo $1
  if [ $1 -gt 0 ]; then
    countdown $(( $1 - 1 ))
  fi
}
countdown 3
```

Unsetting a function

Use `unset -f function_name` to remove a function definition.

```
unset -f myfunc
```

Using function libraries

Source a file containing function definitions to make them available in your script.

```
. ./my_functions.sh
myfunc
```

Arguments

$1 … $9	Parameter 1 … 9
$0	Name of the script itself
$1	First argument
${10}	Positional parameter 10
$#	Number of arguments
$$	Process id of the shell
$*	All arguments
$@	All arguments, starting from first
$-	Current options
$_	Last argument of the previous command

Conditionals and Loops

Conditionals

Integer conditions:

```
[[ NUM -eq NUM ]] # Equal
[[ NUM -ne NUM ]] # Not equal
[[ NUM -lt NUM ]] # Less than
[[ NUM -le NUM ]] # Less than or equal
[[ NUM -gt NUM ]] # Greater than
[[ NUM -ge NUM ]] # Greater than or
equal
(( NUM < NUM ))  # Less than
(( NUM <= NUM )) # Less than or equal
(( NUM > NUM ))  # Greater than
(( NUM >= NUM )) # Greater than or equal
```

String conditions:

```
[[ -z STR ]]   # Empty string
[[ -n STR ]]   # Not empty string
[[ STR == STR ]] # Equal
[[ STR = STR ]]  # Equal (Same above)
[[ STR < STR ]]  # Less than (ASCII)
[[ STR > STR ]]  # Greater than (ASCII)
[[ STR != STR ]] # Not Equal
[[ STR =~ STR ]] # Regexp
```

File conditions:

```
[[ -e FILE ]] # Exists
[[ -d FILE ]] # Directory
[[ -f FILE ]] # File
[[ -h FILE ]] # Symlink
[[ -s FILE ]] # Size is > 0 bytes
[[ -r FILE ]] # Readable
[[ -w FILE ]] # Writable
[[ -x FILE ]] # Executable
[[ f1 -nt f2 ]] # f1 newer than f2
[[ f1 -ot f2 ]] # f2 older than f1
[[ f1 -ef f2 ]] # Same files
```

Bash Loops

Basic `for` loop

Iterates over a list of items.

```
for i in /etc/rc.*; do
  echo "$i"
done
```

C-like `for` loop

Uses arithmetic expressions for iteration.

```
for ((i = 0; i < 10; i++)); do
  echo "$i"
done
```

Ranges in `for` loop

Iterates over a sequence of numbers.

```
for i in {1..5}; do
  echo "Welcome $i"
done
```

`for` loop with step size

Specifies the increment value.

```
for i in {5..50..5}; do
  echo "Welcome $i"
done
```

Auto increment in `while` loop

Increments a variable in each iteration.

```
i=1
while [[ $i -lt 4 ]]; do
  echo "Number: $i"
  ((i++))
done
```

Auto decrement in `while` loop

Decrements a variable in each iteration.

```
i=3
while [[ $i -gt 0 ]]; do
  echo "Number: $i"
  ((i--))
done
```

`continue` statement

Skips the current iteration and proceeds to the next.

```
for number in $(seq 1 3); do
  if [[ $number == 2 ]]; then
    continue
  fi
  echo "$number"
done
```

`break` statement

Exits the loop entirely.

```
for number in $(seq 1 3); do
  if [[ $number == 2 ]]; then
    break
  fi
  echo "$number"
done
```

`until` loop

Executes a block of code until a condition is true.

```
count=0
until [ $count -gt 10 ]; do
  echo "$count"
  ((count++))
done
```

Infinite `while` loop	Loops indefinitely.
	`while true; do`
	`  # Code here`
	`done`
	
	Or shorthand:
	`while :; do`
	`  # Code here`
	`done`

Reading lines from a file	Iterates through each line of a file.
	`cat file.txt | while read line; do`
	`  echo "$line"`
	`done`

Nested Loops	Using loops inside another loop.
	`for i in {1..3}; do`
	`  for j in {a..c}; do`
	`  echo "$i-$j"`
	`  done`
	`done`

Looping through arrays	Iterating through elements of an array.
	`my_array=("apple" "banana" "cherry")`
	`for item in "${my_array[@]}"; do`
	`  echo "$item"`
	`done`

Using `select` loop	Creates a menu for user selection.
	`PS3='Please select an option: '`
	`select opt in "Option 1" "Option 2" "Option 3" "Quit"; do`
	`  case $opt in`
	`  "Option 1")`
	`  echo "Option 1 selected";; # Code to execute for option 1`
	`  "Option 2")`
	`  echo "Option 2 selected";; # Code to execute for option 2`
	`  "Option 3")`
	`  echo "Option 3 selected";; # Code to execute for option 3`
	`  "Quit")`
	`  break;;`
	`  *)`
	`  echo "Invalid option: $REPLY";; # Handle invalid option`
	`  esac`
	`done`

Bash Dictionaries (Associative Arrays)

Defining Dictionaries

Declare an associative array using `declare -A`. This is mandatory before using it as a dictionary.

```
declare -A my_dict
```

Assign key-value pairs individually:

```
my_dict[key1]=value1
my_dict[key2]=value2
```

Assign multiple key-value pairs at once:

```
my_dict=([key1]=value1 [key2]=value2)
```

Example:

```
declare -A sounds
sounds[dog]="bark"
sounds[cat]="meow"
```

Another Example:

```
declare -A colors=([red]="#FF0000"
[green]="#00FF00" [blue]="#0000FF")
```

Accessing Dictionary Elements

Access a value by its key:

```
echo ${my_dict[key1]}
```

If the key doesn't exist, an empty string is returned.

```
echo ${my_dict[nonexistent_key]} #
Outputs nothing
```

Example:

```
echo ${sounds[dog]}  # Outputs: bark
```

Using default values if a key doesn't exist:

```
echo ${my_dict[missing_key]:-
default_value}
```

Listing Keys and Values

Get all values:

```
echo ${my_dict[@]}
echo ${my_dict[*]}
```

Get all keys:

```
echo ${!my_dict[@]}
echo ${!my_dict[*]}
```

Example:

```
echo ${sounds[@]}   # Outputs: bark meow
echo ${!sounds[@]}  # Outputs: dog cat
```

When using `*`, values are separated by the first character of `$IFS` (usually space), while using `@` each value is treated as a separate word.

Dictionary Size and Existence Checks

Get the number of elements in the dictionary:

```
echo ${#my_dict[@]}
```

Check if a key exists:

```
if [[ -v my_dict[key1] ]]; then
  echo "Key exists"
else
  echo "Key does not exist"
fi
```

Example:

```
echo ${#sounds[@]}  # Outputs: 2
```

Deleting Elements

Delete a specific key-value pair:

```
unset my_dict[key1]
```

Delete the entire dictionary:

```
unset my_dict
```

Example:

```
unset sounds[dog]
echo ${sounds[@]} # Outputs: meow
```

Check if the dictionary is empty after deletion:

```
if [[ -z "${!my_dict[@]}" ]]; then
  echo "Dictionary is empty"
fi
```

Iterating Through Dictionaries

Iterate through values:

```
for val in "${my_dict[@]}"; do
  echo "Value: $val"
done
```

Iterate through keys:

```
for key in "${!my_dict[@]}"; do
  echo "Key: $key"
done
```

Iterate through both keys and values:

```
for key in "${!my_dict[@]}"; do
  value=${my_dict[$key]}
  echo "Key: $key, Value: $value"
done
```

Example:

```
for key in "${!sounds[@]}"; do
  echo "The ${key} says ${sounds[$key]}"
done
```

Using Dictionaries with Functions

Pass a dictionary to a function:

```
my_function() {
  local -n dict=$1  # Use nameref to
access the dictionary
  echo "First key: ${!dict[@]}"
}

declare -A example_dict=([a]=1 [b]=2)
my_function example_dict
```

Return a dictionary from a function (less common, usually modify in place using nameref):

```
return_dict() {
  declare -A local_dict=([x]=10 [y]=20)
  echo "local_dict=(${!local_dict[*]}
${local_dict[*]})"  # Output keys and
values
}

return_dict
eval $(return_dict) # Need eval to parse
correctly
echo ${local_dict[x]}
```

Advanced Usage: Configuration and Data Storage

Dictionaries can be used to store configuration settings:

```
declare -A config
config[timeout]=30
config[retries]=3

# Access settings
TIMEOUT=${config[timeout]}
RETRIES=${config[retries]}
```

Simulate data structures:

```
declare -A user
user[name]="John Doe"
user[email]="john.doe@example.com"
user[age]=30

echo "Name: ${user[name]}, Email:
${user[email]}, Age: ${user[age]}"
```

Loading configurations from a file:

```
while IFS='=' read -r key value; do
  config[$key]="$value"
done < config.txt

echo "Setting: ${config[setting_name]}"
```

Bash Miscellaneous

Numeric Calculations

`$((a + 200))` - Add 200 to `$a`

Example:

```
a=10
echo $((a + 200)) # Output: 210
```

`$(( $RANDOM % 200 ))` - Generate a random number between 0 and 199.

Example:

```
echo $(( $RANDOM % 200 )) # Output: A random number, e.g., 42
```

`$((a * b))` - Multiply `$a` and `$b`.

Example:

```
a=5
b=10
echo $((a * b)) # Output: 50
```

`$((a / b))` - Divide `$a` by `$b` (integer division).

Example:

```
a=20
b=3
echo $((a / b)) # Output: 6
```

`$((a - b))` - Subtract `$b` from `$a`.

Example:

```
a=15
b=7
echo $((a - b)) # Output: 8
```

Subshells

`(cd somedir; echo "I'm now in $PWD")` - Execute commands in a subshell.

Changes to the directory within the subshell do not affect the current shell's working directory.

Example:

```
mkdir tempdir
(cd tempdir; pwd)
pwd #Returns to the original directory
rmdir tempdir
```

`var=$(command)` - Assign output of command to variable.

Example:

```
var=$(ls -l)
echo "Listing: $var"
```

Using subshells for parallel execution (using `&` to run in background and `wait` to wait for completion).

Example:

```
(sleep 2; echo "Task 1 done") & (sleep 3; echo "Task 2 done") & wait
```

Inspecting Commands

`command -V cd` - Show information about the command `cd`.

This will tell you if `cd` is a built-in, alias, or external command.

Example:

```
command -V cd
#=&gt; cd is a shell builtin
```

`type command` - Display information about command type.

Example:

```
type ls
#=&gt; ls is aliased to `ls --color=auto`
```

`which command` - Locate the executable file associated with the command.

Example:

```
which bash
#=&gt; /usr/bin/bash
```

Redirection

`command > file` - Redirect standard output to a file.

Example:

```
echo "Hello, world!" > output.txt
```

`command >> file` - Append standard output to a file.

Example:

```
echo "Adding more text." >> output.txt
```

`command 2> file` - Redirect standard error to a file.

Example:

```
lscmd non_existent_file 2> error.log
```

`command 2>&1` - Redirect standard error to standard output.

Example:

```
lscmd non_existent_file 2>&1 | grep "No such file"
```

`command < file` - Redirect standard input from a file.

Example:

```
wc -l < input.txt
```

`command &>/dev/null` - Redirect both standard output and standard error to `/dev/null` (discarding them).

Example:

```
./my_script.sh &>/dev/null
```

Case/Switch Statements

Basic `case` statement structure:

```
case $variable in
  pattern1)
    commands1
    ;;
  pattern2)
    commands2
    ;;
  *)
    default_commands
    ;;
esac
```

Example using `case` for different options:

```
case $1 in
  start)
    echo "Starting service..."
    ;;
  stop)
    echo "Stopping service..."
    ;;
  restart)
    echo "Restarting service..."
    ;;
  *)
    echo "Invalid option. Usage: $0
{start|stop|restart}"
    ;;
esac
```

Using `case` with multiple patterns:

```
case $file_ext in
  jpg|jpeg)
    echo "It's a JPEG image"
    ;;
  png)
    echo "It's a PNG image"
    ;;
  *)
    echo "Unknown file type"
    ;;
esac
```

Trap Errors

`trap 'command' ERR` - Execute `command` when a script exits with a non-zero status.

Example:

```
trap 'echo "Error detected!"' ERR
false # Triggers the trap
```

Using `trap` to handle cleanup operations:

```
trap 'rm -f /tmp/tempfile' EXIT
echo "Creating temp file..."
touch /tmp/tempfile
# Rest of the script
```

Example with line number reporting:

```
trap 'echo "Error at line $LINENO"' ERR
false #Example of error that will
trigger the trap
```

Using a function to handle errors:

```
handle_error() {
  echo "ERROR: ${BASH_SOURCE[1]} at
about ${BASH_LINENO[0]}"
}
set -o errtrace
trap handle_error ERR
```

printf Formatting

`printf "Hello %s, I'm %s" Sven Olga` - Print formatted strings.

Example:

```
printf "Hello %s, I'm %s\n" Sven Olga
# Output: Hello Sven, I'm Olga
```

`printf "Integer: %d, Float: %f" 10 3.14` - Print integers and floats.

Example:

```
printf "Integer: %d, Float: %.2f\n" 10
3.14
# Output: Integer: 10, Float: 3.14
```

`printf "%s\n" "Line 1" "Line 2"` - Print multiple lines.

Example:

```
printf "%s\n" "First Line" "Second Line"
# Output:
# First Line
# Second Line
```

`printf "%10s" "text"` - Right-justify text within a field of width 10.

Example:

```
printf "%10s\n" "sample"
# Output:    sample
```

`printf "%-10s" "text"` - Left-justify text within a field of width 10.

Example:

```
printf "%-10s\n" "sample"
# Output: sample
```

Special Variables

`$?`	Exit status of the last executed command.
	
	Example:
	`ls -l`
	`echo $?`

`$!`	Process ID (PID) of the last background command.
	
	Example:
	`./long_running_script.sh &`
	`echo $!`

`$$`	PID of the current shell. Example: `echo $$`
`$0`	Filename of the current shell script. Example: `echo $0`
`$#`	Number of arguments passed to the script. Example: `echo $#`
`$@`	All arguments passed to the script (as separate words). Example: `echo $@`
`$*`	All arguments passed to the script (as a single word). Example: `echo $*`