

Cheat Sheets

PDF

- [bash_cheat_sheet.pdf](#)
- [bash-shell-scripting.pdf](#)

Bash Parameter

BASH PARAMETER EXPANSION

Parameter Expansion Syntax

Syntax	Description
<code>\${parameter%suffix}</code>	Remove suffix
<code>\${parameter#prefix}</code>	Remove prefix
<code>\${parameter%%suffix}</code>	Remove long suffix
<code>\${parameter##prefix}</code>	Remove long prefix
<code>\${parameter/pattern/string}</code>	Replace first pattern match with string
<code>\${parameter//pattern/string}</code>	Replace all the pattern with string
<code>\${parameter/%pattern/string}</code>	Replace suffix pattern with string
<code>\${parameter/#pattern/string}</code>	Replace prefix pattern with string

Default Values

<code>\${FOO:-val}</code>	Uses val if \$FOO is unset, without changing FOO
<code>\${FOO:=val}</code>	Sets \$FOO to val if unset, changing FOO
<code>\${FOO:+val}</code>	Uses val if \$FOO is set (does not change FOO)
<code>\${FOO:?message}</code>	Show message and exit if \$FOO is unset

Substrings

<code>\${FOO:0:3}</code>	Substring (<i>position</i> , <i>length</i>)
<code>\${FOO:~(-3):3}</code>	Substring from the right

Length

<code>\${#FOO}</code>	Length of \$FOO
-----------------------	-----------------

String Substitutions

```
1 food=           # => food is set to an empty
2                 # string
3
4 echo ${food:-Cake} # => Since $food is empty,
5                   # outputs "Cake"
6
7 STR="/path/to/foo.cpp"
8
9 echo ${STR%.cpp}  # => /path/to/foo
10 echo ${STR%.cpp}.o # => /path/to/foo.o
11 echo ${STR%/*}    # => /path/to
12
13 echo ${STR##*.}   # => cpp (extension)
14 echo ${STR##*/}   # => foo.cpp (basepath)
15
16 echo ${STR#*/}    # => path/to/foo.cpp
17 echo ${STR##*/}   # => foo.cpp
18
19 echo ${STR/foo/bar} # => /path/to/bar.cpp
```

String Slicing

```
1 name="John"
2
3 echo ${name}      # => John
4 echo ${name:0:2}  # => Jo
5 echo ${name::2}   # => Jo
6 echo ${name::-1}  # => Joh
7 echo ${name:~(-1)} # => n
8 echo ${name:~(-2)} # => hn
9 echo ${name:~(-2):2} # => hn
10
11 length=2
12 echo ${name:0:length} # => Jo
```

Basepath & Dirpath

```
1 SRC="/path/to/foo.cpp"
2
3 BASEPATH=${SRC##*/}
4 echo $BASEPATH      # => "foo.cpp"
5
6 DIRPATH=${SRC%$BASEPATH}
7 echo $DIRPATH       # => "/path/to/"
```

String Transformation

```
1 STR="HELLO WORLD!"
2
3 echo ${STR,,}      # => hello world!
4 echo ${STR^^}      # => HELLO WORLD!
5
6 STR="hello world!"
7
8 echo ${STR^}       # => Hello world!
9 echo ${STR^^}      # => HELLO WORLD!
10
11 ARR=(hello World)
12
13 echo "${ARR[@],}"  # => hello world
14 echo "${ARR[@]^}"  # => Hello World
```

sysxplore.com

Bash Loops

BASH SCRIPTING LOOPS BASICS

Bash for loop

```
1 for i in /etc/*; do
2     echo $i
3 done
4
5 # Same as above(alternate
6 syntax), also works with other
7 loop structs
8
9 for i in /etc/*
10 do
11     echo $i
12 done
```

C-like for loop

```
1 for ((i = 0 ; i < 100 ; i++)); do
2     echo $i
3 done
4
5 # Same as above (alternate
6 syntax) also works with other
7 loop structs
8
9 for ((i = 0 ; i < 100 ; i++))
10 do
11     echo $i
12 done
```

For loop ranges

```
1 for i in {1..10}; do
2     echo "Number: $i"
3 done
4
5 # With step size
6
7 # => {START..STOP..STEP}
8
9 for i in {5..50..5}; do
10     echo "Number: $i"
11 done
12
13 done
```

Bash while loop

```
1 # incrementing the value
2 i=1
3 while [[ $i -lt 4 ]]; do
4     echo "Number: $i"
5     ((i++))
6 done
7
8 # decrementing the value
9
10 i=3
11 while [[ $i -gt 0 ]]; do
12     echo "Number: $i"
13     ((i--))
14 done
```

Bash while True loop

```
1 # while true long hand
2
3 while true; do
4     # TODO
5     # TODO
6 done
7
8 # or the shorthand (alternate
9 syntax)
10
11 while ;; do
12     # TODO
13     # TODO
14 done
```

Reading files

```
1 # using pipes
2
3 cat file.txt | while read line
4 do
5     echo $line
6 done
7
8
9 # OR using input redirection
10
11
12 while read line; do
13     echo $line
14 done < "/path/to/txt/file"
```

Continue statement

```
1 # seq command can be used to
2 generate ranges
3
4 for number in $(seq 1 3); do
5
6     if [[ $number = 2 ]];
7     then
8         continue;
9     fi
10
11     echo "$number"
12
13 done
```

Break statement

```
1 for number in $(seq 1 3); do
2
3     if [[ $number = 2 ]]; then
4
5         # Skip entire rest of
6         loop or break out
7         of the loop.
8
9         break;
10
11     fi
12     # This will only print 1
13     echo "$number"
14 done
```

Until or do loop

```
1 # incrementing
2 count=0
3 until [ $count -gt 10 ]; do
4     echo "$count"
5     ((count++))
6 done
7
8 # decrementing
9 count=10
10 until [ $count -eq 0 ]; do
11     echo "$count"
12     ((count--))
13 done
```

sysxplore.com

Bash Basics

BASH SCRIPTING BASICS

```
#!/bin/bash
username="Jay"
filename=$3

read -p "Enter your username: " user
echo "Username: $user"

if [ "$EUID" -ne 0 ]; then
    echo "You are not running this script as the root user."
else
    echo "You are running this script as the root user."
fi

echo "Counting to 5:"
for i in {1..5}; do
    echo "$i"
done

function greet() {
    echo "Hello, $1!"
}
greet "Alice"

echo "Enter a number between 1 and 2: "
read num
case $num in
    1) echo "You chose one." ;;
    2) echo "You chose two." ;;
    *) echo "Invalid choice." ;;
esac

if [ -e "$filename" ] && [ -d "$filename" ]; then
    echo "File exists and is a directory."
else
    echo "File does not exist or is not a directory."
fi

echo "First argument: $1"
echo "Second argument: $2"

cat nonexistent-file.txt 2> /dev/null
echo "Exit status: $?"

fruits=("Apple" "Orange" "Banana")
echo "Fruits: ${fruits[0]}"

declare -A capitals
capitals[USA]="Washington D.C."
capitals[France]="Paris"
echo "Capital of France: ${capitals[France]}"

current_date=$(date)
echo "Today's date is: $current_date"

echo "This is a sample text." > example.txt
find / -name hello.txt &> /dev/null

result=$(( expr 15 - 2 ))
echo $result

SRC="/path/to/foo.cpp"
BASEPATH=${SRC##*/}
echo $BASEPATH

trap 'echo "Received SIGTERM signal. Cleaning up..."; exit' SIGTERM

# This is a single line comment
:' this a multiline
comment'
```

 bash script.sh

sysxplore.com

Bash Brackets

sysxplore.com | BASH BRACKETS: (), {}, \$(), [], [[]]

\$(commands)

Executes a command and captures its output. **Command substitution** allows the result of a command (in this case, `grep`) to be stored in a variable.

```
1 log_file="/var/log/syslog"
2 keyword="error"
3 output=$(grep "$keyword" "$log_file")
```

{ list; }

Executes a group of commands in the same **shell process**. Curly braces group commands together to be executed sequentially in the current environment.

```
1 { sudo apt install exa
2   echo exa
3   echo "Listed files using exa"; }
```

(a b c)

Creates an array of values. Parentheses are used to define an **array**, allowing multiple elements to be stored in one variable.

```
1 files=(log.txt log2.txt log3.txt)
2 for file in "${files[@]"; do
3   echo "Processing $file"
4 done
```

(list)

Executes a list of commands in a separate **subshell**. The commands inside the parentheses run in a child process, isolated from the main shell.

```
1 ( cd /home/user
2   ls
3   whoami )
```

{range}

Expands to multiple strings. Brace expansion is a powerful way to generate sequences or multiple strings, useful for **batch operations**. The range can be numbers or characters.

```
1 for file in backup_{1..4}.tar.gz; do
2   mv $file /var/oldbackups
3 done
```

\${expression}

Modifies variable content. **Parameter expansion** allows you to alter a variable's value, such as changing a file extension from `.txt` to `.bak`.

```
1 filename="report.txt"
2 backup_file="${filename%.txt}.bak"
3 echo "Backup file: $backup_file"
```

\${variable}

Accesses a variable's value. This is another way to **reference** a variable, commonly used when you need to follow it with additional characters or text.

```
1 username="John"
2 greeting="Hello, ${username}!"
3 echo "$greeting"
```

\$((expression))

Performs **arithmetic calculations**. The double parentheses are used for math operations, such as addition, multiplication, etc.

```
1 num1=5
2 num2=3
3 result=$((num1 * num2 + 1))
```

[expression]

Tests a condition using single brackets. The `[]` denotes a **test command** that checks conditions, such as whether a file exists.

```
1 file="/etc/passwd"
2 if [ -f "$file" ]; then
3   echo "File exists"
4 fi
```

[[expression]]

Tests a condition using double brackets. Double brackets are more flexible in Bash, supporting **advanced pattern** matching and logical operators.

```
1 user=$USER
2 if [[ $user = "root" ]]; then
3   echo "You are the root user"
4 fi
```

sysxplore.com

Revision #4

Created 11 November 2024 19:13:23 by Admin

Updated 1 April 2025 20:15:00 by Admin