Complete Bash Scripting Guide

1. Shell Variables

Introduction to Variables

Variables in bash follow C-like naming conventions. They store data that can be referenced and manipulated throughout your script.[1]

Basic Syntax:[2]

```
VAR=VALUE
```

Critical Rules:

- **No spaces** allowed before or after `=`
- Access value using `$VAR` or `${VAR}`
- Case-sensitive names
- Cannot start with digits

User-Defined Variables

Creating and Using Variables:[3]

```
$ MY_NAME=Foolan
$ echo $MY_NAME
Foolan

$ MY_FULL_NAME=Foolan Barik
Barik: command not found    # Error! Space without quotes

$ MY_FULL_NAME="Foolan Barik"
$ echo $MY_FULL_NAME
Foolan Barik
```

Deleting Variables:[4]

```
$ MY_NAME="Foolan"
$ echo $MY_NAME
Foolan

$ unset MY_NAME
$ echo $MY_NAME
                    # Empty - variable deleted
```

[1] bash.pdf
[2] bash.pdf
[3] bash.pdf
[4] bash.pdf

Understanding Quotes

Bash has three types of quotes with different behaviors:[5]

1. Double Quotes (") - Expand Variables

```
$ MYNAME="Foolan Barik"
$ echo "Welcome $MYNAME"
Welcome Foolan Barik

$ echo "Today is `date`"
Today is Tue Oct 28 19:39:00 IST 2025
```

2. Single Quotes (') - Literal Text (No Expansion)

```
$ echo 'Welcome $MYNAME'
Welcome $MYNAME

$ echo 'Today is `date`'
Today is `date`
```

3. Backticks (“‘) or$()‘ - Command Substitution

```
$ FILES=`ls /`
$ echo $FILES
bin boot dev etc home lib usr var

$ FILES=$(ls /)
$ echo $FILES
bin boot dev etc home lib usr var
```

Recommendation: Use `$()` instead of backticks for better nesting.[6]

Environment Variables

Bash provides many predefined environment variables:[7]

Common Environment Variables:

Variable	Description	Example
`$HOME`	Home directory	`/home/username`
`$PATH`	Command search paths	`/usr/bin:/bin:/usr/local/bin`
`$USER`	Current username	`foolan`
`$SHELL`	Current shell	`/bin/bash`
`$PWD`	Present working directory	`/home/foolan/documents`
`$HOSTNAME`	Machine name	`mycomputer`
`$RANDOM`	Random number (0-32767)	`15234`

[5] bash.pdf
[6] bash.pdf
[7] bash.pdf

Variable	Description	Example

Viewing All Variables:[8]

```
$ set
BASH=/bin/bash
HOME=/home/foolan
PATH=/usr/local/bin:/usr/bin:/bin
USER=foolan
...
```

Special Variables (Positional Parameters)

Built-in Special Variables:[9]

Variable	Meaning
`$0`	Script/command name
`$1, $2, ... $9`	Positional arguments 1-9
`${10}, ${11}, ...`	Arguments 10+ (use braces)
`$#`	Number of arguments
`$*`	All arguments as single string
`$@`	All arguments as separate strings
`$?`	Exit status of last command
`$$`	Current process ID
`$!`	Process ID of last background command

Example:[10]

```
$ parameters() {
> echo "Command: $0"
> echo "Number of arguments: $#"
> echo "All arguments (\$*): $*"
> echo "All arguments (\$@): $@"
> echo "First parameter: $1"
> echo "Second parameter: $2"
> }

$ parameters foolan barik kumar
Command: bash
Number of arguments: 3
All arguments ($*): foolan barik kumar
```

[8] bash.pdf
[9] bash.pdf
[10] bash.pdf

```
All arguments ($@): foolan barik kumar
First parameter: foolan
Second parameter: barik
```

Difference between $* and $@:

```
$ show_args() {
> for arg in "$*"; do
>   echo "Argument: $arg"
> done
> }

$ show_args one two three
Argument: one two three    # Treated as single argument

$ show_args2() {
> for arg in "$@"; do
>   echo "Argument: $arg"
> done
> }

$ show_args2 one two three
Argument: one
Argument: two
Argument: three            # Treated as separate arguments
```

Exporting Variables

Export variables to make them available to child processes:[11]

```
$ MYNAME=Foolan
$ bash                      # Start new shell
$ echo $MYNAME
                            # Variable not available

$ exit                      # Return to parent
$ export MYNAME
$ bash
$ echo $MYNAME
Foolan                      # Now available!
```

Combined Declaration and Export:[12]

```
$ export MY_NAME=Foolan
```

Checking Shell Level:[13]

[11] bash.pdf
[12] bash.pdf
[13] bash.pdf

```
$ echo $SHLVL
1
$ bash
$ echo $SHLVL
2
```

Reading User Input

Basic Input:[14]

```
$ echo -n "Enter your name: "
Enter your name: Foolan Barik
$ read MYNAME
$ echo $MYNAME
Foolan Barik
```

Inline Prompt:[15]

```
$ read -p "Enter your name: " MYNAME
Enter your name: Foolan Barik
$ echo $MYNAME
Foolan Barik
```

Multiple Variables:[16]

```
$ read -p "Enter first and last name: " FIRST LAST
Enter first and last name: Foolan Kumar Barik
$ echo $FIRST
Foolan
$ echo $LAST
Kumar Barik    # Extra words go to last variable
```

Silent Input (for passwords):

```
$ read -sp "Enter password: " PASSWORD
Enter password:
$ echo $PASSWORD
mypassword
```

Read-Only Variables

Make variables immutable using `declare -r`:[17]

```
$ MYNAME="Foolan Barik"
$ declare -r MYNAME
$ MYNAME="Someone Else"
```

[14] bash.pdf
[15] bash.pdf
[16] bash.pdf
[17] bash.pdf

```
bash: MYNAME: readonly variable

$ unset MYNAME
bash: unset: MYNAME: cannot unset: readonly variable
```

---

2. String Operations

String Length

Get the length of a string using `${#VAR}`:[18]

```
$ S="abcdefgh"
$ echo ${#S}
8

$ NAME="Foolan Barik"
$ echo "Name has ${#NAME} characters"
Name has 12 characters
```

Substring Extraction

From Position i to End:[19]

```
$ S="abcdefgh"
$ echo ${S:4}
efgh

$ echo ${S:0}
abcdefgh
```

Last i Characters (Note the space before minus):[20]

```
$ S="abcdefgh"
$ echo ${S: -4}
efgh

$ echo ${S: -1}
h
```

j Characters from Position i:[21]

```
$ S="abcdefgh"
$ echo ${S:4:4}
efgh
```

[18] bash.pdf
[19] bash.pdf
[20] bash.pdf
[21] bash.pdf

```
$ echo ${S:2:3}
cde
```

From i to j Positions from End:[22]

```
$ S="abcdefgh"
$ echo ${S:4:-2}
ef

$ echo ${S:2:-1}
bcdefg
```

String Concatenation

Simply place variables next to each other:[23]

```
$ S="abcdefgh"
$ T="ghijklmnop"
$ S="$S$T"
$ echo "$S has length ${#S}"
abcdefghghijklmnop has length 18

$ FIRST="Foolan"
$ LAST="Barik"
$ FULL="$FIRST $LAST"
$ echo $FULL
Foolan Barik
```

String Replacement

Replace First Occurrence:

```
$ S="hello world hello"
$ echo ${S/hello/hi}
hi world hello
```

Replace All Occurrences:

```
$ S="hello world hello"
$ echo ${S//hello/hi}
hi world hi
```

Remove Pattern:

```
$ FILENAME="document.txt"
$ echo ${FILENAME/.txt/}
document
```

[22] bash.pdf
[23] bash.pdf

```
$ PATH="/usr/local/bin:/usr/bin:/bin"
$ echo ${PATH//:/,}
/usr/local/bin,/usr/bin,/bin
```

String Case Conversion

Uppercase/Lowercase:

```
$ NAME="Foolan Barik"
$ echo ${NAME^^}
FOOLAN BARIK

$ echo ${NAME,,}
foolan barik

$ echo ${NAME^}
Foolan barik    # First character uppercase
```

Practical String Example

```
#!/bin/bash
# Extract filename components
FULLPATH="/home/user/documents/report.pdf"

# Get just filename
FILENAME="${FULLPATH##*/}"
echo "Filename: $FILENAME"           # report.pdf

# Get directory
DIR="${FULLPATH%/*}"
echo "Directory: $DIR"               # /home/user/documents

# Get extension
EXT="${FILENAME##*.}"
echo "Extension: $EXT"               # pdf

# Get name without extension
NAME="${FILENAME%.*}"
echo "Name: $NAME"                   # report
```

---

3. Arrays in Bash

Indexed Arrays

Declaring Arrays:[24]

```
$ declare -a MYARR
```

Setting Elements:[25]

```
$ MYARR[^0]="zero"
$ MYARR[^1]="one"
$ MYARR[^2]="two"
$ MYARR[^4]="four"    # Can skip indices
```

Quick Initialization:[26]

```
$ P=(2 3 5 7 11 13)
$ echo ${P[@]}
2 3 5 7 11 13

$ NAMES=("Alice" "Bob" "Charlie")
$ echo ${NAMES[^1]}
Bob
```

Accessing Array Elements

Individual Elements:[27]

```
$ MYARR[^0]="zero"
$ MYARR[^1]="one"
$ echo ${MYARR[^0]}
zero

$ echo ${MYARR[^1]}
one
```

All Elements:[28]

```
$ P=(2 3 5 7)
$ echo ${P[@]}
2 3 5 7

$ echo ${P[*]}
2 3 5 7
```

[24] bash.pdf
[25] bash.pdf
[26] bash.pdf
[27] bash.pdf
[28] bash.pdf

Array Indices:[29]

```
$ MYARR[^0]="zero"
$ MYARR[^1]="one"
$ MYARR[^2]="two"
$ MYARR[^4]="four"

$ echo ${!MYARR[@]}
0 1 2 4
```

Array Length:[30]

```
$ P=(2 3 5 7 11)
$ echo ${#P[@]}
5

$ MYARR[^0]="zero"; MYARR[^4]="four"
$ echo ${#MYARR[@]}
2
```

Array Operations

Appending Elements:[31]

```
$ P=(2 3 5 7)
$ P+=(11 13 17 19)
$ echo ${P[@]}
2 3 5 7 11 13 17 19

$ P[${#P[@]}]=23    # Append single element
$ echo ${P[@]}
2 3 5 7 11 13 17 19 23
```

Inserting at Position:[32]

```
$ P=(2 3 5 7 11 13 17 19 23 29 31 37)
$ P=(${P[@]:0:8} 21 23 29 ${P[@]:8})
$ echo ${P[@]}
2 3 5 7 11 13 17 19 21 23 29 23 29 31 37
```

Deleting Elements:[33]

```
$ P=(2 3 5 7 11 13 17 19 21 23 29 31 37)
$ unset P[^8]
$ echo ${P[@]}
```

[29] bash.pdf
[30] bash.pdf
[31] bash.pdf
[32] bash.pdf
[33] bash.pdf

```
2 3 5 7 11 13 17 19 23 29 31 37

$ echo ${!P[@]}
0 1 2 3 4 5 6 7 9 10 11 12    # Index 8 is missing
```

Compacting Array (Re-indexing):[34]

```
$ P=(${P[@]})
$ echo ${!P[@]}
0 1 2 3 4 5 6 7 8 9 10 11    # Continuous indices
```

Concatenating Arrays:[35]

```
$ P=(2 3 5 7 11 13)
$ Q=(17 19 23 29)
$ P=(${P[@]} ${Q[@]})
$ echo ${P[@]}
2 3 5 7 11 13 17 19 23 29
```

Array Slicing:[36]

```
$ P=(2 3 5 7 11 13 17 19 23)
$ echo ${P[@]:2:4}
5 7 11 13

$ echo ${P[@]:5}
13 17 19 23
```

Associative Arrays (Hashes)

Declaration and Initialization:[37]

```
$ declare -A MYINFO
$ MYINFO["name"]="Foolan Barik"
$ MYINFO["fname"]="Foolan"
$ MYINFO["lname"]="Barik"
$ MYINFO["cgpa"]="9.87"

$ echo ${MYINFO[fname]}
Foolan
```

Combined Declaration:[38]

```
$ declare -A MYINFO=(
>    ["name"]="Foolan Barik"
>    ["fname"]="Foolan"
```

[34] bash.pdf
[35] bash.pdf
[36] bash.pdf
[37] bash.pdf
[38] bash.pdf

```
>   ["lname"]="Barik"
>   ["cgpa"]="9.87"
>   ["height"]="5'08''"
> )

$ echo "${MYINFO[fname]} ${MYINFO[lname]}"
Foolan Barik
```

Accessing Keys:[39]

```
$ echo ${!MYINFO[@]}
fname height lname name cgpa

$ for key in ${!MYINFO[@]}; do
>   echo "$key: ${MYINFO[$key]}"
> done
fname: Foolan
height: 5'08''
lname: Barik
name: Foolan Barik
cgpa: 9.87
```

Practical Array Example

File: process_scores.sh

```
#!/bin/bash
# Process student scores

declare -a NAMES=("Alice" "Bob" "Charlie" "David" "Eve")
declare -a SCORES=(85 92 78 95 88)

echo "Student Scores:"
echo "---------------"

for i in ${!NAMES[@]}; do
    echo "${NAMES[$i]}: ${SCORES[$i]}"
done

# Calculate average
TOTAL=0
for score in ${SCORES[@]}; do
    TOTAL=$((TOTAL + score))
done

AVG=$((TOTAL / ${#SCORES[@]}))
```

[39] bash.pdf

```
echo "---------------"
echo "Average Score: $AVG"
```

Output:

```
Student Scores:
---------------
Alice: 85
Bob: 92
Charlie: 78
David: 95
Eve: 88
---------------
Average Score: 87
```

---

4. Arithmetic Operations

Integer Arithmetic

Use `$((...))` for arithmetic operations:[40]

Basic Operations:[41]

```
$ a=3
$ b=4
$ c=-5

$ echo $((a + b))
7

$ echo $((a + b * c - 6))
-23

$ z=$((a ** 2 + b ** 2))
$ echo $z
25
```

Operators:

Operator	Meaning	Example
`+`	Addition	`$((5 + 3))` → 8
`-`	Subtraction	`$((5 - 3))` → 2
`*`	Multiplication	`$((5 * 3))` → 15
`/`	Division	`$((10 / 3))` → 3

[40] bash.pdf
[41] bash.pdf

Operator	Meaning	Example
`%`	Modulo	`$((10 % 3))` → 1
`**`	Exponentiation	`$((2 ** 10))` → 1024

Note: Inside `$(( ))`, the `$` before variable names is optional:[42]

```
$ a=3; b=4
$ echo $((a + b))
7

$ echo $(($a + $b))
7
```

Increment and Decrement

Pre and Post Operations:

```
$ n=5
$ echo $((n++))
5
$ echo $n
6

$ echo $((++n))
7
$ echo $n
7

$ echo $((n--))
7
$ echo $n
6

$ echo $((--n))
5
```

Compound Assignment

```
$ a=10
$ ((a += 5))
$ echo $a
15

$ ((a *= 2))
```

[42] bash.pdf

```
$ echo $a
30

$ ((a /= 3))
$ echo $a
10
```

Comparison in Arithmetic Context

Use `(( ))` for numeric comparisons:

```
$ a=5
$ b=10
$ if ((a < b)); then
>   echo "a is less than b"
> fi
a is less than b

$ ((a > 3)) && echo "a is greater than 3"
a is greater than 3
```

Fibonacci Array Example

Computing Fibonacci Numbers:[43]

```
$ declare -a FIB=([^0]=0 [^1]=1)

$ n=2; FIB[$n]=$((FIB[n-1] + FIB[n-2]))
$ n=3; FIB[$n]=$((FIB[n-1] + FIB[n-2]))
$ n=4; FIB[$n]=$((FIB[n-1] + FIB[n-2]))
$ n=5; FIB[$n]=$((FIB[n-1] + FIB[n-2]))

$ echo ${FIB[@]}
0 1 1 2 3 5
```

Floating-Point Arithmetic

Bash doesn't support floating-point natively. Use `bc` calculator:[44]

Basic bc Usage:[45]

```
$ num=22
$ den=7
$ approxpi=`echo "$num / $den" | bc`
$ echo $approxpi
3
```

[43] bash.pdf
[44] bash.pdf
[45] bash.pdf

Setting Precision with scale:[46]

```
$ approxpi=`echo "scale=10; $num / $den" | bc`
$ echo $approxpi
3.1428571428

$ num=355
$ den=113
$ echo "scale=15; $num / $den" | bc
3.141592920353982
```

Mathematical Functions in bc:

```
# Square root
$ echo "scale=5; sqrt(2)" | bc
1.41421

# Power
$ echo "scale=5; 2^10" | bc
1024

# Sine (requires bc -l)
$ echo "scale=5; s(0)" | bc -l
0
```

Practical Arithmetic Example

File: calculator.sh

```
#!/bin/bash
# Simple calculator

read -p "Enter first number: " num1
read -p "Enter operator (+, -, *, /, %): " op
read -p "Enter second number: " num2

case $op in
    +) result=$((num1 + num2)) ;;
    -) result=$((num1 - num2)) ;;
    \*) result=$((num1 * num2)) ;;
    /)
        if [ $num2 -eq 0 ]; then
            echo "Error: Division by zero"
            exit 1
        fi
        result=$((num1 / num2))
```

[46] bash.pdf

```
        ;;
    %) result=$((num1 % num2)) ;;
    *)
        echo "Invalid operator"
        exit 1
        ;;
esac

echo "$num1 $op $num2 = $result"
```

---

5. Functions in Bash

Defining Functions

Two Syntaxes:[47]

```
# Syntax 1: With 'function' keyword
function FNAME() {
    commands
}

# Syntax 2: Without 'function' keyword
FNAME() {
    commands
}
```

Simple Example:[48]

```
$ function twopower() {
> echo "Usage: twopower exponent"
> echo "2 to the power $1 is $((2 ** $1))"
> }

$ twopower 10
Usage: twopower exponent
2 to the power 10 is 1024
```

Function Arguments

Functions access arguments like scripts using `$1`, `$2`, etc.:[49]

```
$ greet() {
> echo "Hello, $1!"
> if [ -n "$2" ]; then
```

[47] bash.pdf
[48] bash.pdf
[49] bash.pdf

```
>   echo "Welcome to $2"
> fi
> }

$ greet Foolan "Systems Programming"
Hello, Foolan!
Welcome to Systems Programming

$ greet Alice
Hello, Alice!
```

Return Values

Method 1: Using Global Variables[50]

```
#!/bin/bash
function hypotenuse() {
    local a=$1
    local b=$2
    a=$((a * a))
    b=$((b * b))
    csqr=$((a + b))    # Global variable
    c=`echo "scale=10; sqrt($csqr)" | bc`    # Global variable
}

read -p "Enter a and b: " a b
hypotenuse $a $b
echo "a = $a, b = $b, c = $c, csqr = $csqr"
```

Output:

```
$ ./hypo1.sh
Enter a and b: 5 6
a = 5, b = 6, c = 7.8102496759, csqr = 61
```

Method 2: Using Echo (Recommended)[51]

```
#!/bin/bash
function hypotenuse() {
    local a=$1
    local b=$2
    a=$((a * a))
    b=$((b * b))
    local csqr=$((a + b))
    echo `echo "scale=10; sqrt($csqr)" | bc`
}
```

[50] bash_1.pdf
[51] bash_1.pdf

```
read -p "Enter a and b: " a b
c=`hypotenuse $a $b`
echo "a = $a, b = $b, c = $c"
```

Output:

```
$ ./hypo2.sh
Enter a and b: 5 6
a = 5, b = 6, c = 7.8102496759
```

Method 3: Using return (Exit Codes Only)

```
$ is_even() {
> local num=$1
> if ((num % 2 == 0)); then
>   return 0    # Success/True
> else
>   return 1    # Failure/False
> fi
> }

$ is_even 4
$ echo $?
0

$ is_even 5
$ echo $?
1

$ if is_even 10; then
>   echo "Even number"
> fi
Even number
```

Variable Scope

Use `local` for function-local variables:[52]

```
$ x=3; y=4; z=5

$ fx() {
> local x=6
> echo "Inside fx: x = $x, y = $y, z = $z"
> }

$ fx
```

[52] bash.pdf

```
Inside fx: x = 6, y = 4, z = 5

$ echo "Outside: x = $x"
Outside: x = 3
```

Nested Function Calls:[53]

```
$ x=3; y=4; z=5

$ fx() {
> local x=6
> echo "In fx: x = $x, y = $y, z = $z"
> }

$ fxy() {
> local y=7
> local x=9
> echo "In fxy: x = $x, y = $y, z = $z"
> fx
> echo "Back in fxy: x = $x, y = $y"
> }

$ fxy
In fxy: x = 9, y = 7, z = 5
In fx: x = 6, y = 7, z = 5
Back in fxy: x = 9, y = 7

$ echo "Global: x = $x, y = $y, z = $z"
Global: x = 3, y = 4, z = 5
```

Practical Function Examples

Example 1: Factorial Calculator

```
#!/bin/bash
function factorial() {
    local n=$1
    if [ $n -le 1 ]; then
        echo 1
    else
        local prev=`factorial $((n - 1))`
        echo $((n * prev))
    fi
}

read -p "Enter a number: " num
```

[53] bash.pdf

```
result=`factorial $num`
echo "Factorial of $num is $result"
```

Example 2: String Reversal[54]

```
#!/bin/bash
function reverse() {
    local S=$1
    local Slen=${#S}
    local T

    case $Slen in
        0|1) echo "$S" ;;
        *)
            T=${S:0:-1}
            T=`reverse "$T"`
            echo "${S: -1}$T"
            ;;
    esac
}

read -p "Enter a string: " S
echo -n "reverse($S) = "
S=`reverse "$S"`
echo "$S"
```

Output:

```
$ ./reversal.sh
Enter a string: a bc def ghij klmno pqrstu
reverse(a bc def ghij klmno pqrstu) = utsrqp onmlk jihg fed cb a
```

Example 3: Checking Prime Numbers

```
#!/bin/bash
function is_prime() {
    local n=$1

    if [ $n -lt 2 ]; then
        return 1
    fi

    if [ $n -eq 2 ]; then
        return 0
    fi

    if [ $((n % 2)) -eq 0 ]; then
```

[54] bash_1.pdf

```
        return 1
    fi

    local i=3
    local limit=`echo "scale=0; sqrt($n)" | bc`

    while [ $i -le $limit ]; do
        if [ $((n % i)) -eq 0 ]; then
            return 1
        fi
        i=$((i + 2))
    done

    return 0
}

read -p "Enter a number: " num
if is_prime $num; then
    echo "$num is prime"
else
    echo "$num is not prime"
fi
```

---

6. Command Execution

Running Commands

Direct Execution:[55]

```
$ ls /
bin boot dev etc home lib usr var

$ date
Tue Oct 28 19:39:00 IST 2025

$ whoami
foolan
```

Command Substitution

Using Backticks:[56]

```
$ FILES=`ls /`
$ echo $FILES
```

[55] bash.pdf
[56] bash.pdf

```
bin boot dev etc home lib usr var

$ TODAY=`date +%Y-%m-%d`
$ echo $TODAY
2025-10-28
```

Using $() (Preferred):[57]

```
$ FILES=$(ls /)
$ echo $FILES
bin boot dev etc home lib usr var

$ USERS=$(who | wc -l)
$ echo "Number of users logged in: $USERS"
Number of users logged in: 3
```

Nested Command Substitution:

```
# With backticks (difficult to read)
$ echo `echo \`echo hello\``

# With $() (much clearer)
$ echo $(echo $(echo hello))
hello
```

Running Commands Non-Interactively

Using bash -c:[58]

```
$ echo $SHLVL
1

$ bash -c 'cal March 2023'
     March 2023
Su Mo Tu We Th Fr Sa
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31
```

Checking Command Success

Exit Status:[59]

[57] bash.pdf
[58] bash.pdf
[59] bash_1.pdf

```
$ ls /
bin boot dev etc home lib usr var
$ echo $?
0    # Success

$ ls /nonexistent
ls: cannot access '/nonexistent': No such file or directory
$ echo $?
2    # Failure
```

Logical Operators

AND (&&):

```
$ mkdir testdir && cd testdir
$ pwd
/home/user/testdir

$ false && echo "This won't print"
```

OR (||):

```
$ [ -f myfile.txt ] || echo "File doesn't exist"
File doesn't exist

$ grep "pattern" file.txt || echo "Pattern not found"
```

Combining:

```
$ command1 && echo "Success" || echo "Failure"
```

Background Processes

```
# Run in background
$ long_running_command &
[^1] 12345

# Check jobs
$ jobs
[^1]+ Running    long_running_command &

# Bring to foreground
$ fg %1

# Send to background again
# Press Ctrl+Z
$ bg %1
```

---

7. Shell Script Basics

Creating Shell Scripts

Basic Script Structure:[60]

```
#!/bin/bash
# This is a comment
# Script name: hello.sh

echo "Hello, world!"
```

Shebang Line:

- `#!/bin/bash` - Specifies the interpreter
- Must be the first line
- Makes the script executable as a standalone program

Making Scripts Executable:[61]

```
$ chmod 755 hello.sh
# or
$ chmod +x hello.sh

$ ./hello.sh
Hello, world!
```

Script with Arguments

File: greet.sh

```
#!/bin/bash
# Greet a user

if [ $# -eq 0 ]; then
    echo "Usage: $0 NAME"
    exit 1
fi

echo "Hello, $1!"
echo "Welcome to bash scripting."
```

Running:

```
$ ./greet.sh
Usage: ./greet.sh NAME

$ ./greet.sh Foolan
```

[60] bash_1.pdf
[61] bash_1.pdf

```
Hello, Foolan!
Welcome to bash scripting.
```

Multi-Argument Scripts

File: sum.sh[62]

```bash
#!/bin/bash
# Sum multiple numbers

if [ $# -eq 0 ]; then
    echo "Usage: $0 num1 num2 ..."
    exit 1
fi

sum=0
for num in "$@"; do
    sum=$((sum + num))
done

echo "Sum of $# numbers: $sum"
```

Running:

```
$ ./sum.sh 10 20 30 40
Sum of 4 numbers: 100
```

Exit Codes

Setting Exit Codes:

```bash
#!/bin/bash
# Check if file exists

if [ ! -f "$1" ]; then
    echo "Error: File not found"
    exit 1    # Error exit code
fi

echo "File exists"
exit 0    # Success exit code
```

Standard Exit Codes:

- 0 - Success
- 1 - General error
- 2 - Misuse of shell command
- 126 - Command cannot execute

[62] bash_1.pdf

- 127 - Command not found
- 130 - Script terminated by Ctrl+C

Script Template

File: template.sh

```
#!/bin/bash
################################################################################
# Script Name: template.sh
# Description: Template for bash scripts
# Author: Your Name
# Date: 2025-10-28
# Usage: ./template.sh [options] arguments
################################################################################

# Exit on error
set -e

# Exit on undefined variable
set -u

# Constants
readonly SCRIPT_NAME=$(basename "$0")
readonly SCRIPT_DIR=$(dirname "$0")

# Functions
function usage() {
    echo "Usage: $SCRIPT_NAME [OPTIONS] ARGS"
    echo ""
    echo "Options:"
    echo "  -h, --help    Show this help message"
    echo "  -v, --verbose Verbose output"
    exit 0
}

function error_exit() {
    echo "ERROR: $1" >&2
    exit 1
}

# Parse command-line arguments
VERBOSE=0

while [ $# -gt 0 ]; do
    case $1 in
```

```
        -h|--help)
            usage
            ;;
        -v|--verbose)
            VERBOSE=1
            shift
            ;;
        *)
            echo "Unknown option: $1"
            usage
            ;;
    esac
done

# Main script logic
if [ $VERBOSE -eq 1 ]; then
    echo "Running in verbose mode"
fi

echo "Script started successfully"

exit 0
```

---

8. Interactive Scripts

Reading User Input

Simple Input:[63]

```
#!/bin/bash
echo -n "Enter your name: "
read name
echo "Hello, $name!"
```

Using read with prompt:

```
#!/bin/bash
read -p "Enter your age: " age
echo "You are $age years old"
```

Menu-Based Scripts

File: menu.sh

```
#!/bin/bash
# Interactive menu
```

[63] bash_1.pdf

```
while true; do
    echo ""
    echo "===== Main Menu ====="
    echo "1. List files"
    echo "2. Show date"
    echo "3. Show current directory"
    echo "4. Exit"
    echo "====================="
    read -p "Enter your choice [1-4]: " choice

    case $choice in
        1)
            echo "Files in current directory:"
            ls -l
            ;;
        2)
            echo "Current date and time:"
            date
            ;;
        3)
            echo "Current directory:"
            pwd
            ;;
        4)
            echo "Goodbye!"
            exit 0
            ;;
        *)
            echo "Invalid choice. Please try again."
            ;;
    esac

    read -p "Press Enter to continue..."
done
```

File Finder Script

File: findall.sh[64]

```
#!/bin/bash
# Find all files with given extension

echo -n "*** Enter an extension (without the dot): "
read extn
```

[64] bash_1.pdf

```
echo "*** Okay, finding all files in your home area with extension $extn"
ls -R ~ | grep "\.$extn$"

echo "*** That's all you have. Bye."
```

Running:[65]

```
$ ./findall.sh
*** Enter an extension (without the dot): tif
*** Okay, finding all files in your home area with extension tif
centralimage-1500.tif
formulas-hires.tif
Crypto.tif
*** That's all you have. Bye.
```

Confirmation Prompts

File: confirm.sh

```
#!/bin/bash
# Delete file with confirmation

if [ $# -eq 0 ]; then
    echo "Usage: $0 filename"
    exit 1
fi

filename=$1

if [ ! -f "$filename" ]; then
    echo "Error: File '$filename' not found"
    exit 1
fi

read -p "Are you sure you want to delete '$filename'? (y/n): " confirm

case $confirm in
    [yY]|[yY][eE][sS])
        rm "$filename"
        echo "File deleted successfully"
        ;;
    [nN]|[nN][oO])
        echo "Operation cancelled"
        ;;
    *)
```

[65] bash_1.pdf

```
        echo "Invalid response. Operation cancelled"
        ;;
esac
```

Password Input

File: login.sh

```
#!/bin/bash
# Simple password check

CORRECT_PASSWORD="secret123"

read -p "Username: " username
read -sp "Password: " password
echo ""

if [ "$password" = "$CORRECT_PASSWORD" ]; then
    echo "Login successful! Welcome, $username"
else
    echo "Login failed!"
    exit 1
fi
```

Multi-Field Form

File: registration.sh

```
#!/bin/bash
# User registration form

echo "===== User Registration ====="
read -p "First Name: " firstname
read -p "Last Name: " lastname
read -p "Email: " email
read -p "Age: " age
read -sp "Password: " password
echo ""
read -sp "Confirm Password: " password2
echo ""

if [ "$password" != "$password2" ]; then
    echo "Error: Passwords don't match"
    exit 1
fi

echo ""
```

```
echo "===== Registration Summary ====="
echo "Name: $firstname $lastname"
echo "Email: $email"
echo "Age: $age"
echo ""
read -p "Confirm registration? (y/n): " confirm

if [[ $confirm =~ ^[Yy]$ ]]; then
    echo "Registration successful!"
    # Save to file or database here
else
    echo "Registration cancelled"
fi
```

---

9. Running Other Programs

Executing External Commands

Direct Execution:

```
#!/bin/bash
# Run external programs

# List files
ls -la

# Show disk usage
df -h

# Show processes
ps aux | head -n 10
```

Capturing Command Output

Storing in Variables:

```
#!/bin/bash
# Capture output

FILES=$(ls *.txt)
echo "Text files: $FILES"

USER_COUNT=$(who | wc -l)
echo "Users logged in: $USER_COUNT"
```

```
DISK_FREE=$(df -h / | awk 'NR==2 {print $4}')
echo "Free disk space: $DISK_FREE"
```

Checking if Commands Exist

```
#!/bin/bash
# Check if command exists

if command -v git &> /dev/null; then
    echo "Git is installed"
    git --version
else
    echo "Git is not installed"
fi

# Alternative method
if which python3 > /dev/null 2>&1; then
    echo "Python3 is available"
else
    echo "Python3 is not available"
fi
```

Running with Specific Environment

```
#!/bin/bash
# Run command with custom environment

# Set temporary environment variable
export LANG=C
date

# Run command in modified PATH
PATH="/custom/path:$PATH" mycommand

# Run with clean environment
env -i HOME="$HOME" bash -c 'echo $PATH'
```

Timeout for Commands

```
#!/bin/bash
# Run command with timeout

timeout 5s long_running_command
if [ $? -eq 124 ]; then
    echo "Command timed out"
fi
```

Piping Between Programs

```
#!/bin/bash
# Chain commands

# Count .txt files
ls *.txt | wc -l

# Find and sort
find . -name "*.log" | sort | head -n 10

# Process CSV
cat data.csv | grep "pattern" | cut -d',' -f1,3 | sort -u
```

Running Programs in Parallel

```
#!/bin/bash
# Run multiple commands in parallel

command1 &
PID1=$!

command2 &
PID2=$!

command3 &
PID3=$!

# Wait for all to complete
wait $PID1 $PID2 $PID3

echo "All commands completed"
```

Compiling and Running C Programs

```
#!/bin/bash
# Compile and run C program

SOURCE="program.c"
BINARY="program"

if [ ! -f "$SOURCE" ]; then
    echo "Error: Source file not found"
    exit 1
fi

echo "Compiling $SOURCE..."
```

```
gcc -Wall -g -o "$BINARY" "$SOURCE"

if [ $? -ne 0 ]; then
    echo "Compilation failed"
    exit 1
fi

echo "Compilation successful"
echo "Running $BINARY..."
./"$BINARY"
```

---

10. Conditionals

if Statements

Basic Syntax:[66]

```
if condition; then
    commands
fi
```

With else:[67]

```
if condition; then
    commands1
else
    commands2
fi
```

With elif:[68]

```
if condition1; then
    commands1
elif condition2; then
    commands2
else
    commands3
fi
```

Numeric Comparisons

Test Operators:[69]

---

[66] bash_1.pdf
[67] bash_1.pdf
[68] bash_1.pdf
[69] bash_1.pdf

Operator	Meaning
`-eq`	Equal to
`-ne`	Not equal to
`-lt`	Less than
`-le`	Less than or equal
`-gt`	Greater than
`-ge`	Greater than or equal

Examples:[70]

```
$ x=3; y=4; z=5

$ if [ $y -gt $x ]; then
>   echo "$y is greater than $x"
> fi
4 is greater than $x

$ if [ $((x**2 + y**2)) -eq $((z**2)) ]; then
>   echo "Pythagorean triple!"
> fi
Pythagorean triple!
```

Arithmetic Context (Preferred for Numbers):

```
#!/bin/bash
a=10
b=20

if ((a < b)); then
    echo "a is less than b"
fi

if ((a > 5 && a < 15)); then
    echo "a is between 5 and 15"
fi
```

String Comparisons

Operators:[71]

Operator	Meaning
`=` or `==`	Strings are equal
`!=`	Strings are not equal

[70] bash_1.pdf
[71] bash_1.pdf

Operator	Meaning
<	Less than (alphabetically)
>	Greater than (alphabetically)
-z	String is empty
-n	String is not empty

Examples:[72]

```
$ x="Foolan"
$ y="Foolan Barik"

$ if [ "$x" == "$y" ]; then
>   echo "Same"
> else
>   echo "Different"
> fi
Different

$ if [ -z "$z" ]; then
>   echo "Variable z is empty"
> fi
Variable z is empty

$ if [ -n "$x" ]; then
>   echo "Variable x is not empty"
> fi
Variable x is not empty
```

Pattern Matching:

```
#!/bin/bash
filename="document.pdf"

if [[ $filename == *.pdf ]]; then
    echo "PDF file detected"
fi

if [[ $filename =~ ^[a-z]+\. ]]; then
    echo "Starts with lowercase letters"
fi
```

File Tests

Common File Tests:[73]

[72] bash_1.pdf
[73] bash_1.pdf

Test	Meaning
`-e FILE`	File exists
`-f FILE`	Regular file exists
`-d FILE`	Directory exists
`-L FILE`	Symbolic link exists
`-r FILE`	File is readable
`-w FILE`	File is writable
`-x FILE`	File is executable
`-s FILE`	File exists and is not empty
`-N FILE`	File modified since last read
`FILE1 -nt FILE2`	FILE1 is newer than FILE2
`FILE1 -ot FILE2`	FILE1 is older than FILE2

Example Script:[74]

```
#!/bin/bash
# File attribute checker

if [ -z "$1" ]; then
    echo "Usage: $0 filename"
    exit 1
fi

fname=$1

if [ -e "$fname" ]; then
    echo "\"$fname\" exists"
else
    echo "\"$fname\" does not exist"
    exit 0
fi

if [ -f "$fname" ]; then
    echo "\"$fname\" is a regular file"
fi

if [ -d "$fname" ]; then
    echo "\"$fname\" is a directory"
fi

if [ -L "$fname" ]; then
    echo "\"$fname\" is a symbolic link"
fi
```

[74] bash_1.pdf

```
echo -n "Permissions:"
[ -r "$fname" ] && echo -n " read"
[ -w "$fname" ] && echo -n " write"
[ -x "$fname" ] && echo -n " execute"
echo ""

if [ -s "$fname" ]; then
    size=$(stat -f%z "$fname" 2>/dev/null || stat -c%s "$fname")
    echo "File size: $size bytes"
else
    echo "File is empty"
fi
```

Logical Operators

AND (`&&` or `-a`):

```
$ x=5
$ if [ $x -gt 0 ] && [ $x -lt 10 ]; then
>   echo "x is between 0 and 10"
> fi
x is between 0 and 10

# Alternative
$ if [ $x -gt 0 -a $x -lt 10 ]; then
>   echo "x is between 0 and 10"
> fi
```

OR (`||` or `-o`):

```
$ x=15
$ if [ $x -lt 0 ] || [ $x -gt 10 ]; then
>   echo "x is outside 0-10 range"
> fi
x is outside 0-10 range
```

NOT (`!`):

```
$ if [ ! -f "nonexistent.txt" ]; then
>   echo "File does not exist"
> fi
File does not exist
```

case Statements

Syntax:[75]

[75] bash_1.pdf

```
case value in
    pattern1)
        commands
        ;;
    pattern2)
        commands
        ;;
    *)
        default_commands
        ;;
esac
```

Example:[76]

```
#!/bin/bash
# File extension checker

if [ $# -eq 0 ]; then
    echo "Usage: $0 filename"
    exit 1
fi

filename=$1

case $filename in
    *.txt)
        echo "Text file"
        ;;
    *.jpg|*.jpeg|*.png|*.gif)
        echo "Image file"
        ;;
    *.pdf)
        echo "PDF document"
        ;;
    *.sh)
        echo "Shell script"
        ;;
    *)
        echo "Unknown file type"
        ;;
esac
```

Pattern Matching in case:

```
#!/bin/bash
# Grade calculator
```

[76] bash_1.pdf

```
read -p "Enter your score: " score

case $score in
    [^9][0-9]|100)
        echo "Grade: A"
        ;;
    [^8][0-9])
        echo "Grade: B"
        ;;
    [^7][0-9])
        echo "Grade: C"
        ;;
    [^6][0-9])
        echo "Grade: D"
        ;;
    *)
        echo "Grade: F"
        ;;
esac
```

Practical Conditional Examples

Example: File Backup Script

```
#!/bin/bash
# Backup file with checks

if [ $# -ne 2 ]; then
    echo "Usage: $0 source_file backup_dir"
    exit 1
fi

source=$1
backup_dir=$2

if [ ! -f "$source" ]; then
    echo "Error: Source file doesn't exist"
    exit 1
fi

if [ ! -d "$backup_dir" ]; then
    echo "Backup directory doesn't exist. Creating..."
    mkdir -p "$backup_dir"
fi
```

```
backup_name="$backup_dir/$(basename "$source").$(date +%Y%m%d_%H%M%S)"

if cp "$source" "$backup_name"; then
    echo "Backup created: $backup_name"
else
    echo "Error: Backup failed"
    exit 1
fi
```

---

11. Loops

for Loops

List-Based for Loop:[77]

```
# Simple list
for item in one two three four five; do
    echo $item
done

# Brace expansion
for n in {1..10}; do
    echo $n
done

# Step with brace expansion
for n in {0..100..10}; do
    echo $n
done
```

Iterating Over Arguments:[78]

```
#!/bin/bash
# Process all arguments

for arg in "$@"; do
    echo "Processing: $arg"
done

# Shorter form (implicit)
for arg; do
    echo "Processing: $arg"
done
```

[77] bash_1.pdf
[78] bash_1.pdf

Iterating Over Files:

```
#!/bin/bash
# Process all .txt files

for file in *.txt; do
    if [ -f "$file" ]; then
        echo "Processing $file"
        wc -l "$file"
    fi
done
```

C-Style for Loop:[79]

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

# Multiple variables
for (( i=0, j=10; i<10; i++, j-- )); do
    echo "i=$i, j=$j"
done
```

while Loops

Basic while Loop:[80]

```
n=0
while [ $n -lt 10 ]; do
    echo $n
    n=$((n + 1))
done
```

Arithmetic Condition:[81]

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

Reading from File:

```
#!/bin/bash
# Read file line by line

while read -r line; do
```

[79] bash_1.pdf
[80] bash_1.pdf
[81] bash_1.pdf

```
    echo "Line: $line"
done < input.txt
```

Infinite Loop:

```
while true; do
    echo "Running..."
    sleep 1
done

# Or
while :; do
    echo "Running..."
    sleep 1
done
```

until Loops

Basic until Loop:[82]

```
n=0
until [ $n -eq 10 ]; do
    echo $n
    n=$((n + 1))
done
```

Arithmetic Condition:

```
i=0
until (( i >= 10 )); do
    echo $i
    (( i++ ))
done
```

Loop Control

break - Exit Loop:[83]

```
for i in {1..100}; do
    if [ $i -eq 50 ]; then
        echo "Breaking at $i"
        break
    fi
    echo $i
done
```

continue - Skip Iteration:

[82] bash_1.pdf
[83] bash_1.pdf

```
for i in {1..10}; do
    if [ $((i % 2)) -eq 0 ]; then
        continue    # Skip even numbers
    fi
    echo $i
done
```

Nested Loops with break:

```
#!/bin/bash
# Break out of nested loop

found=0
for i in {1..10}; do
    for j in {1..10}; do
        if [ $((i * j)) -eq 24 ]; then
            echo "Found: $i x $j = 24"
            found=1
            break 2    # Break out of both loops
        fi
    done
    if [ $found -eq 1 ]; then
        break
    fi
done
```

Practical Loop Examples

Example 1: Fibonacci Calculator[84]

```
#!/bin/bash
# Compute Fibonacci numbers iteratively

function computerest() {
    local n=$1
    while [ $n -le $2 ]; do
        F[$n]=$((F[n-1] + F[n-2]))
        n=$((n + 1))
    done
}

declare -ia F=([^0]=0 [^1]=1)
N=1

while true; do
    read -p "Enter n: " n
```

[84] bash_1.pdf

```
    if [ $n -lt 0 ]; then
        echo "Enter a positive integer please"
        continue
    fi

    if [ $n -gt $N ]; then
        echo "Computing F($((N+1))) through F($n)"
        computerest $((N+1)) $n
        N=$n
    fi

    echo "F($n) = ${F[$n]}"

    read -p "Repeat (y/n)? " resp
    case $resp in
        [nN]*) echo "Bye..."; exit 0 ;;
    esac
done
```

Example 2: Sum Positive and Negative Separately[85]

```
#!/bin/bash
# Sum positive and negative integers separately

if [ $# -eq 0 ]; then
    echo "Usage: $0 num1 num2 ..."
    exit 1
fi

pos_sum=0
neg_sum=0

for num in "$@"; do
    if [ $num -gt 0 ]; then
        pos_sum=$((pos_sum + num))
    elif [ $num -lt 0 ]; then
        neg_sum=$((neg_sum + num))
    fi
done

echo "Sum of positive numbers: $pos_sum"
echo "Sum of negative numbers: $neg_sum"
```

Example 3: Directory Tree Explorer[86]

[85] bash_1.pdf
[86] bash_1.pdf

```bash
#!/bin/bash
# Explore directory tree recursively

function exploredir() {
    local currentdir=$1
    local currentlev=$2
    local lev=0

    # Print indentation
    while [ $lev -lt $currentlev ]; do
        echo -n "    "
        lev=$((lev + 1))
    done

    echo -n "$currentdir"

    if [ ! -r "$currentdir" ] || [ ! -x "$currentdir" ]; then
        echo " [Unable to explore further]"
    else
        echo ""
        for entry in "$currentdir"/*; do
            if [ -d "$entry" ]; then
                exploredir "$entry" $((currentlev + 1))
            fi
        done
    fi
}

if [ $# -eq 0 ]; then
    rootdir="."
else
    rootdir="$1"
fi

if [ ! -d "$rootdir" ]; then
    echo "$rootdir is not a directory"
    exit 1
fi

exploredir "$rootdir" 0
```

Example 4: Find Files Modified in Last N Days

```bash
#!/bin/bash
# Find files modified in last N days

if [ $# -ne 2 ]; then
```

```
    echo "Usage: $0 directory days"
    exit 1
fi

dir=$1
days=$2

if [ ! -d "$dir" ]; then
    echo "Error: $dir is not a directory"
    exit 1
fi

echo "Files modified in last $days days:"
find "$dir" -type f -mtime -$days | while read -r file; do
    echo "  $file"
done
```

---

12. File Processing in Bash

Reading Files Line by Line

Method 1: Using while read[87]

```
#!/bin/bash
# Read file into array

if [ $# -eq 0 ]; then
    echo "Usage: $0 filename"
    exit 1
fi

fname=$1

if [ ! -f "$fname" ] || [ ! -r "$fname" ]; then
    echo "Cannot read file: $fname"
    exit 1
fi

echo -n "Reading file $fname: "
L=()
while read -r line; do
    L+=("$line")
done < "$fname"
```

[87] bash_1.pdf

```
echo "${#L[@]} lines read"

# Print lines
for i in "${!L[@]}"; do
    echo "Line $((i+1)): ${L[$i]}"
done
```

Method 2: Using mapfile/readarray

```
#!/bin/bash
# Read file using mapfile

mapfile -t lines < filename.txt

for line in "${lines[@]}"; do
    echo "$line"
done
```

Processing Multiple Files

File: file2array.sh[88]

```
#!/bin/bash
# Read multiple files

if [ $# -lt 1 ]; then
    echo "Usage: $0 file1 [file2 ...]"
    exit 1
fi

for fname in "$@"; do
    if [ ! -f "$fname" ] || [ ! -r "$fname" ]; then
        echo "--- Unable to read $fname"
        continue
    fi

    echo -n "+++ Reading file $fname: "
    L=()
    while read -r line; do
        L+=("$line")
    done < "$fname"

    echo "${#L[@]} lines read"
done
```

[88] bash_1.pdf

File Statistics

```
#!/bin/bash
# Analyze text file

if [ $# -eq 0 ]; then
    echo "Usage: $0 filename"
    exit 1
fi

file=$1

if [ ! -f "$file" ]; then
    echo "File not found: $file"
    exit 1
fi

lines=$(wc -l < "$file")
words=$(wc -w < "$file")
chars=$(wc -c < "$file")

echo "File: $file"
echo "Lines: $lines"
echo "Words: $words"
echo "Characters: $chars"
```

CSV File Processing

```
#!/bin/bash
# Process CSV file

csv_file="data.csv"

# Skip header and process data
tail -n +2 "$csv_file" | while IFS=',' read -r name age city; do
    echo "Name: $name, Age: $age, City: $city"
done
```

File Searching and Filtering

```
#!/bin/bash
# Search for pattern in files

if [ $# -lt 2 ]; then
    echo "Usage: $0 pattern file1 [file2 ...]"
    exit 1
fi
```

```
pattern=$1
shift

for file in "$@"; do
    if [ -f "$file" ]; then
        count=$(grep -c "$pattern" "$file")
        echo "$file: $count matches"
    fi
done
```

File Backup Script

```
#!/bin/bash
# Backup files with specific extension

if [ $# -ne 2 ]; then
    echo "Usage: $0 extension backup_dir"
    exit 1
fi

ext=$1
backup_dir=$2

mkdir -p "$backup_dir"

for file in *."$ext"; do
    if [ -f "$file" ]; then
        cp "$file" "$backup_dir/"
        echo "Backed up: $file"
    fi
done

echo "Backup complete to $backup_dir"
```

Log File Analysis

```
#!/bin/bash
# Analyze log file

logfile="/var/log/syslog"

if [ ! -r "$logfile" ]; then
    echo "Cannot read log file"
    exit 1
fi
```

```
echo "=== Log Analysis ==="
echo "Total lines: $(wc -l < "$logfile")"
echo "Error count: $(grep -c ERROR "$logfile")"
echo "Warning count: $(grep -c WARNING "$logfile")"
echo ""
echo "Recent errors:"
grep ERROR "$logfile" | tail -n 5
```

File Comparison

```
#!/bin/bash
# Compare two files

if [ $# -ne 2 ]; then
    echo "Usage: $0 file1 file2"
    exit 1
fi

file1=$1
file2=$2

if [ ! -f "$file1" ] || [ ! -f "$file2" ]; then
    echo "One or both files not found"
    exit 1
fi

if cmp -s "$file1" "$file2"; then
    echo "Files are identical"
else
    echo "Files are different"
    echo ""
    echo "Differences:"
    diff "$file1" "$file2"
fi
```

Find and Process Files

```
#!/bin/bash
# Find files and process them

# Find all .txt files and count lines
find . -name "*.txt" -type f | while read -r file; do
    lines=$(wc -l < "$file")
    echo "$file: $lines lines"
done
```

```
# Find large files
echo ""
echo "Files larger than 1MB:"
find . -type f -size +1M -exec ls -lh {} \; | awk '{print $9, $5}'
```

File Renaming

```
#!/bin/bash
# Rename files by replacing spaces with underscores

for file in *\ *; do
    if [ -f "$file" ]; then
        newname=$(echo "$file" | tr ' ' '_')
        mv "$file" "$newname"
        echo "Renamed: $file -> $newname"
    fi
done
```

Directory Processing

```
#!/bin/bash
# List only subdirectories

for entry in *; do
    if [ -d "$entry" ]; then
        echo "$entry"
    fi
done

# Alternative using find
find . -maxdepth 1 -type d -not -name "."
```

File Content Transformation

```
#!/bin/bash
# Convert file to uppercase

if [ $# -ne 1 ]; then
    echo "Usage: $0 filename"
    exit 1
fi

input=$1
output="${input%.txt}_upper.txt"
```

```
tr '[:lower:]' '[:upper:]' < "$input" > "$output"
echo "Created: $output"
```

Monitoring File Changes

```
#!/bin/bash
# Monitor file for changes

if [ $# -ne 1 ]; then
    echo "Usage: $0 filename"
    exit 1
fi

file=$1

if [ ! -f "$file" ]; then
    echo "File not found: $file"
    exit 1
fi

old_md5=$(md5sum "$file" | cut -d' ' -f1)

while true; do
    sleep 5
    new_md5=$(md5sum "$file" | cut -d' ' -f1)

    if [ "$old_md5" != "$new_md5" ]; then
        echo "File changed at $(date)"
        old_md5=$new_md5
    fi
done
```

Summary

This comprehensive guide covers all essential aspects of Bash scripting:

Variables - User-defined, environment, and special variables with proper quoting and scoping

String Operations - Length, substring extraction, concatenation, replacement, and case conversion

Arrays - Indexed and associative arrays with full manipulation capabilities

Arithmetic - Integer operations with `$(( ))` and floating-point with `bc`

Functions - Reusable code blocks with local/global scope and multiple return methods

Command Execution - Running programs, command substitution, piping, and parallel execution

Scripts - Proper structure with shebang, arguments, and exit codes

Interactive Scripts - User input, menus, confirmations, and forms

Running Programs - External command execution, output capture, and environment control

Conditionals - Numeric, string, and file tests with `if`, `elif`, `else`, and `case`

Loops - `for`, `while`, `until` loops with control flow (`break`, `continue`)

File Processing - Reading, writing, analyzing, transforming, and monitoring files

By mastering these concepts and practicing with the provided examples, you can write robust, maintainable bash scripts for system administration, automation, data processing, and complex workflows.[89][90]

[89] bash_1.pdf
[90] bash.pdf