Comprehensive Guide to Conditionals (if-else) and Loops in Unix/Linux Shell Scripting (Bash)

This is a **complete, detailed reference** covering everything you need to know about conditional statements and loops in Bash (the default shell on most modern Linux distributions and macOS). I'll cover syntax, all common operators, pitfalls, best practices, version compatibility, advanced patterns, and plenty of real-world examples.

Everything here works in **Bash 3+** unless explicitly noted otherwise. I'll highlight version-specific features (e.g., variable brace expansion only in Bash 4+).

---

1. Conditional Statements

A. Basic `if` Syntax

```
Bash

if condition; then
    commands_when_true
elif another_condition; then          # optional, can have many
    commands_when_this_is_true
else                                  # optional
    commands_when_all_false
fi                                        # ← always required to close
```

Key rules:

- `;` after condition is optional if you use a newline.
- Spaces around `[` and `]` are **mandatory** in `[ ]` test.
- Use `[[ ]]` (Bash/Ksh/Zsh) for safer, more powerful tests (avoids word splitting, supports regex, etc.).
- Use `(( ))` for arithmetic only (no need for `$` on variables inside).

B. Test Commands and Operators

There are three main ways to test conditions:

1. `test` **or** `[ ]` (POSIX portable, works in sh/dash too)
2. `[[ ]]` (Bash/Zsh/Ksh – recommended for Bash scripts)
3. `(())` (Arithmetic evaluation only – integers/math)

File Test Operators (work in both `[ ]` and `[[ ]]`)

Operator	Meaning	Example
`-f file`	File exists and is regular	`[ -f "/etc/passwd" ]`
`-d dir`	Directory exists	`[ -d "/tmp" ]`
`-e file`	Exists (any type)	`[ -e "missing.txt" ] && echo "exists"`
`-r file`	Readable	`[ -r "$file" ]`
`-w file`	Writable	`[ -w "$file" ]`
`-x file`	Executable	`[ -x "/bin/ls" ]`
`-s file`	Exists and size > 0	`[ -s "log.txt" ]`
`-L file`	Symbolic link	`[ -L "/bin/sh" ]`
`-nt`	Newer than (modification time)	`[ "a.txt" -nt "b.txt" ]`
`-ot`	Older than	`[ "backup" -ot "original" ]`

String Test Operators

Operator	`[ ]` meaning	`[[ ]]` meaning (extra features)
`=` , `==`	Equal	`==` also supports pattern matching in `[[ ]]`
`!=`	Not equal	Same
`-z str`	String empty	Same
`-n str`	String non-empty	Same
`<` , `>`	Lexicographic (need escaping)	Safe in `[[ ]]` , no escaping needed
`=~`	N/A	Regex match (Bash only)

Examples:

```
Bash
str="hello"

if [ "$str" = "hello" ]; then
    echo "match"
fi

if [[ $str =~ ^h.*o$ ]]; then   # regex
    echo "matches pattern"
fi
```

Integer/Arithmetic Operators (use `(())` or `[ ]` with `-eq` etc.)

	Operator		Meaning	In (())	In [ ]
-eq	equal	==	-eq		
-ne	not equal	!=	-ne		
-lt	less than	<	-lt		
-le	less or equal	<=	-le		
-gt	greater than	>	-gt		
-ge	greater or equal	>=	-ge		

Examples:

```
Bash

a=10
b=20

if (( a < b )); then          # preferred for arithmetic
    echo "$a is smaller"
fi

if [ "$a" -lt "$b" ]; then    # portable way
    echo "$a is smaller"
fi
```

Logical Operators

Meaning	In [ ]	In [[ ]]	In (())
AND	[cond1] && [cond2] or separate if lines	cond1 && cond2	cond1 && cond2
OR	[cond1 `]		[cond2]`
NOT	! [cond]	! cond	! cond

C. Practical Examples

```bash
#!/bin/bash

read -p "Enter your age: " age

if (( age < 0 )); then
    echo "Invalid age"
elif (( age < 13 )); then
    echo "Child"
elif (( age < 18 )); then
    echo "Teenager"
elif (( age < 65 )); then
    echo "Adult"
else
    echo "Senior"
fi
```

File processing:

```bash
file="data.txt"

if [[ -r "$file" && -s "$file" ]]; then
    echo "File readable and non-empty"
elif [[ -f "$file" ]]; then
    echo "File exists but empty or not readable"
else
    echo "File not found"
fi
```

D. `case` Statement (Alternative to long if-elif chains)

```Bash
case "$var" in
    pattern1) commands ;;
    pattern2|pattern3) commands ;;   # multiple patterns
    "literal string") commands ;;
    *) default_commands ;;          # catch-all
esac
```

Example:

```Bash
read -p "Yes or No? " ans

case "${ans,,}" in         # convert to lowercase
    yes|y) echo "Affirmative" ;;
    no|n)  echo "Negative" ;;
    *)     echo "Invalid input" ;;
esac
```

---

2. Loops

A. `for` Loop

Four main styles:

1. **Classic list style** (most portable)

```Bash
for var in item1 item2 "item with space" *.txt
do
    echo "Processing: $var"
done
```

1. **C-style arithmetic loop** (Bash 3+ – very common)

```bash
for (( i = start; i <= end; i += step )); do
    commands
done
```

- Works in **Bash 3+** (2004). Fully portable in Bash.
- Variables don't need `$` inside `(())`.

1. **Brace expansion** `{start..end}` (Bash 3+ for literals, Bash 4+ for variables)

```bash
# Literals – works in Bash 3+
for i in {1..10}; do echo $i; done

# With step – Bash 4+
for i in {0..20..2}; do echo $i; done   # 0 2 4 ... 20

# Variables – Bash 4+ only
start=5
end=15
for i in {$start..$end}; do            # WRONG – expands literally
    echo $i
done

# Correct variable expansion – Bash 4+ only
for i in $(eval echo {$start..$end}); do echo $i; done
# OR (cleaner)
for i in {${start}..${end}}; do echo $i; done
```

1. **Over command output** (command substitution)

```bash
for file in $(ls *.log); do          # not recommended – breaks on spaces
    echo "$file"
done

# Better: use glob or while read
for file in *.log; do
    echo "$file"
done
```

1. `seq` **command** (external, works everywhere)

```bash
for i in $(seq 1 10); do echo $i; done
for i in $(seq 1 2 20); do echo $i; done   # step 2
for i in $(seq 10 -1 1); do echo $i; done # reverse
```

B. `while` Loop

Runs while condition is true.

```bash
count=1
while (( count <= 10 )); do
    echo $count
    ((count++))
done

# Or POSIX style
while [ $count -le 10 ]; do
    echo $count
    count=$((count + 1))
done
```

Reading lines from file/stdin:

```
Bash

while IFS= read -r line; do
    echo "Line: $line"
done < file.txt
```

Infinite loop:

```
Bash

while true; do
    commands
    sleep 1
done
```

C. `until` Loop (opposite of while)

```
Bash

count=1
until (( count > 10 )); do
    echo $count
    ((count++))
done
```

D. `select` Loop (for menus)

```
Bash

PS3="Choose option: "   # custom prompt

select fruit in apple banana orange quit; do
    case $fruit in
        apple)  echo "You chose apple" ;;
        banana) echo "You chose banana" ;;
        orange) echo "You chose orange" ;;
        quit)   break ;;
        *)      echo "Invalid" ;;
    esac
done
```

E. Loop Controls

- `break` → exit loop immediately
- `continue` → skip to next iteration
- `break N` / `continue N` → affect N nested levels

Example:

```
Bash

for i in {1..10}; do
    if (( i % 2 == 0 )); then
        continue   # skip evens
    fi
    if (( i > 7 )); then
        break      # stop at 8
    fi
    echo $i
done
```

---

3. Combining Conditionals and Loops (Real-world Patterns)

Processing files with conditions:

```
Bash

for file in /var/log/*.log; do
    if [[ -f "$file" && -r "$file" ]]; then
        size=$(stat -c%s "$file")
        if (( size > 10*1024*1024 )); then   # >10MB
            echo "Compressing $file"
            gzip "$file"
        fi
    fi
done
```

Countdown with while + if:

```bash
seconds=10
while (( seconds > 0 )); do
    echo "T-minus $seconds"
    ((seconds--))
    sleep 1
done
echo "Blast off!"
```

---

4. Best Practices & Common Pitfalls

- Always quote variables: `"$var"` – prevents word splitting and globbing.
- Use `[[ ]]` over `[ ]` in Bash scripts.
- Prefer `(( ))` for arithmetic.
- Avoid parsing `ls` output – use globs.
- For portable scripts (sh), avoid Bashisms like `[[ ]]`, `(())`, C-style for.
- Enable strict mode:

```bash
#!/usr/bin/env bash
set -euo pipefail   # exit on error, undefined var, pipe failure
IFS=$'\n\t'         # safe word splitting
```

- Debug with `set -x` (prints commands) or `bash -x script.sh`.

This covers **everything essential** (and more) about if-else and loops in Bash. You now have all syntax variations, operators, version notes, and practical patterns.

If you have a specific script you're working on or want to see how to do something particular (e.g., nested loops, reading CSV, parallel processing), just paste your code or describe the task — I'll help refine it! 😊