11.1 Bash-Scripting Grundlagen

Bash-Scripting Grundlagen

Bash-Skripte sind Textdateien, in denen mehrere Shell-Befehle gespeichert werden.

Statt Befehle immer wieder einzeln in die Shell einzugeben, kann man sie in einem Skript zusammenfassen und wiederholt ausführen.

Für Fachinformatiker Systemintegration ist Bash-Scripting wichtig, weil viele administrative Aufgaben automatisiert werden können.

Beispiele:

- Backups starten
- Logdateien prüfen
- Dienste überwachen
- Dateien aufräumen
- Benutzerinformationen auswerten
- Netzwerkprüfungen durchführen
- wiederkehrende Befehle automatisieren

Merksatz:

Ein Bash-Skript ist eine automatisierte Befehlsfolge.

Lernziele

Nach dieser Seite solltest du erklären können:

- was ein Bash-Skript ist
- was ein Shebang ist
- wie man ein Skript ausführbar macht
- wie Variablen genutzt werden
- wie Parameter übergeben werden
- wie if-Abfragen funktionieren
- wie Schleifen grundsätzlich funktionieren
- was Exit-Codes bedeuten
- warum Kommentare wichtig sind

```
- wie man einfache Admin-Aufgaben automatisiert
```

Grundidee

Ein Bash-Skript ist eine normale Textdatei mit Shell-Befehlen.

Beispiel:

```
echo "Backup startet"
date
df -h
```

Wenn diese Befehle in einer Datei gespeichert werden, kann man sie später als Skript ausführen.

Typische Dateiendung:

```
.sh
```

Beispiel:

```
backup.sh
```

Wichtig:

```
Die Endung .sh ist hilfreich,
aber nicht allein entscheidend.
Entscheidend sind Inhalt,
Rechte
und Ausführung.
```

Merksatz:

```
Bash-Skripte bestehen aus Shell-Befehlen.
```

Warum Skripte nutzen?

Skripte helfen bei:

```
- Wiederholung
- Automatisierung
- Standardisierung
- Dokumentation
- Fehlervermeidung
- schneller Ausführung
- reproduzierbaren Abläufen
```

Beispiel:

```
Statt jeden Tag mehrere Befehle manuell einzugeben,
kann ein Skript diese Befehle automatisch ausführen.
```

Merksatz:

```
Skripte machen wiederkehrende Aufgaben wiederholbar.
```

Ein erstes Skript

Datei erstellen:

```
nano hallo.sh
```

Inhalt:

```
#!/bin/bash

echo "Hallo Linux"
date
whoami
```

Speichern und schließen.

Ausführbar machen:

```
chmod +x hallo.sh
```

Starten:

```
./hallo.sh
```

Merksatz:

```
Skript erstellen,
ausführbar machen,
starten.
```

Shebang

Die erste Zeile eines Skripts ist häufig der Shebang.

Beispiel:

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

Bedeutung:

```
Das Skript soll mit /bin/bash ausgeführt werden.
```

Alternative:

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

Bedeutung:

```
bash wird über die Umgebung gesucht.
```

Wichtig:

```
Der Shebang ist besonders wichtig,
wenn ein Skript direkt gestartet wird.
```

Merksatz:

```
Shebang legt fest,
welcher Interpreter das Skript ausführt.
```

Skript ausführen

Es gibt mehrere Möglichkeiten, ein Skript auszuführen.

Direkt starten:

```
./script.sh
```

Dafür braucht das Skript Ausführrecht.

Ausführen mit bash:

```
bash script.sh
```

Dafür braucht das Skript nicht zwingend Ausführrecht, weil bash die Datei liest.

Unterschied:

Startart	Bedeutung
./script.sh	Skript direkt ausführen, Shebang und x-Recht wichtig
bash script.sh	Skript mit Bash ausführen
sh script.sh	Skript mit sh ausführen, kann anders funktionieren

Merksatz:

```
./script.sh nutzt Shebang und Ausführrecht.
bash script.sh nutzt Bash direkt.
```

Ausführrecht setzen

Ein Skript direkt ausführbar machen:

```
chmod +x script.sh
```

Prüfen:

```
ls -l script.sh
```

Beispiel:

```
-rwxr-xr-x script.sh
```

Das x bedeutet:

```
ausführbar
```

Wenn x fehlt, kommt oft:

```
Permission denied
```

Merksatz:

```
Direkt gestartete Skripte brauchen x-Recht.
```

Aktuelles Verzeichnis und ./ erklären

Wenn man ein Skript im aktuellen Verzeichnis starten möchte, nutzt man:

```
./script.sh
```

Bedeutung:

```
Starte script.sh aus dem aktuellen Verzeichnis.
```

Warum nicht einfach:

```
script.sh
```

Weil das aktuelle Verzeichnis normalerweise nicht im PATH enthalten ist.

Merksatz:

```
./ bedeutet:
aus dem aktuellen Verzeichnis.
```

Kommentare

Kommentare erklären, was ein Skript macht.

Kommentarzeilen beginnen mit:

```
#
```

Beispiel:

```
# Dieses Skript zeigt Systeminformationen an

echo "System:"
hostname
```

Kommentare werden nicht ausgeführt.

Vorteile:

```
verständlicher

wartbarer

besser für Übergabe

besser für Fehlersuche
```

Merksatz:

```
Kommentare erklären den Zweck,
nicht jeden einzelnen offensichtlichen Befehl.
```

---

echo

echo gibt Text aus.

Beispiele:

```
echo "Hallo"

echo "Backup startet"

echo "Aktueller Benutzer: $USER"
```

Typische Nutzung:

```
Statusmeldungen

Debug-Ausgaben

einfache Benutzerinformationen
```

Merksatz:

```
echo schreibt Text in die Ausgabe.
```

Variablen

Variablen speichern Werte.

Beispiel:

```
NAME="Felix"

echo "$NAME"
```

Wichtig:

```
Keine Leerzeichen um das Gleichheitszeichen.
```

Richtig:

```
NAME="Felix"
```

Falsch:

```
NAME = "Felix"
```

Merksatz:

```
Variable setzen ohne Leerzeichen.
```

Variablen verwenden

Variablen werden mit $ angesprochen.

Beispiel:

```
SERVER="web01"

echo "Servername: $SERVER"
```

Sicherer Stil:

```
echo "Servername: ${SERVER}"
```

Warum?

```
${SERVER} grenzt den Variablennamen sauber ab.
```

Merksatz:

```
$VARIABLE liest den Wert einer Variable.
```

Anführungszeichen bei Variablen

Variablen sollten meistens in doppelte Anführungszeichen gesetzt werden.

Beispiel:

```
DATEI="Meine Datei.txt"

cat "$DATEI"
```

Ohne Anführungszeichen würde die Shell den Dateinamen wegen des Leerzeichens falsch aufteilen.

Merksatz:

```
Variablen in Skripten meistens quoten:
"$VARIABLE"
```

Einfache und doppelte Anführungszeichen

Zeichen	Wirkung
" "	Variablen werden ausgewertet
' '	Variablen werden nicht ausgewertet

Beispiel:

```
NAME="Felix"

echo "Hallo $NAME"
```

Ausgabe:

```
Hallo Felix
```

Beispiel:

```
echo 'Hallo $NAME'
```

Ausgabe:

```
Hallo $NAME
```

Merksatz:

```
Doppelte Quotes erlauben Variablen.
Einfache Quotes schützen Text.
```

Befehlsausgabe in Variable speichern

Die Ausgabe eines Befehls kann in einer Variable gespeichert werden.

Beispiel:

```
HEUTE=$(date)

echo "$HEUTE"
```

Hostname speichern:

```
HOST=$(hostname)

echo "System: $HOST"
```

Merksatz:

```
$(befehl) speichert Befehlsausgabe.
```

Parameter an Skripte übergeben

Skripte können Parameter erhalten.

Beispielaufruf:

```
./backup.sh /home/felix
```

Im Skript:

```
echo "Erster Parameter: $1"
```

Wichtige Parameter:

Variable	Bedeutung

$0	Name des Skripts
$1	erster Parameter
$2	zweiter Parameter
$3	dritter Parameter
$#	Anzahl der Parameter
"$@"	alle Parameter einzeln sauber übergeben

Merksatz:

```
$1 ist der erste übergebene Parameter.
```

Beispiel mit Parameter

Skript:

```
#!/bin/bash

echo "Skriptname: $0"
echo "Erster Parameter: $1"
echo "Anzahl Parameter: $#"
```

Start:

```
./test.sh server01
```

Ausgabe sinngemäß:

```
Skriptname: ./test.sh
Erster Parameter: server01
Anzahl Parameter: 1
```

Merksatz:

```
Parameter machen Skripte flexibel.
```

Prüfen, ob Parameter fehlt

Beispiel:

```
#!/bin/bash

if [ -z "$1" ]; then
    echo "Fehler: Bitte Dateiname angeben"
    exit 1
fi

echo "Datei: $1"
```

Bedeutung:

```
-z prüft,
ob eine Zeichenkette leer ist.
```

Merksatz:

```
Eingaben prüfen,
bevor das Skript weiterarbeitet.
```

Exit-Code

Ein Skript kann mit einem Exit-Code beendet werden.

Beispiel:

```
exit 0
```

bedeutet:

```
erfolgreich
```

Beispiel:

```
exit 1
```

bedeutet:

```
Fehler
```

Allgemein:

Exit-Code	Bedeutung
0	Erfolg
ungleich 0	Fehler oder besonderer Zustand

Merksatz:

```
Exit-Code 0 bedeutet Erfolg.
```

Letzten Exit-Code anzeigen

Nach einem Befehl:

```
echo $?
```

Beispiel:

```
ls /etc

echo $?
```

Wenn der Befehl erfolgreich war, ist der Exit-Code meist:

```
0
```

Wenn der Befehl fehlschlägt, ist er ungleich 0.

Merksatz:

```
$? enthält den Exit-Code des letzten Befehls.
```

---

if-Abfrage

Mit if kann ein Skript Entscheidungen treffen.

Grundstruktur:

```
if [ Bedingung ]; then
    Befehl
fi
```

Beispiel:

```
if [ -f "/etc/hosts" ]; then
    echo "Datei existiert"
fi
```

Merksatz:

```
if führt Befehle abhängig von Bedingungen aus.
```

---

if, else und elif

Beispiel:

```
if [ "$1" = "start" ]; then
    echo "Starte Dienst"
elif [ "$1" = "stop" ]; then
    echo "Stoppe Dienst"
else
    echo "Unbekannter Parameter"
fi
```

Bedeutung:

Schlüsselwort	Bedeutung
if	wenn
elif	sonst wenn
else	sonst
fi	Ende der if-Abfrage

Merksatz:

```
fi beendet die if-Abfrage.
```

Wichtige Dateitests

Test	Bedeutung
-f DATEI	normale Datei existiert
-d PFAD	Verzeichnis existiert
-e PFAD	Datei oder Verzeichnis existiert
-r DATEI	lesbar
-w DATEI	beschreibbar
-x DATEI	ausführbar
-s DATEI	Datei existiert und ist nicht leer

Beispiele:

```
if [ -d "/var/log" ]; then
    echo "Logverzeichnis vorhanden"
fi

if [ -x "./script.sh" ]; then
    echo "Skript ist ausführbar"
fi
```

Merksatz:

```
Dateitests prüfen Pfade und Rechte.
```

Zeichenketten vergleichen

Test	Bedeutung
"$A" = "$B"	gleich
"$A" != "$B"	ungleich
-z "$A"	leer
-n "$A"	nicht leer

Beispiel:

```
if [ "$USER" = "root" ]; then
    echo "Du bist root"
fi
```

Merksatz:

```
Strings immer in Quotes vergleichen.
```

Zahlen vergleichen

Test	Bedeutung
-eq	gleich
-ne	ungleich
-lt	kleiner als
-le	kleiner oder gleich
-gt	größer als
-ge	größer oder gleich

Beispiel:

```
ANZAHL=5

if [ "$ANZAHL" -gt 3 ]; then
    echo "Mehr als 3"
fi
```

Merksatz:

```
Zahlen in Bash mit -eq,
-lt,
-gt vergleichen.
```

---

for-Schleife

Eine for-Schleife wiederholt Befehle für mehrere Werte.

Beispiel:

```
for DATEI in *.log; do
    echo "Logdatei: $DATEI"
done
```

Bedeutung:

```
Für jede Datei,
die auf .log endet,
wird der Befehl ausgeführt.
```

Merksatz:

```
for läuft über eine Liste von Werten.
```

---

while-Schleife

Eine while-Schleife läuft, solange eine Bedingung wahr ist.

Beispiel:

```
ZAHL=1

while [ "$ZAHL" -le 5 ]; do
    echo "$ZAHL"
    ZAHL=$((ZAHL + 1))
done
```

Merksatz:

```
while läuft,
solange die Bedingung stimmt.
```

---

case-Abfrage

case eignet sich gut, wenn ein Parameter mehrere mögliche Werte haben kann.

Beispiel:

```
case "$1" in
    start)
        echo "Starte"
        ;;
    stop)
        echo "Stoppe"
        ;;
    restart)
        echo "Starte neu"
        ;;
    *)
        echo "Unbekannter Parameter"
        exit 1
        ;;
esac
```

Merksatz:

```
case ist übersichtlich für mehrere Auswahlmöglichkeiten.
```

Rechnen in Bash

Einfache Rechnungen:

```
ZAHL=5

ERGEBNIS=$((ZAHL + 3))

echo "$ERGEBNIS"
```

Weitere Beispiele:

```
A=10
B=2

echo $((A + B))
echo $((A - B))
echo $((A * B))
```

```
echo $((A / B))
```

Wichtig:

```
Bash rechnet hier ganzzahlig.
```

Merksatz:

```
$(( ... )) nutzt arithmetische Auswertung.
```

---

read

Mit read kann ein Skript Benutzereingaben lesen.

Beispiel:

```
echo "Name eingeben:"
read NAME

echo "Hallo $NAME"
```

Mit Eingabeaufforderung:

```
read -p "Name: " NAME
```

Merksatz:

```
read liest Eingaben vom Benutzer.
```

Funktionen

Funktionen fassen wiederverwendbare Befehle zusammen.

Beispiel:

```
check_disk() {
    df -h
```

```
}

check_disk
```

Vorteile:

```
Skript wird übersichtlicher

Wiederholungen werden reduziert

Aufgaben werden gegliedert
```

Merksatz:

```
Funktionen machen Skripte strukturierter.
```

---

set -e

Mit set -e kann ein Skript beendet werden, wenn ein Befehl fehlschlägt.

Beispiel:

```
#!/bin/bash
set -e

mkdir /tmp/testordner
cp datei.txt /tmp/testordner/
```

Wichtig:

```
set -e kann hilfreich sein,
muss aber bewusst genutzt werden,
weil nicht jeder ungleiche Exit-Code automatisch ein echter Fehler sein muss.
```

Merksatz:

```
set -e beendet das Skript bei Fehlern schneller.
```

---

set -u

Mit set -u erzeugt die Shell einen Fehler, wenn eine nicht gesetzte Variable verwendet wird.

Beispiel:

```
set -u
```

Vorteil:

```
Tippfehler bei Variablennamen fallen schneller auf.
```

Merksatz:

```
set -u schützt vor unbemerkten leeren Variablen.
```

---

set -o pipefail

Normalerweise zählt bei einer Pipe oft nur der Exit-Code des letzten Befehls.

Mit:

```
set -o pipefail
```

wird ein Fehler innerhalb einer Pipe besser erkannt.

Beispiel:

```
grep "Fehler" datei.txt | wc -l
```

Wenn grep wegen fehlender Datei fehlschlägt, soll das Skript dies erkennen können.

Merksatz:

```
pipefail macht Fehler in Pipes sichtbarer.
```

Häufiger sicherer Skriptanfang

Ein häufig genutzter sicherer Einstieg ist:

```
#!/bin/bash
set -euo pipefail
```

Bedeutung:

Teil	Bedeutung
set -e	bei Fehler abbrechen
set -u	nicht gesetzte Variablen als Fehler behandeln
set -o pipefail	Fehler in Pipes erkennen

Wichtig:

```
Diese Optionen sind nützlich,
aber man muss verstehen,
wie sie wirken.
```

Merksatz:

```
set -euo pipefail macht Skripte strenger.
```

Temporäre Dateien

Skripte sollten temporäre Dateien vorsichtig verwenden.

Besser als feste Namen:

```
mktemp
```

Beispiel:

```
TEMPFILE=$(mktemp)

echo "Test" > "$TEMPFILE"

cat "$TEMPFILE"

rm "$TEMPFILE"
```

Warum?

```
eindeutiger Dateiname

weniger Konflikte

sicherer als feste Namen in /tmp
```

Merksatz:

```
Temporäre Dateien besser mit mktemp erzeugen.
```

Skript mit Logausgabe

Beispiel:

```
#!/bin/bash

LOGDATEI="/tmp/mein-script.log"

echo "Start: $(date)" >> "$LOGDATEI"
df -h >> "$LOGDATEI"
echo "Ende: $(date)" >> "$LOGDATEI"
```

Merksatz:

```
Skripte sollten wichtige Aktionen protokollieren.
```

Beispiel: Speicher prüfen

Ein einfaches Skript:

```
#!/bin/bash

echo "Speicherprüfung auf $(hostname)"
df -h
```

Erweitert mit Logdatei:

```
#!/bin/bash

LOG="/tmp/speichercheck.log"

echo "Prüfung: $(date)" >> "$LOG"
df -h >> "$LOG"
```

Merksatz:

```
Kleine Skripte können Admin-Prüfungen automatisieren.
```

Beispiel: Dienststatus prüfen

Beispiel:

```
#!/bin/bash

DIENST="ssh"

if systemctl is-active --quiet "$DIENST"; then
    echo "$DIENST läuft"
else
    echo "$DIENST läuft nicht"
    exit 1
fi
```

Bedeutung:

```
Das Skript prüft,
ob der Dienst aktiv ist.
```

Merksatz:

```
Skripte können Dienstzustände automatisch prüfen.
```

Beispiel: Datei sichern

Beispiel:

```
#!/bin/bash

DATEI="/etc/hosts"
BACKUP="/tmp/hosts.backup"

if [ -f "$DATEI" ]; then
    cp "$DATEI" "$BACKUP"
    echo "Backup erstellt: $BACKUP"
else
    echo "Datei nicht gefunden: $DATEI"
    exit 1
fi
```

Merksatz:

```
Vor Änderungen können Skripte automatisch Sicherungen erstellen.
```

Cron und Skripte

Skripte können regelmäßig über cron gestartet werden.

Beispielhafte Aufgaben:

```
tägliches Backup

Logprüfung

Speicherprüfung

Aufräumen temporärer Dateien

Monitoring-Check
```

Wichtig:

```
Cron hat oft eine andere Umgebung als die interaktive Shell.
```

Deshalb in Skripten möglichst:

```
absolute Pfade nutzen

PATH bewusst setzen

Logs schreiben

Fehlerausgaben umleiten
```

Merksatz:

```
Skripte in cron brauchen klare Pfade und Logging.
```

Typische Fehler in Bash-Skripten

Fehler	Ursache
command not found	Paket fehlt, PATH falsch oder Tippfehler
Permission denied	Skript nicht ausführbar oder Rechteproblem
Variable leer	nicht gesetzt oder falsch geschrieben
Syntax error	Klammer, Quote oder Schlüsselwort fehlt
Datei nicht gefunden	relativer Pfad falsch
Skript läuft per Hand, aber nicht in cron	andere Umgebung oder PATH
if funktioniert nicht	Leerzeichen bei [ ] falsch
Schleife verarbeitet Dateinamen falsch	Variablen nicht gequotet
Pipe-Fehler bleibt unbemerkt	pipefail fehlt
rm löscht zu viel	Variable leer oder Wildcard falsch

Wichtig bei [ ]

Bei if-Tests mit [ ] sind Leerzeichen wichtig.

Richtig:

```
if [ "$A" = "$B" ]; then
```

Falsch:

```
if ["$A"="$B"]; then
```

Warum?

```
[ ist ein eigener Befehl
und braucht getrennte Argumente.
```

Merksatz:

```
Bei [ ] müssen Leerzeichen stehen.
```

Variablen immer quoten

Unsicher:

```
rm $DATEI
```

Besser:

```
rm "$DATEI"
```

Warum?

```
Leerzeichen,
Sonderzeichen
oder leere Variablen können gefährlich werden.
```

Beispiel:

```
DATEI="Meine Datei.txt"
```

Ohne Quotes würde die Shell daraus mehrere Argumente machen.

Merksatz:

```
In Skripten Variablen fast immer in doppelte Quotes setzen.
```

Relative und absolute Pfade

Ein Skript kann aus verschiedenen Verzeichnissen gestartet werden.

Deshalb können relative Pfade Probleme machen.

Beispiel:

```
cp config.txt /tmp/
```

funktioniert nur, wenn config.txt im aktuellen Verzeichnis liegt.

Sicherer:

```
absolute Pfade verwenden
```

oder das Skriptverzeichnis bestimmen.

Merksatz:

```
Skripte sollten nicht blind vom aktuellen Verzeichnis abhängen.
```

Skripte und Root-Rechte

Manche Skripte brauchen administrative Rechte.

Beispiele:

```
Dienste neu starten

Pakete installieren

Dateien unter /etc ändern

Benutzer verwalten
```

Prüfen, ob Skript als root läuft:

```
if [ "$EUID" -ne 0 ]; then
    echo "Bitte als root oder mit sudo ausführen"
    exit 1
fi
```

Merksatz:

```
Administrative Skripte sollten Rechte bewusst prüfen.
```

Skript debuggen

Bash-Skript mit Debug-Ausgabe starten:

```
bash -x script.sh
```

Oder im Skript:

```
set -x
```

Bedeutung:

```
Befehle werden beim Ausführen angezeigt.
```

Ausschalten:

```
set +x
```

Merksatz:

```
bash -x hilft bei der Fehlersuche in Skripten.
```

ShellCheck

ShellCheck ist ein Werkzeug, das Bash-Skripte auf typische Fehler prüft.

Es erkennt zum Beispiel:

```
fehlende Quotes

unsichere Variablennutzung

mögliche Syntaxprobleme

ungenutzte Variablen

problematische Konstruktionen
```

Befehl, falls installiert:

```
shellcheck script.sh
```

Merksatz:

```
ShellCheck hilft,
Bash-Fehler früh zu finden.
```

Sichere Arbeitsweise beim Scripting

Vor produktiver Nutzung:

```
Skript in Testumgebung prüfen

mit harmlosen Beispieldaten testen

keine gefährlichen rm-Befehle ohne Prüfung

Variablen quoten

Fehlerbehandlung einbauen

Logs schreiben

Exit-Codes sinnvoll setzen

Backup vor Änderungen erstellen
```

Pfade bewusst wählen

Rechte prüfen

Merksatz:

Skripte automatisieren auch Fehler,
wenn man sie nicht prüft.

Typische Prüfungsfragen

Frage	Kurzantwort
Was ist ein Bash-Skript?	Textdatei mit Shell-Befehlen
Was ist ein Shebang?	legt Interpreter fest
Wie macht man ein Skript ausführbar?	chmod +x script.sh
Wie startet man ein Skript im aktuellen Verzeichnis?	./script.sh
Was ist $1?	erster Parameter
Was ist $#?	Anzahl der Parameter
Was ist "$@"?	alle Parameter sauber einzeln
Was bedeutet exit 0?	erfolgreich beenden
Was bedeutet exit 1?	mit Fehler beenden
Was macht if?	Bedingung prüfen
Was macht for?	über Werte iterieren
Was macht while?	solange Bedingung wahr ist
Was macht case?	mehrere Auswahlfälle behandeln
Warum Variablen quoten?	Schutz vor Leerzeichen und Aufteilung
Warum Logs in Skripten?	Nachvollziehbarkeit

Typische Prüfungsfallen

Falle	Richtig denken
Shebang vergessen	Skript wird eventuell mit falscher Shell ausgeführt
chmod +x vergessen	direktes Starten schlägt fehl
./ vor Skript vergessen	aktuelles Verzeichnis ist meist nicht im PATH

Falle	Richtig denken
Leerzeichen bei Variablen setzen	NAME="Wert", nicht NAME = "Wert"
Variablen nicht quoten	Probleme bei Leerzeichen oder leeren Werten
[ ] ohne Leerzeichen schreiben	Syntaxfehler
relative Pfade blind nutzen	Startverzeichnis kann anders sein
cron wie interaktive Shell behandeln	cron hat andere Umgebung
Exit-Codes ignorieren	Automatisierung erkennt Fehler nicht sauber
rm mit leerer Variable	sehr gefährlich
Skript nicht testen	Fehler werden automatisiert

IHK-sichere Kurzformulierung

Ein Bash-Skript ist eine Textdatei mit Shell-Befehlen, die automatisiert ausgeführt werden können. Die Shebang-Zeile wie #!/bin/bash legt fest, welcher Interpreter verwendet wird. Damit ein Skript direkt gestartet werden kann, benötigt es Ausführrechte, zum Beispiel mit chmod +x script.sh, und wird im aktuellen Verzeichnis mit ./script.sh gestartet. Variablen speichern Werte und sollten in Skripten meist in doppelte Anführungszeichen gesetzt werden. Parameter wie $1, $2 und $# machen Skripte flexibel. Mit if, case, for und while können Bedingungen und Wiederholungen umgesetzt werden. Exit-Codes zeigen Erfolg oder Fehler an und sind wichtig für Automatisierung, Monitoring und cronjobs. Skripte sollten getestet, dokumentiert und mit sinnvoller Fehlerbehandlung geschrieben werden.

Merksätze

```
Bash-Skript = automatisierte Befehlsfolge.

Shebang legt den Interpreter fest.

#!/bin/bash startet mit Bash.

chmod +x macht Skripte ausführbar.

./script.sh startet Skript aus aktuellem Verzeichnis.

bash script.sh startet Skript über Bash.

Kommentare erklären den Zweck.

echo gibt Text aus.
```

Variablen speichern Werte.

Keine Leerzeichen beim Setzen von Variablen.

$1 ist der erste Parameter.

$# ist die Anzahl der Parameter.

"$@" steht für alle Parameter sauber einzeln.

$(befehl) speichert Befehlsausgabe.

exit 0 bedeutet Erfolg.

exit 1 bedeutet Fehler.

$? zeigt den letzten Exit-Code.

if prüft Bedingungen.

elif bedeutet sonst wenn.

else bedeutet sonst.

fi beendet if.

for läuft über Werte.

while läuft solange eine Bedingung wahr ist.

case ist gut für mehrere Auswahlmöglichkeiten.

$((...)) rechnet ganzzahlig.

read liest Benutzereingaben.

Funktionen strukturieren Skripte.

set -e bricht bei Fehlern ab.

set -u erkennt nicht gesetzte Variablen.

pipefail erkennt Fehler in Pipes besser.

mktemp erzeugt temporäre Dateien sicherer.

Cron hat oft andere Umgebung als die normale Shell.

Variablen in Skripten meistens quoten.

Bei [ ] sind Leerzeichen wichtig.

Skripte immer testen,
bevor sie produktiv laufen.

Skripte automatisieren auch Fehler,
wenn sie falsch geschrieben sind.