ADMIN:DOCS

Bash Grundkurs #4: Funktionen in Bash

Strukturiere und modularisiere deine Bash-Skripte mit Funktionen: Deklaration, Parameterübergabe, Rückgabewerte (return vs. echo), lokale Variablen-Scopes und Funktionsbibliotheken.

Kategorie: Bash-Grundlagen **Veröffentlicht:** 25.10.2024 **Zuletzt geprüft:** 10.11.2024

Willkommen zum vierten Teil unserer technischen Serie über Linux und Bash-Programmierung!

In den ersten drei Modulen unseres Grundkurses haben wir gelernt, wie man Skripte anlegt und absichert, wie Variablen und Parameter-Expansionen funktionieren und wie man den Programmablauf über Kontrollstrukturen und Schleifen steuert.

Mit wachsender Komplexität eines Skripts entsteht jedoch schnell unübersichtlicher, redundanter Code (*„Spaghetti-Code“*). Wenn dieselbe Validierung, derselbe Logeintrag oder dieselbe Datenbankabfrage an zehn verschiedenen Stellen wiederholt wird, schleichen sich Fehler ein und die Wartung wird zum Albtraum. Die Lösung im professionellen Scripting lautet: **Modularisierung durch Funktionen** (*DRY-Prinzip: Don't Repeat Yourself*).

In dieser Lektion lernst du, wie du Bash-Funktionen sauber deklarierst, Argumente übergibst, Variablen durch `local` vor globalen Seiteneffekten kapselst, den fundamentalen Unterschied zwischen `return` (Statuscodes) und `echo` (Datenrückgabe) meisterst und eigene **wiederverwendbare Funktionsbibliotheken** erstellst.

Deklaration & Aufruf von Bash-Funktionen

Eine Funktion ist ein benannter, wiederverwendbarer Codeblock innerhalb deines Skripts. Sie verhält sich für die aufrufende Shell fast exakt wie ein eigenständiges Kommando oder ein

Quelle: https://admindocs.de/de/bash-grundlagen/bash-grundkurs-funktionen-in-bash

Shell-Built-in.



Die Syntaxformen im Vergleich

In der Bash existieren drei verschiedene Schreibweisen zur Definition einer Funktion:

```
# 1. POSIX-konforme Syntax (EMPFOHLEN):
meine_funktion() {
    echo "Ausführung der Funktion..."
}

# 2. Mit dem Schlüsselwort 'function' (Bash-spezifisch):
function meine_funktion {
    echo "Ausführung der Funktion..."
}
```

```
# 3. Kombination (Bash-spezifisch, redundant):
function meine_funktion() {
    echo "Ausführung der Funktion..."
}
```

> **Best Practice:** Verwende die Standardform `funktionsname() { ... }`. Sie ist kurz, lesbar, 100 % POSIX-kompatibel und funktioniert nahtlos in Bash, Dash und Zsh.

Der Funktionsaufruf

Im Gegensatz zu Programmiersprachen wie Python, C oder PHP werden Funktionen in der Shell **ohne runde Klammern** aufgerufen:

```
# Definition
gruss() {
    echo "Hallo aus der Funktion!"
}

# Aufruf (einfach der Name wie ein normaler Linux-Befehl):
gruss
```

Parameterübergabe: Argumente in Funktionen verarbeiten

Funktionen in der Bash besitzen ihre eigenen, isolierten **Positionsparameter**. Wenn du Argumente an eine Funktion übergibst, werden `$1`, `$2`, `$#` und `"$@"` innerhalb des Funktionskörpers temporär mit den Werten des Funktionsaufrufs belegt – unabhängig von den ursprünglichen Skriptargumenten.

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

# Funktion erwartet 2 Parameter: Vorname und Nachname
begruessung() {
    local vorname="$1"
    local nachname="$2"

    echo "Hallo $vorname $nachname! Schön, dass du da bist."
    echo "Anzahl übergebener Parameter an die Funktion: $#"
}

# Aufruf mit zwei Argumenten:
begruessung "Max" "Mustermann"
```

Wichtige Parameter-Variablen innerhalb von Funktionen:

Variable	Bedeutung im Funktionskontext
`$1`, `$2` .. `$9`	Erstes, zweites bis neuntes Argument der Funktion
`${10}`, `${11}`	Argumente ab Position 10 (geschweifte Klammern zwingend!)
`$#`	Anzahl aller an die Funktion übergebenen Argumente
`"$@"`	Alle Funktionsargumente als separat geschützte Parameter
`$FUNCNAME`	Name der aktuell ausgeführten Funktion (ideal für Logging)

Parameter mit `shift` sequenziell abarbeiten

Mit dem Befehl `shift` schiebst du die Parameterliste um eine Position nach links (`$2` wird zu `$1`, `$3` wird zu `$2` usw.):

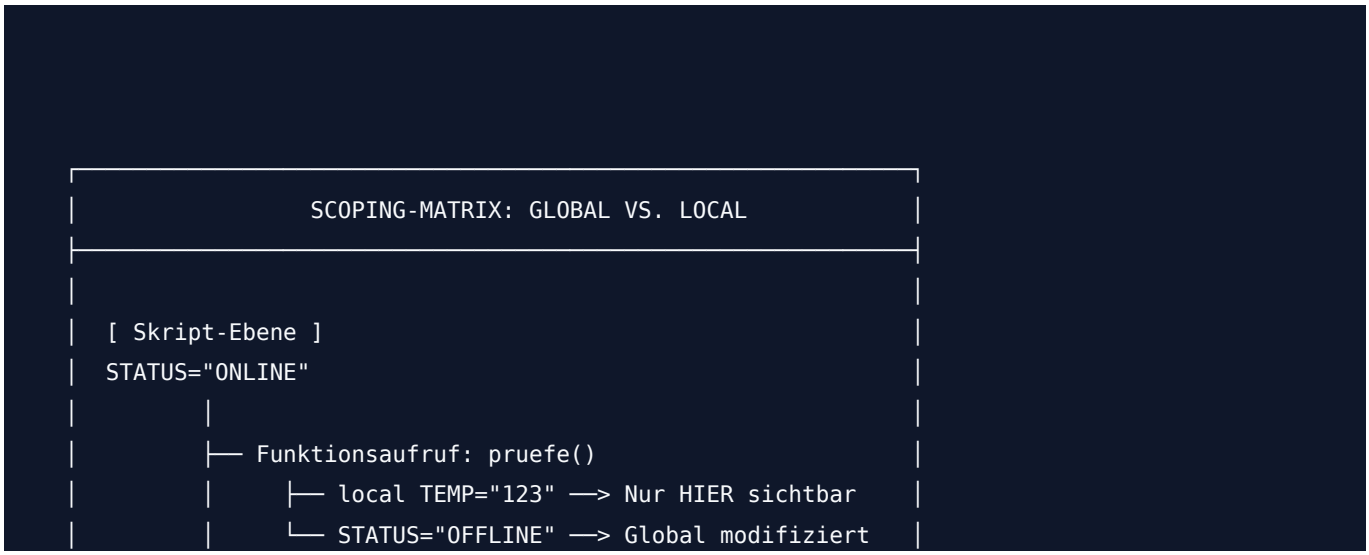
```
kopiere_dateien() {
    local zielordner="$1"
    shift # Erstes Argument (Zielordner) entfernen

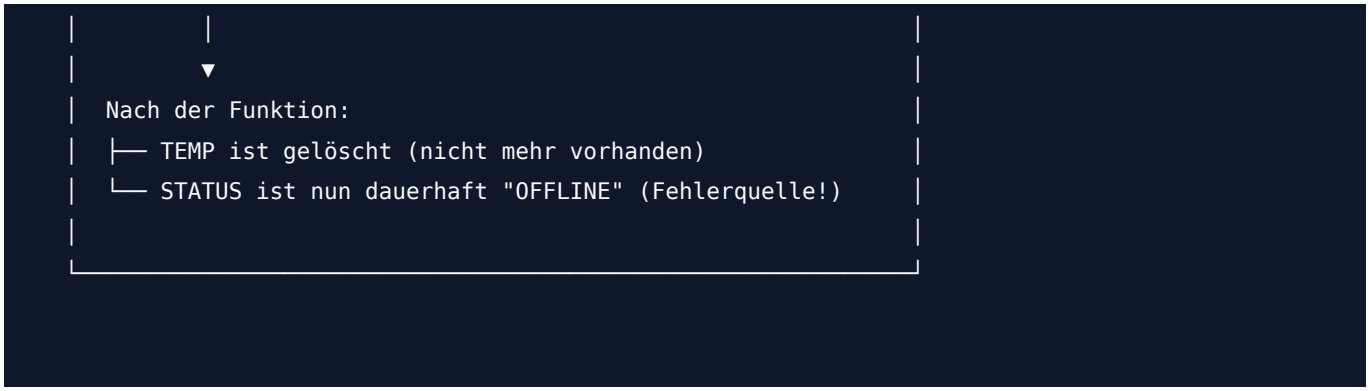
    # Der Rest in "$@" sind nun alle Quelldateien:
    for datei in "$@"; do
        echo "Kopiere $datei nach $zielordner..."
        cp "$datei" "$zielordner/"
    done
}

kopiere_dateien "/backup" "app.log" "db.sql" "config.ini"
```

Gültigkeitsbereiche: Lokale Variablen mit `local` erzwingen

In der Bash sind Variablen standardmäßig **immer global**. Wenn du innerhalb einer Funktion eine Variable ohne das Schlüsselwort `local` deklarierst, überschreibst du unbemerkt gleichnamige Variablen im Hauptprogramm!





Praxisbeispiel: Vermeidung fataler Seiteneffekte

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

# Globale Zählervariable im Hauptprogramm
i=100

zaehler_funktion() {
    # OHNE 'local' würde hier die globale Variable $i auf 0 gesetzt werden!
    local i=0
    while [[ $i -lt 3 ]]; do
        echo "Funktions-Schleife: $i"
        (( i++ ))
    done
}

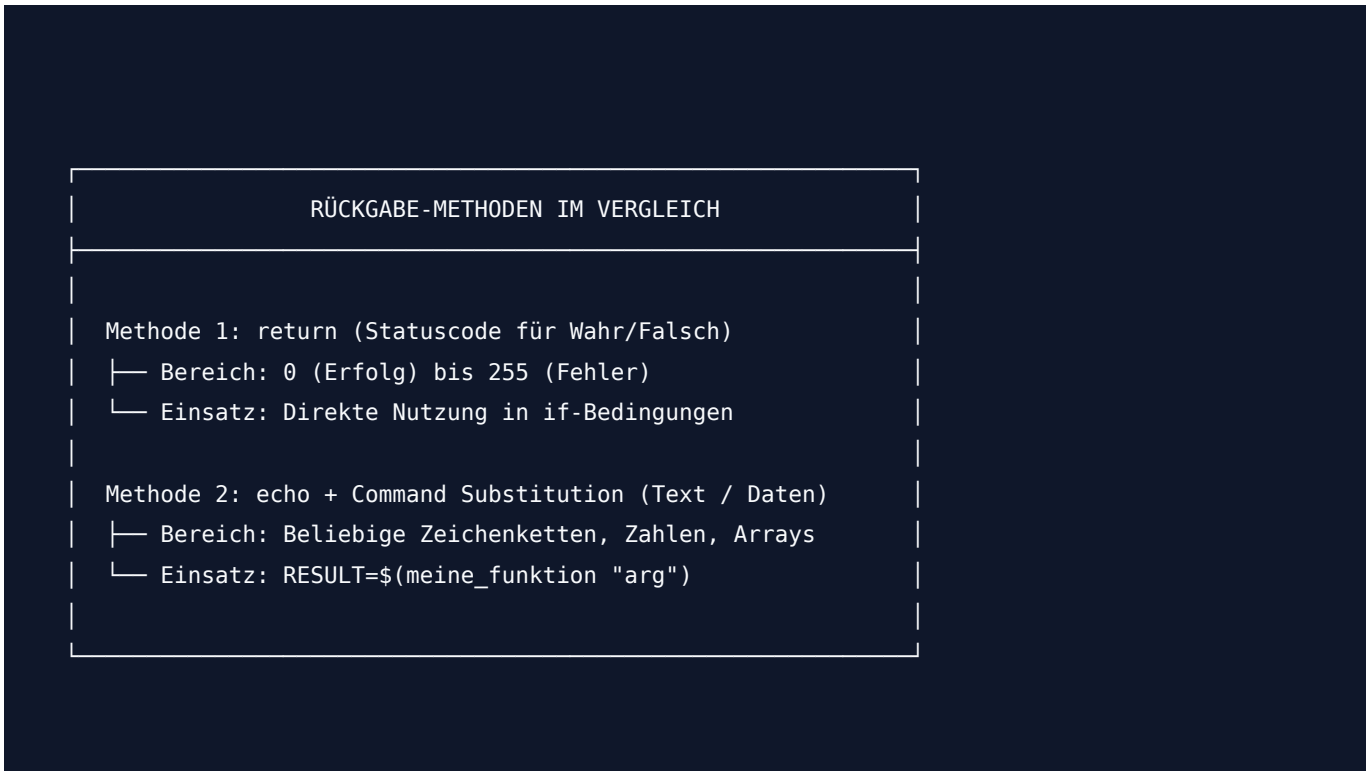
echo "Vorher: i = $i"   # Gibt 100 aus
zaehler_funktion
echo "Nachher: i = $i"  # Gibt weiterhin 100 aus (dank 'local'!)
```

> ⚠ **Sicherheitsregel:** Deklariere innerhalb von Funktionen **ausnahmslos jede Hilfsvariable mit** `local`. Das verhindert unvorhersehbare Bugs in größeren Skripten.

Rückgabewerte: `return` (Statuscodes) vs. `echo` (Datenrückgabe)

Eines der größten Missverständnisse bei Bash-Einsteigern ist die Funktionsweise von `return`:

- `return [0-255]`**:** Gibt **keine Textdaten**, sondern ausschließlich einen numerischen **Exit-Statuscode** (0 bis 255) an die Shell zurück.
- **Daten/Ergebnisse zurückgeben:** Erfolgt über die Standardausgabe (`echo`) und wird mit Command Substitution `$(funktion)` abgefangen.



1. `return` für Statusabfragen (Prädikat-Funktionen)

Funktionen, die logische Prüfungen durchführen, nutzen `return 0` für Erfolg (wahr) und `return 1` für Fehler (falsch):

```
# Prüfen, ob das Skript mit Root-Rechten läuft
```

```
is_root() {
    if [[ $EUID -eq 0 ]]; then
        return 0 # Wahr / Success
    else
        return 1 # Falsch / Error
    fi
}

# Direkte Nutzung in einer if-Verzweigung:
if is_root; then
    echo "Skript läuft als Administrator (Root)."
else
    echo "Fehler: Dieses Skript benötigt Root-Rechte!" >&2
    exit 1
fi
```

2. Daten & Werte über Command Substitution zurückgeben

Wenn eine Funktion eine Zeichenkette oder ein Rechenergebnis erzeugen soll, gibt sie den Wert per `echo` aus:

```
berechne_netto() {
    local brutto="$1"
    local steuersatz="$2"

    # Mathematische Ganzzahlberechnung
    local netto=$(( brutto * 100 / (100 + steuersatz) ))

    echo "$netto" # Textausgabe auf stdout
}

# Ergebnis in einer Variable abfangen:
NETTO_BETRAG=$(berechne_netto 119 19)
echo "Nettobetrag: ${NETTO_BETRAG} EUR"
```

Modularisierung: Funktionsbibliotheken mit `source` einbinden

In professionellen Linux-Umgebungen werden allgemeine Hilfsfunktionen (wie Logging, Konfigurations-Parsing oder Netzwerkprüfungen) in separate **Bibliotheksdateien** ausgelagert und per `source` (oder `.`) in Skripte geladen.

1. Erstelle eine Logging-Bibliothek (`lib/logging.sh`):

```
# lib/logging.sh - Wiederverwendbare Logging-Bibliothek

log_info() {
    echo -e "\e[1;32m[INFO]\e[0m $(date '+%Y-%m-%d %H:%M:%S') - $*"
}

log_warn() {
    echo -e "\e[1;33m[WARN]\e[0m $(date '+%Y-%m-%d %H:%M:%S') - $*" >&2
}

log_error() {
    echo -e "\e[1;31m[ERROR]\e[0m $(date '+%Y-%m-%d %H:%M:%S') - $*" >&2
}
```

2. Binde die Bibliothek im Hauptskript (`app.sh`) ein:

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

# Absoluten Pfad des Skriptverzeichnisses ermitteln
```

```
SKRIPT_DIR="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"

# Bibliothek einlesen
# shellcheck source=lib/logging.sh
source "$SKRIPT_DIR/lib/logging.sh"

log_info "Starte Anwendungs-Backup..."
log_warn "Speicherplatz auf /backup ist zu 85% belegt."
log_error "Konnte Datenbankverbindung nicht herstellen!"
```

Rekursive Funktionen in der Bash

Eine Funktion kann sich selbst aufrufen (*Rekursion*). Wichtig ist dabei immer eine eindeutige **Abbruchbedingung**, um eine Endlosschleife und einen Stack-Overflow zu vermeiden.

Beispiel: Fakultät berechnen ($n!$)

```
fakultaet() {
    local n="$1"

    # Abbruchbedingung: 0! = 1 und 1! = 1
    if [[ $n -le 1 ]]; then
        echo 1
        return 0
    fi

    # Rekursiver Aufruf: n * fakultaet(n - 1)
    local vorherige
    vorherige=$(fakultaet $(( n - 1 )))
    echo $(( n * vorherige ))
}

echo "Fakultät von 5 ist: $(fakultaet 5)"  # 120
```

Quelle: https://admindocs.de/de/bash-grundlagen/bash-grundkurs-funktionen-in-bash

Übungen & Praxis-Check

Praxisaufgaben zur Lektion #4:

1. **Aufgabe 1 (Host-Erreichbarkeits-Checker):**

Schreibe eine Funktion `check_host()`, die einen Hostnamen oder eine IP als Parameter `$1` entgegennimmt, einen `ping -c 1 -W 2 "$1"` ausführt und `return 0` bei Erfolg bzw. `return 1` bei Nichterreichbarkeit zurückgibt.

1. **Aufgabe 2 (Universeller Taschenrechner):**

Erstelle eine Funktion `calc()`, die drei Argumente erwartet (`Zahl1`, `Operator`, `Zahl2`), mit `case` die Operation (`+`, `-`, `*`, `/`) ausführt und das Ergebnis via `echo` zurückgibt. Fange eine Division durch Null mit einer Fehlermeldung und `return 1` ab.

1. **Aufgabe 3 (String-Trimmer):**

Schreibe eine Funktion `trim()`, die führende und nachfolgende Leerzeichen aus einer übergebenen Zeichenkette entfernt.

Musterlösung zu Aufgabe 2:

```
#!/usr/bin/env bash
set -euo pipefail
IFS=$'\n\t'

calc() {
    local a="${1:?Erste Zahl fehlt}"
    local op="${2:?Operator fehlt (+, -, *, /)}"
    local b="${3:?Zweite Zahl fehlt}"
```

```
    case "$op" in
        +)
            echo $(( a + b ))
            ;;
        -)
            echo $(( a - b ))
            ;;
        \*)
            echo $(( a * b ))
            ;;
        /)
            if [[ $b -eq 0 ]]; then
                echo "Fehler: Division durch Null nicht erlaubt!" >&2
                return 1
            fi
            echo $(( a / b ))
            ;;
        *)
            echo "Fehler: Unbekannter Operator '$op'!" >&2
            return 1
            ;;
    esac
}

# Testlauf
echo "Ergebnis: $(calc 42 + 8)"
echo "Ergebnis: $(calc 100 / 4)"
```

Befehlsreferenz (Cheatsheet)

Syntax / Befehl	Beschreibung & Funktion
`name() { ... }`	Standard-Deklaration einer Funktion in der Bash
`name "arg1" "arg2"`	Funktionsaufruf mit zwei Argumenten (ohne Klammern!)
`local VAR="wert"`	Deklariert eine Variable mit lokalem Funktions-Scope
`$1`, `$2`, `"$@"`	Zugriff auf die Positionsparameter der Funktion

Syntax / Befehl	Beschreibung & Funktion
`shift`	Schiebt Parameterliste um eine Position nach links
`return 0`	Beendet die Funktion mit Erfolgsstatus (0 bis 255)
`return 1`	Beendet die Funktion mit Fehlerstatus
`VAR=$(func)`	Fängt Textausgaben (`stdout`) der Funktion in Variable ab
`source lib.sh`	Liest externe Funktionsbibliothek in das Skript ein
`$FUNCNAME`	Enthält den Namen der aktuell ausgeführten Funktion

Weiterführende Ressourcen

Ressource	Beschreibung
GNU Bash Shell Functions	Offizielle GNU-Referenz zur Funktionsdeklaration und Scopes
Bash-Grundkurs #3: Kontrollstrukturen	Der vorherige Teil: Verzweigungen, Tests und Schleifen
Linux Befehlszeilenprozessor Leitfaden	Fundamentales Wissen zu Shells, I/O-Streams und Pipes
chmod & Dateiberechtigungen	Linux-Berechtigungskonzepte im Detail

Fazit

Funktionen sind der entscheidende Schritt von einfachen linearen Skripten zu professioneller Softwarearchitektur in der Linux-Administration. Durch die saubere Kapselung von Variablen mit `local`, die klare Trennung von Fehler-Rückgabecodes (`return`) und Daten (`echo`) sowie die Auslagerung in modulare Bibliotheken werden deine Skripte übersichtlich, robust und zukunftssicher.

> □□ **Tipp:** Erstelle dir einen zentralen Ordner `lib/` für wiederkehrende Funktionen wie Logging, Konfigurationsprüfungen oder Benutzervalidierungen. Das spart bei jedem neuen

> Skript wertvolle Zeit und vereinheitlicht das Fehlerhandling auf deinen Servern.

In der nächsten Lektion widmen wir uns der Kommunikation deiner Skripte mit der Außenwelt:

Weiter geht es mit: Bash Grundkurs #5: Ein- und Ausgabe in Bash

Zur Kursübersicht: Alle Lektionen des Bash-Grundkurses