

Klar, hier kommt ein ausführliches **Bash-Cheatsheet für Anfänger**, das dir Schritt für Schritt die Grundlagen erklärt — inklusive wie man Terminalbefehle in Bash ausführt — und gleichzeitig auf wichtige Themen vorbereitet, die du für Fortgeschrittene brauchst.

Bash-Cheatsheet für Anfänger – mit Erklärungen

1. Was ist Bash?

- **Bash** (Bourne Again SHell) ist eine Kommandozeilen-Shell, die in Linux, macOS und auch Windows (über WSL oder Git Bash) verwendet wird.
 - Mit Bash kannst du Befehle eingeben, Programme starten und Skripte schreiben.
 - Ein **Terminal** ist das Fenster, in dem du Bash-Befehle eintippst.
-

2. Terminal öffnen & Befehle ausführen

- Öffne dein Terminal (z.B. "Terminal" auf macOS/Linux, "Git Bash" oder "WSL" auf Windows).
- Du siehst eine Eingabeaufforderung (z.B. \$ oder username@pc : ~\$).
- Gib einen Befehl ein und drücke **Enter**, um ihn auszuführen.

Beispiel:

```
ls
```

Zeigt alle Dateien und Ordner im aktuellen Verzeichnis an.

3. Grundlegende Terminalbefehle

Befehl	Erklärung	Beispiel
pwd	Zeigt den aktuellen Ordner (Pfad)	pwd
ls	Listet Dateien/Ordner im aktuellen Ordner	ls -l (mit Details)
cd	Wechsel in ein anderes Verzeichnis	cd Dokumente
mkdir	Neuen Ordner erstellen	mkdir test
rm	Datei löschen	rm datei.txt
rmdir	Ordner löschen (leer)	rmdir test
cp	Datei kopieren	cp quelle.txt ziel.txt
mv	Datei verschieben/umbenennen	mv alt.txt neu.txt
cat	Inhalt einer Datei anzeigen	cat datei.txt
echo	Text ausgeben	echo Hallo Welt

4. Variablen in Bash

- Variablen speichern Werte (Texte, Zahlen).
- So setzt du eine Variable (kein Leerzeichen vor/nach =):

```
name="Max"  
alter=25
```

- So liest du die Variable aus (mit \$ davor):

```
echo "Hallo $name"  
echo "Alter: $alter"
```

5. Kommentare

- Alles nach # ist ein Kommentar und wird nicht ausgeführt.

```
# Das ist ein Kommentar  
echo "Dieser Befehl wird ausgeführt"
```

6. Einfache Bedingungen (if)

- Damit kannst du Entscheidungen treffen.

```
if [[ "$name" == "Max" ]]; then  
    echo "Hallo Max"  
else  
    echo "Du bist nicht Max"  
fi
```

- [[...]] ist die Bedingung, == bedeutet gleich.
-

7. Schleifen

- Wiederhole Befehle mehrfach.

```
for i in {1..5}; do  
    echo "Zahl: $i"  
done
```

- Ausgabe:

```
Zahl: 1  
Zahl: 2  
Zahl: 3  
Zahl: 4  
Zahl: 5
```

8. Funktionen

- Gruppieren Befehle in einer Funktion, die du später aufrufen kannst.

```
gruessen() {  
    echo "Hallo $1"  
}
```

```
gruessen "Max" # Aufruf mit Parameter "Max"
```

9. Eingaben lesen

- Frage den Nutzer nach Eingaben:

```
read -p "Wie heißt du? " name  
echo "Hallo $name"
```

10. Dateien lesen & schreiben

- Schreibe Variablen in eine Datei:

```
echo "$name" > name.txt
```

- Lese Inhalt aus einer Datei:

```
inhalt=$(cat name.txt)  
echo "Dateiinhalt: $inhalt"
```

- Oder direkt:

```
read -r inhalt < name.txt  
echo "$inhalt"
```

11. Programme und Befehle ausführen

- Du kannst beliebige Programme/Befehle im Terminal aufrufen.

Beispiel:

```
date # zeigt das aktuelle Datum an
```

- Möchtest du die Ausgabe eines Befehls in einer Variable speichern, mach das so:

```
heute=$(date +%Y-%m-%d)  
echo "Heute ist $heute"
```

12. Pipelines und Umleitungen

- Verbinde Befehle mit |, sodass die Ausgabe vom ersten Befehl als Eingabe in den zweiten geht.

```
ls -l | grep ".txt"
```

- Umleitung > schreibt Ausgabe in Datei (überschreibt):

```
echo "Hallo" > hallo.txt
```

- Anhängen >> schreibt an das Ende der Datei:

```
echo "Welt" >> hallo.txt
```

- Fehlerausgaben umleiten:

```
ls nicht_existierend.txt 2> fehler.log
```

13. Wichtige Bash-Konzepte

Parameter Expansion (Variablen-Manipulation)

```
text="Hallo Welt"
```

```
echo "${text:6:4}" # Ausgabe: Welt (Substring ab Index 6, Länge 4)
```

```
echo "${#text}" # Länge des Strings (10)
```

Array (Liste)

```
früchte=("Apfel" "Banane" "Kirsche")
```

```
echo "${früchte[1]}" # Banane
```

```
echo "${#früchte[@]}" # Anzahl (3)
```

14. Fehlerbehandlung

- Skript anhalten, wenn ein Befehl fehlschlägt:

```
set -e
```

- Befehle debuggen (anzeigen lassen, was ausgeführt wird):

```
set -x
```

15. Ein einfaches Beispiel: "Hallo-Welt"-Skript

Speichere das hier als `hallo.sh`:

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

```
# Skript sagt Hallo und fragt den Namen ab
```

```
echo "Hallo! Wie heißt du?"
```

```
read name
```

```
echo "Schön dich kennenzulernen, $name!"
```

- Ausführbar machen:

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

- Ausführen:
`./hallo.sh`
-

16. Nützliche Tipps

- Mit TAB kannst du Befehle oder Dateinamen automatisch vervollständigen.
 - Mit Pfeiltasten kannst du durch die Befehls-Historie blättern.
 - `ctrl + C` stoppt aktuell laufende Befehle.
 - Skripte beginnen meist mit der Zeile `#!/bin/bash` (Shebang) – sagt dem System, dass Bash verwendet werden soll.
-

17. Was kommt als Nächstes?

Wenn du diese Basics beherrschst, kannst du tiefer in:

- komplexe Bedingungen,
 - Schleifen mit Arrays,
 - Funktionen mit Rückgabewerten,
 - Fehlerbehandlung und Debugging,
 - Prozesssteuerung,
 - und das Schreiben größerer Skripte eintauchen.
-

Falls du möchtest, kann ich dir danach das **Fortgeschrittene Bash-Cheatsheet** geben, das ich vorhin vorbereitet habe, mit den erweiterten Themen.

Möchtest du Beispiele zu bestimmten Themen oder eine Übung?



Bash-Cheatsheet — Fortgeschritten

1. Variablen & Parameter

```
# Variable setzen  
name="Max"
```

```
# Variable lesen (ohne $ in Zuweisung, mit $ beim Lesen)  
echo "Hallo $name"
```

```
# Lesen von Kommando-Ausgabe in Variable  
date=$(date +"%Y-%m-%d")
```

```
# Default-Werte
echo "${var:-default}" # Wenn var leer oder unset, "default"
echo "${var:=default}" # Wenn var leer/unset, setze auf "default" und gib aus

# Array
arr=(eins zwei drei)
echo "${arr[1]}"      # Ausgabe: zwei
echo "${#arr[@]}"      # Anzahl Elemente
```

2. Bedingungen (if / test)

```
if [[ -f "datei.txt" ]]; then
    echo "Datei existiert"
fi

# String vergleichen
if [[ "$name" == "Max" ]]; then echo "Hallo Max"; fi

# Numerische Vergleiche
if (( 5 > 3 )); then echo "5 größer als 3"; fi

# Mehrere Bedingungen
if [[ -f "datei.txt" && -s "datei.txt" ]]; then echo "Datei existiert und ist
nicht leer"; fi

# case
case "$var" in
    start) echo "Starte";;
    stop) echo "Stoppe";;
    *) echo "Unbekannt";;
esac
```

3. Schleifen

```
# for-Schleife über Liste
for i in {1..5}; do
    echo "Zahl: $i"
done

# for über Array
for element in "${arr[@]"}; do
    echo "$element"
done

# while-Schleife
count=1
while (( count <= 5 )); do
    echo "$count"
    ((count++))
done

# until-Schleife (läuft bis Bedingung wahr wird)
until [[ -f "fertig.txt" ]]; do
    sleep 1
done
```

4. Funktionen

```
funktion_name() {  
    echo "Parameter 1: $1"  
    return 0  
}
```

```
funktion_name "Hallo"
```

```
# Funktionen können auch Werte via stdout zurückgeben (bash gibt nur Statuscodes zurück)  
ergebnis=$(funktion_name "Test")  
echo "$ergebnis"
```

5. Input/Output

```
# Ausgabe mit echo  
echo "Hallo Welt"
```

```
# Ausgabe formatieren  
printf "Name: %s, Alter: %d\n" "Max" 25
```

```
# Eingabe lesen  
read -p "Gib deinen Namen ein: " user
```

```
# Mehrere Eingaben lesen  
read -p "Zahl 1: " a  
read -p "Zahl 2: " b
```

```
# Piping  
ls -l | grep ".txt"
```

```
# Umleitung  
ls > dateien.txt           # Ausgabe in Datei speichern  
ls 2> fehler.log          # Fehlerausgabe in Datei speichern  
ls &> alles.log            # stdout + stderr in eine Datei
```

```
# Hier-Strings & Here-Documents  
grep "Max" <<< "$text"
```

```
cat << EOF > datei.txt  
Zeile 1  
Zeile 2  
EOF
```

6. Prozesse

```
# Hintergrundprozess  
sleep 30 &
```

```
# Jobs anzeigen  
jobs
```

```
# Prozess mit PID beenden
```

```
kill 1234

# Prozess mit Namen beenden
pkill firefox

# Warten auf Prozess
wait %1

# Prozessstatus abfragen
ps aux | grep bash
```

7. Dateien & Verzeichnisse

```
# Prüfen, ob Datei oder Verzeichnis existiert
[[ -f "datei.txt" ]] && echo "Datei"
[[ -d "/home" ]] && echo "Verzeichnis"

# Erstellen
mkdir -p /tmp/testdir

# Dateien kopieren/verschieben
cp quelle.txt ziel.txt
mv alt.txt neu.txt

# Dateien löschen
rm datei.txt

# Finden
find /tmp -type f -name "*.log" -mtime -7 # Logdateien der letzten 7 Tage

# Dateiinhalte anzeigen
head -n 10 datei.txt
tail -f /var/log/syslog
```

8. String-Manipulation

```
text="Hallo Welt"

# Länge
echo "${#text}" # 10

# Substring (Startindex 6, Länge 4)
echo "${text:6:4}" # Welt

# Ersetzen (erstes Vorkommen)
echo "${text/Welt/Bash}" # Hallo Bash

# Ersetzen (alle Vorkommen)
echo "${text//l/L}" # HaLLo WeLt
```

9. Parameter Expansion & Special Vars

```
# Alle Argumente zählen
echo "$#"

```

```
# Alle Argumente
echo "$@"

# Position der letzten Pipeline in $PIPESTATUS (z.B. Fehlerstatus)
ls /nichtvorhanden
echo "${PIPESTATUS[0]}" # gibt Status von ls

# Exit-Status letzter Befehl
echo "$?"

# Verzeichnis des aktuellen Skripts
script_dir="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"

# Zufallszahl (0..32767)
echo "$RANDOM"
```

10. Error Handling & Debugging

```
# Skript soll bei Fehlern abbrechen
set -e

# Debug-Modus aktivieren (Befehle anzeigen)
set -x

# Eigener Fehlercheck
if ! cp quelle.txt ziel.txt; then
    echo "Fehler beim Kopieren" >&2
    exit 1
fi

# Trap zum Aufräumen bei Skriptende oder Signal
trap "echo 'Skript wird beendet'; exit" SIGINT SIGTERM
```

11. Arrays und Schleifen

```
# Mehrdimensionales Array (simuliert)
matrix=( "1 2 3" "4 5 6" "7 8 9" )

for row in "${matrix[@]}; do
    for val in $row; do
        echo "$val"
    done
done
```

12. Spezielle Tricks

```
# Kommando nur ausführen, wenn vorheriger Befehl erfolgreich war
mkdir test && cd test

# Kommando ausführen, wenn vorheriger Befehl fehlschlug
mkdir test || echo "Ordner existiert schon"

# Mehrere Kommandos in einer Zeile
```

```
echo "Start"; ls; echo "Ende"

# Here-String (Text an Kommando geben)
grep "Max" <<< "Max Mustermann"

# Parameter als Array aus Zeichen splitten
IFS=',' read -ra teile <<< "a,b,c,d"
echo "${teile[2]}" # c
```

Wenn du willst, kann ich dir das auch als PDF oder in einer anderen Form vorbereiten. Willst du noch bestimmte Themen rein?

Um **Variablen in .json-Dateien** mit Bash zu speichern, gibt es mehrere Möglichkeiten. Die gängigste Methode ist, **Text mit der richtigen JSON-Struktur** zu erzeugen und in eine Datei zu schreiben. Hier zeige ich dir die **saubere, sichere und flexible** Art:

✅ Methode 1: Manuell JSON zusammenbauen (einfach, aber fehleranfällig)

```
# Bash-Variablen
name="Max"
alter=25

# JSON-Datei schreiben
echo "{
  \"name\": \"$name\",
  \"alter\": $alter
}" > daten.json
```

📁 Ergebnis (daten.json):

```
{
  "name": "Max",
  "alter": 25
}
```

⚠️ **Achtung bei Sonderzeichen** (z. B. Anführungszeichen im Namen) – sie müssen in JSON escaped werden, was mit Bash allein schwierig ist. → Dafür ist `jq` besser (siehe Methode 2).

✅ Methode 2: JSON sicher mit `jq` erzeugen (empfohlen)

[`jq`](#) ist ein Tool zum Bearbeiten und Erzeugen von JSON. Du kannst es wie folgt nutzen:

```
# Installieren (wenn nötig)
# sudo apt install jq          # Debian/Ubuntu
# brew install jq              # macOS

# Bash-Variablen
name="Max"
alter=25

# Mit jq JSON sicher erzeugen
```

```
jq -n \
--arg name "$name" \
--argjson alter "$alter" \
'{name: $name, alter: $alter}' > daten.json
```

Vorteil: Auch Sonderzeichen, Zeilenumbrüche oder Anführungszeichen im Namen werden automatisch korrekt maskiert.



Methode 3: Arrays oder Listen speichern

```
# Bash-Array
hobbys=("lesen" "fußball" "programmieren")

# jq erwartet JSON-Array
jq -n \
--argjson hobbys "$(\printf '%s\n' "${hobbys[@]}" | jq -R . | jq -s .)" \
'{hobbys: $hobbys}' > hobbys.json
```

Ausgabe:

```
{
  "hobbys": ["lesen", "fußball", "programmieren"]
}
```



Bonus: Mehrere Werte gemeinsam speichern

```
jq -n \
--arg name "$name" \
--argjson alter "$alter" \
--argjson hobbys "$(\printf '%s\n' "${hobbys[@]}" | jq -R . | jq -s .)" \
'{name: $name, alter: $alter, hobbys: $hobbys}' > daten.json
```



Datei laden (lesen) mit Bash + jq

```
# Name auslesen
name=$(jq -r '.name' daten.json)
echo "Name aus JSON: $name"

# Liste durchgehen
jq -r '.hobbys[]' daten.json
```

Fazit

Methode	Vorteile	Nachteile
Manuelles echo	Schnell, keine extra Tools	Unsicher bei Sonderzeichen
jq verwenden	Sicher, lesbar, robust	Externes Tool notwendig

Möchtest du ein Beispielskript, das Variablen abfragt, in JSON speichert **und wieder lädt**?

Du kannst `.sh`-Dateien (Shell-Skripte) in Python ganz einfach mit dem `subprocess`-Modul ausführen. Hier sind verschiedene Möglichkeiten, je nachdem was du brauchst:

✓ 1. Einfaches Ausführen eines Shell-Skripts

```
import subprocess

subprocess.run(["./skript.sh"])
```

Voraussetzung: Die Datei `skript.sh` ist ausführbar (`chmod +x skript.sh`).

✓ 2. Mit Bash ausführen (wenn Skript nicht ausführbar ist)

```
import subprocess

subprocess.run(["bash", "skript.sh"])
```

✓ 3. Ausgabe erfassen

Wenn du die Ausgabe des Skripts in Python weiterverwenden möchtest:

```
import subprocess

result = subprocess.run(["bash", "skript.sh"], capture_output=True, text=True)

print("Ausgabe:", result.stdout)
print("Fehler:", result.stderr)
```

✓ 4. Mit Fehlerbehandlung

```
import subprocess

try:
    subprocess.run(["bash", "skript.sh"], check=True)
    print("Skript erfolgreich ausgeführt.")
except subprocess.CalledProcessError as e:
    print("Fehler beim Ausführen des Skripts:", e)
```

📌 Tipp: Pfade sicher angeben

Falls dein Skript z. B. im `scripts/`-Ordner liegt, nutze lieber `os.path`:

```
import subprocess
import os

pfad = os.path.join("scripts", "mein_skript.sh")
subprocess.run(["bash", pfad])
```

Wenn du magst, kann ich dir auch eine Beispielstruktur zeigen, wo Python ein `.sh`-Installationsskript startet und den Fortschritt im Terminal anzeigt. Sag einfach Bescheid!