Von Befehlen zum Skript

TECH STARTER

Von Befehlen zum Skript

grep und Reguläre Ausdrücke

TECH STARTER

grep

- ➔ grep = “global regular expression print”
- ➔ Suche in Dateien nach einem gegebenen Muster
- ➔ Gibt die Zeile mit dem angegebenen Muster rot markiert aus
  - **-i** Die Suche ist case-insensitiv, ignoriert also Groß-/Kleinschreibung.
  - **-r** Die Suche ist rekursiv (sie sucht in allen Dateien innerhalb des angegebenen Verzeichnisses und seiner Unterverzeichnisse).
  - **-c** Die Suche zählt die Anzahl der Treffer.
  - **-v** Kehrt die Übereinstimmung um und gibt Zeilen aus, die nicht dem Suchbegriff entsprechen.
  - **-E** Schaltet erweiterte reguläre Ausdrücke ein (benötigt von einigen fortgeschritteneren Metazeichen wie |, + und ?).

```
$ grep bash /etc/passwd
root:x:0:0:root:/root:/bin/bash
user:x:1001:1001:User,,,,:/home/user:/bin/bash
```

Reguläre Ausdrücke

```
$ echo "aaabbb1" > text.txt
$ echo "abab2" >> text.txt
$ echo "noone2" >> text.txt
$ echo "class1" >> text.txt
$ echo "alien2" >> text.txt
$ cat text.txt
aaabbb1
abab2
noone2
class1
alien2
```

```
$ grep "ab" text.txt
aaabbb1
abab2
$ grep "[ab]" text.txt
aaabbb1
abab2
class1
alien2
```

```
$ grep "^a" text.txt
aaabbb1
abab2
alien2
$ grep "2$" text.txt
abab2
noone2
alien2
```

- ➔ Reguläre Ausdrücke sind leistungsstarke Werkzeuge zur Beschreibung von Mustern in Textdateien und zur Extraktion von Daten aus diesen, besonders nützlich in Programmiersprachen.
- ➔ Sie erfordern präzise Formulierung, da jedes Zeichen im Muster zählt und sowohl ASCII- als auch Unicode-Zeichen verwendet werden können.
- ➔ Metazeichen:
  - ◆ **[^abcABC]** Übereinstimmung eines beliebigen Zeichens außer denen innerhalb der Klammern.
  - ◆ **[a-z]** Übereinstimmung eines beliebigen Zeichens im angegebenen Bereich.
  - ◆ **[^a-z]** Übereinstimmung eines beliebigen Zeichens außer denen im angegebenen Bereich. sun|moon Übereinstimmung mit einer der angegebenen Zeichenketten.
  - ◆ **^** Zeilenbeginn
  - ◆ **$** Zeilenende
- ➔ Zur Wiederholung eines Musters:
  - ◆ ***** Null oder mehr Vorkommen des vorhergehenden Musters.
  - ◆ **+** Ein oder mehr Vorkommen des vorhergehenden Musters.
  - ◆ **?** Null oder ein Vorkommen des vorhergehenden Musters.

```
$ grep -E "ab.+" text.txt
aaabbb1
abab2
```

TECH STARTER

Von Befehlen zum Skript

Vi und Nano

TECH
STARTER

Häufig genutzte Texteditoren

```
#!/bin/bash

# This is our first comment. It is also good practice to document all scripts.

echo "Hello World!"
```

- ➔ Linux-Anwendung arbeiten in einer Umgebung, in der selten grafische Texteditoren zur Verfügung stehen
- ➔ Bearbeitung von Textdateien über die Befehlszeile
  - vi
  - nano

Vi

```
#!/bin/bash

# This is our first comment. It is also good practice to document all scripts.

echo "Hello World!"
```

- ➔ Standardmäßig auf fast jedem Linux-System
- ➔ Klon → vi IMproved (vim)
- ➔ 3 verschiedene Modi
  - ◆ Navigationsmodus
  - ◆ Einfügemodus
  - ◆ Befehlsmodus

Vi

Navigationsmodus

- ➔ H - Cursor nach links
- ➔ J - Cursor nach unten
- ➔ K - Cursor nach oben
- ➔ L - Cursor nach rechts

Vi

Navigationsmodus

- ➔ W - springt zum Anfang des nächsten Wortes
- ➔ E - springt zum Ende des nächsten Wortes
- ➔ B - springt zum Anfang des vorherigen Wortes

Vi

Navigationsmodus

- ➔ 0 - Springt zum Anfang der aktuellen Zeile
- ➔ ^ - springt zum ersten nicht-leeren Zeichen der aktuellen Zeile
- ➔ $ - springt zum Ende der aktuellen Zeile

Vi

Navigationsmodus

- ➔ Ctrl + u - Scrollt eine halbe Seite nach oben
- ➔ Ctrl + d - Scrollt eine halbe Seite nach unten
- ➔ Ctrl + b - Scrollt eine Seite nach oben
- ➔ Ctrl + f - Scrollt eine Seite nach unten

Vi

Navigationsmodus

- ➔ gg - springt zum Anfang der Datei
- ➔ G - springt zum Ende der Datei
- ➔ Ng - Springt zu Zeile n (z.B. 5g zu Zeile 5)

Vi

Navigationsmodus

- ➔ / - sucht nach einem Text nach vorne. Nach dem Eingeben des Suchbegriffs kannst du mir n weiter vorwärts und N rückwärts durch die Suchergebnisse springen
- ➔ ? - sucht nach einem Text rückwärts

Vi

```
#!/bin/bash

# This is our first comment. It is also good practice to document all scripts.

echo "Hello World!"
```

- ➔ Standardmäßig auf fast jedem Linux-System
- ➔ Klon → vi IMproved (vim)
- ➔ 3 verschiedene Modi
  - ◆ Navigationsmodus
  - ◆ Einfügemodus
  - ◆ Befehlsmodus

Vi

Wechsle in den Einfügemodus

- ➔ i - Wechselt in den Einfügemodus an der aktuellen Cursorposition (vor dem aktuellen Zeichen)
- ➔ I - Wechselt in den Einfügemodus am Anfang der aktuellen Zeile
- ➔ o - fügt neue Zeile unter der aktuellen ein und wechselt in Einfügemodus (O über)
- ➔ c: Löscht den Text, abhängig von dem folgenden Bewegungsbefehl (z.B. cw löscht bis zum Ende des aktuellen Wortes) und wechselt in den Einfügemodus.
- ➔ s: Löscht das Zeichen unter dem Cursor und wechselt in den Einfügemodus.
- ➔ S: Löscht die gesamte Zeile und wechselt in den Einfügemodus (entspricht cc).

Vi

Arbeit im Einfügemodus

- ➔ Sobald man im Einfügemodus ist, kann man Text eingeben, wie man es in einem Texteditor tun würde
- ➔ Backspace: Löscht das Zeichen links vom Cursor.
- ➔ Enter: Fügt eine neue Zeile ein.
- ➔ Ctrl + h: Funktioniert wie Backspace.
- ➔ Ctrl + w: Löscht das Wort links vom Cursor.
- ➔ Ctrl + u: Löscht den Text bis zum Zeilenanfang.
- ➔ Ctrl + j: Fügt eine neue Zeile ein (ähnlich wie Enter).
- ➔ Ctrl + o: Ermöglicht die Ausführung eines Befehls im Normalmodus, kehrt aber nach der Ausführung in den Einfügemodus zurück.
- ➔ Esc: Beendet den Einfügemodus und kehrt in den Normalmodus zurück.

Vi

Zurück in den Navigationsmodus

➔ Esc: Dies ist die häufigste Methode, um den Einfügemodus zu verlassen und in den Normalmodus zurückzukehren, wo du weitere Befehle ausführen kannst.

Vi

Zurück in den Navigationsmodus

Der Einfügemodus in Vim funktioniert ähnlich wie der Textbearbeitungsmodus in anderen Texteditoren. Der Hauptunterschied besteht darin, dass du in Vim explizit in diesen Modus wechseln musst, um Text einfügen oder bearbeiten zu können.

➔ Esc: Dies ist die häufigste Methode, um den Einfügemodus zu verlassen und in den Normalmodus zurückzukehren, wo du weitere Befehle ausführen kannst.

Vi

Wechsel in den Befehlsmodus

➔ : (Doppelpunkt): Dies ist die häufigste Methode, um in den Befehlsmodus zu wechseln. Nachdem du : gedrückt hast, erscheint unten im Bildschirm eine Eingabezeile, in die du deinen Befehl eingeben kannst.

Vi

Befehlsmodus

- ➔ :w Speichert die Datei (write).
- ➔ [Dateiname]: Speichert die Datei unter dem angegebenen Dateinamen.
- ➔ :q Beendet Vim (quit).
- ➔ :q! Beendet Vim ohne zu speichern (force quit).
- ➔ :wq Speichert die Datei und beendet Vim.
- ➔ oder ZZ: Speichert die Datei (falls Änderungen vorhanden) und beendet Vim.
- ➔ [Dateiname]: Öffnet eine Datei zur Bearbeitung.
- ➔ [Dateiname]: Speichert die Datei unter einem neuen Namen und bleibt in der aktuellen Datei.

Vi

Aus den Befehlsmodus

- ➔ Nach der Eingabe eines Befehls drückst du die Enter-Taste, um den Befehl auszuführen.
- ➔ Wenn du während der Befehlsausführung den Befehl abbrechen möchtest, drücke Esc.

Nano

- ➔ neuer
- ➔ Einfacher zu bedienen als vi (?)
- ➔ Hat keine unterschiedlichen Modi
- ➔ Benutzer kann beim Start mit der Eingabe beginnen und verwendet Strg, um auf die am unteren Bildschirmrand gedruckten Werkzeuge zuzugreifen

Nano

- ➔ neuer
- ➔ Einfacher zu bedienen als vi (?)
- ➔ Hat keine unterschiedlichen Modi
- ➔ Benutzer kann beim Start mit der Eingabe beginnen und verwendet Strg, um auf die am unteren Bildschirmrand gedruckten Werkzeuge zuzugreifen

Texteditoren sind eine Frage der persönlichen Präferenz

Von Befehlen zum Skript

Grundlagen von Shell-Skripten

TECH STARTER

Grundlagen von Shell-Skripten

- ➔ Bisher haben wir Befehle von der Shell aus ausgeführt
- ➔ Können Befehle auch in eine Datei schreiben
- ➔ Wird die Datei ausgeführt, so werden diese Befehle nacheinander ausgeführt
- ➔ Skript = ausführbare Datei
- ➔ Bash ist eine Programmiersprache und eine Shell

Ausgaben anzeigen

```
$ echo 'echo "Hello World!"' > new_script
$ cat new_script
echo "Hello World!"
```

Die Datei new_script enthält nun den echo-Befehl

- ➔ echo
- ➔ Gibt ein Argument in die Standardausgabe aus
- ➔ Können das auch in ein Skript schreiben

Ein Skript ausführbar machen

```
$ ./new_script
/bin/bash: ./new_script: Permission denied
```

Hmm.. Hier fehlen uns offenbar Berechtigungen zum Ausführen

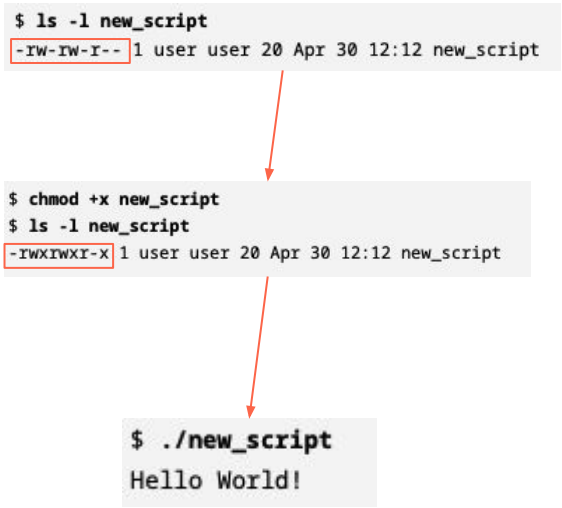
- ➔ Schauen wir uns an, was gemacht werden muss, damit die Datei das tut, was wir erwarten
- ➔ Können den Namen des Skripts nicht einfach so eingeben..
- ➔ Ausführbare Befehle waren in der Umgebungsvariablen PATH gespeichert
  - ◆ nachgucken mit which [Befehl]

```
$ which ls
/bin/ls
```

- ➔ Wir können jetzt new_script in PATH schieben, oder wir ändern die Art und Weise, wie wir das Skript aufrufen
- ➔ Dazu müssen wir den aktuellen Standort angeben, wenn wir das Skript aufrufen wollen

```
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
```

Ausführungsrechte



- ➔ Wir untersuchen unsere Datei einmal
- ➔ Wir sehen, dass die Berechtigungen für die Datei standardmäßig auf 664 (rw-rw-r–) gesetzt ist.
- ➔ Mit dem chmod-Befehl setzen wir außerdem die Ausführungsrechte (rwx-rwx-r-x)
  - ◆ Gibt allen Benutzern dann Ausführungsrechte (später gucken wir uns das nochmal an, nicht so sicher)
- ➔ Jetzt können wir das Skript ausführen

Definition des Interpreters

- ➔ new_script → Textdatei
- ➔ Typ des Interpreters!
- ➔ 1. Zeile des Skripts “Bang Line” bzw. “Shebang”
  - ◆ zeigt dem System, wie die Datei ausgeführt werden soll
- ➔ Absoluter Pfad zur ausführbaren Bash-Datei

```
$ which bash
/bin/bash
```

Shebang

```
#!/bin/bash

# This is our first comment. It is also good practice to document all scripts.

echo "Hello World!"
```

- ➔ #!absoluterPfad
- ➔ Kommentare im Skript mit #

Suffix

```
#!/bin/bash

# This is our first comment. It is also good practice to document all scripts.

echo "Hello World!"
```

- ➔ Speichere Datei als new_script.sh
- ➔ Konvention: Bash-Skripte mit .sh oder .bash kennzeichnen
  - ◆ Python-Skripte z.B. mit .py

Variablen

```
#!/bin/bash

# This is our first comment. It is also good practice to comment all scripts.

username=Carol

echo "Hello $username!"
```

```
$ ./new_script.sh
Hello Carol!
```

- ➔ Wenn wir neue Sitzung vom Terminal aus starten, setzt die Shell bereits einige Variablen, wie z.B. die Variable PATH (Umgebungsvariable)
  - ◆ Definiert Eigenschaften in unserer Shell-Umgebung
- ➔ Variable username mit Wert Carol
- ➔ Aufruf der Variablen mit $username

Variablen

```
#!/bin/bash

# This is our first comment. It is also good practice to comment all scripts.

username=Carol
x=2
y=4
z=$x+$y
echo "Hello $username!"
echo "$x + $y"
echo "$z"
```

```
$ ./new_script.sh
Hello Carol!
2 + 4
2+4
```

- ➔ Konventionen:
  - Variablennamen dürfen nur alphanumerische Zeichen oder Unterstriche enthalten und sind case-sensitiv. username und Username werden als unterschiedliche Variablen behandelt.
  - Die Variablenersetzung kann auch das Format ${username} haben, mit dem Zusatz des { }. Dies ist ebenfalls akzeptabel.
  - Variablen in Bash haben einen impliziten Typ und werden als Zeichenketten betrachtet, was bedeutet, dass die Ausführung mathematischer Funktionen in Bash komplizierter ist als in anderen Programmiersprachen wie etwa C/C++:

Variablen

```
#!/bin/bash

# This is our first comment. It is also good practice to comment all scripts.

username=Carol Smith

echo "Hello $username!"
```

```
$ ./new_script.sh
./new_script.sh: line 5: Smith: command not found
Hello !
```

➔ Was ist hier das Problem?

Variablen

```
#!/bin/bash

# This is our first comment. It is also good practice to comment all scripts.

username="Carol Smith"
echo "Hello $username!"
```

```
$ ./new_script.sh
Hello Carol Smith!
```

➔ Was ist hier das Problem?

- Leerzeichen wird als Ende der Variablenzuordnung gesehen
- Lösung: “Carol Smith”

Variablen

```
#!/bin/bash

# This is our first comment. It is also good practice to comment all scripts.

username="Carol Smith"

echo "Hello $username!"
echo 'Hello $username!'
```

```
$ ./new_script.sh
Hello Carol Smith!
Hello $username!
```

➔ Achtung: Was würde hier ‘ (einfaches Anführungszeichen) bewirken?

Argumente

```
#!/bin/bash

# This is our first comment. It is also good practice to comment all scripts.

username=$1
echo "Hello $username!"
```

- ➔ Anstatt username direkt im Skript einen Wert zuzuweisen, weisen wir ihm den Wert einer neuen Variable $1 zu
- ➔ Diese verweist auf den Wert des ersten Arguments auf

```
$ ./new_script.sh Carol
Hello Carol!
```

TECH STARTER

- ➔ Befehl ist aufgebaut in:
  - ◆ Befehl + opt. Parameter + opt. Argument
- ➔ Argumente können auch bei der Ausführung an das Skript übergeben werden und ändern das Verhalten des Skripts

Argumente

```
#!/bin/bash

# This is our first comment. It is also good practice to comment all scripts.

username=$1
echo "Hello $username!"
```

- ➔ Anstatt username direkt im Skript einen Wert zuzuweisen, weisen wir ihm den Wert einer neuen Variable $1 zu
- ➔ Diese verweist auf den Wert des ersten Arguments auf

- ➔ Befehl ist aufgebaut in:
  - ◆ Befehl + opt. Parameter + opt. Argument
- ➔ Argumente können auch bei der Ausführung an das Skript

```
#!/bin/bash

# This is our first comment. It is also good practice to comment all scripts.

username1=$1
username2=$2
echo "Hello $username1 and $username2!"
```

```
$ ./new_script.sh Carol Dave
Hello Carol and Dave!
```

Bis zu 9 Argumente!

TECH STARTER

Anzahl der Argumente zurückgeben

```
#!/bin/bash

# This is our first comment. It is also good practice to comment all scripts.

username=$1

echo "Hello $username!"
echo "Number of arguments: $#."
```

```
$ ./new_script.sh Carol Dave
Hello Carol!
Number of arguments: 2.
```

- ➔ $1 und $2 (usw.) enthalten den Wert von Positionsargumenten
- ➔ $# enthält die Anzahl der Argumente

Bedingungslogik

```
#!/bin/bash

# A simple script to greet a single user.

if [ $# -eq 1 ]
then
  username=$1
  echo "Hello $username!"
else
  echo "Please enter only one argument."
fi
echo "Number of arguments: $#."
```

- ➔ Syntax bedingter Anweisungen in Bash
  - Die Bedingung, die wir testen, ist die Anzahl der Benutzer, die in der Variablen $# enthalten ist. Wir würden gerne wissen, ob $# den Wert 1 hat.
  - Wenn die Bedingung wahr (true) ist, ist die Aktion, die wir ausführen, die Begrüßung des Benutzers.
  - Wenn die Bedingung falsch (false) ist, geben wir eine Fehlermeldung aus.

Bedingungslogik

```
#!/bin/bash

# A simple script to greet a single user.

if [ $# -eq 1 ]
then
  username=$1

  echo "Hello $username!"
else
  echo "Please enter only one argument."
fi
echo "Number of arguments: $#."
```

Hier liegt die Bedingungslogik drin

➔ Syntax bedingter Anweisungen in Bash

- Die Bedingung, die wir testen, ist die Anzahl der Benutzer, die in der Variablen $# enthalten ist. Wir würden gerne wissen, ob $# den Wert 1 hat.
- Wenn die Bedingung wahr (true) ist, ist die Aktion, die wir ausführen, die Begrüßung des Benutzers.
- Wenn die Bedingung falsch (false) ist, geben wir eine Fehlermeldung aus.

Bedingungslogik

```
#!/bin/bash

# A simple script to greet a single user.

if [ $# -eq 1 ]
then
  username=$1

  echo "Hello $username!"
else
  echo "Please enter only one argument."
fi
echo "Number of arguments: $#."
```

Hier liegt die Bedingungslogik drin

```
$ ./new_script.sh
Please enter only one argument.
Number of arguments: 0.
$ ./new_script.sh Carol
Hello Carol!
Number of arguments: 1.
```

TECH STARTER

➔ Syntax bedingter Anweisungen in Bash

- Die Bedingung, die wir testen, ist die Anzahl der Benutzer, die in der Variablen $# enthalten ist. Wir würden gerne wissen, ob $# den Wert 1 hat.
- Wenn die Bedingung wahr (true) ist, ist die Aktion, die wir ausführen, die Begrüßung des Benutzers.
- Wenn die Bedingung falsch (false) ist, geben wir eine Fehlermeldung aus.

Bedingungslogik

```
#!/bin/bash

# A simple script to greet a single user.

if [ $# -eq 1 ]
then
  username=$1

  echo "Hello $username!"
else
  echo "Please enter only one argument."
fi
echo "Number of arguments: $#."
```

Hier liegt die Bedingungslogik drin

```
$ ./new_script.sh
Please enter only one argument.
Number of arguments: 0.
$ ./new_script.sh Carol
Hello Carol!
Number of arguments: 1.
```

- ➔ if-Anweisung kann folgendes prüfen:
  - -ne Nicht gleich
  - -gt Größer als
  - -ge Größer oder gleich
  - -lt Kleiner als
  - -le Kleiner oder gleich
- ➔ Im Beispiel testen wir mit -eq, ob der Wert von $# gleich 1 ist

Aufgabe

```
$ PATH=~/scripts
$ ls
Command 'ls' is available in '/bin/ls'
The command could not be located because '/bin' is not included in the PATH environment
variable.
ls: command not found
```

➔ Der Benutzer gibt folgendes in seine Shell ein.

- Was hat der Benutzer gemacht?
- Welcher Befehl kombiniert den aktuellen Wert von PATH mit neuen Verzeichnis ~/scripts

Lösung

```
$ PATH=~/scripts
$ ls
Command 'ls' is available in '/bin/ls'
The command could not be located because '/bin' is not included in the PATH environment variable.
ls: command not found
```

- ➔ Der Benutzer gibt folgendes in seine Shell ein.
  - ◆ Was hat der Benutzer gemacht?
    - Der Benutzer hat den Inhalt von PATH mit dem Verzeichnis ~/scripts überschrieben. Der Befehl ls kann nicht mehr gefunden werden, da er nicht in PATH enthalten ist. Beachten, dass diese Änderung nur die aktuelle Sitzung betrifft. Wenn du dich aus- und wieder einloggst, sind die Änderungen rückgängig gemacht.
  - ◆ Welcher Befehl kombiniert den aktuellen Wert von PATH mit neuen Verzeichnis ~/scripts
    - PATH=$PATH:~/scripts

Aufgabe

```
>  /!bin/bash

> fruit1 = Apples
> fruit2 = Oranges

  if [ $1 -lt $# ]
  then
    echo "This is like comparing $fruit1 and $fruit2!"
> elif [$1 -gt $2 ]
  then
>   echo '$fruit1 win!'
  else
>   echo "Fruit2 win!"
> done
```

- ➔ Betrachte das folgende Skript. Beachte, dass es elif verwendet, um nach einer zweiten Bedingung zu suchen.
- ➔ Alle Zeilen mit > enthalten Fehler. Korrigiere diese

Lösung

```
#!/bin/bash

fruit1=Apples
fruit2=Oranges

if [ $1 -lt $# ]
then
  echo "This is like comparing $fruit1 and $fruit2!"
elif [ $1 -gt $2 ]
then
  echo "$fruit1 win!"
else
  echo "$fruit2 win!"
fi
```

- ➔ Betrachte das folgende Skript. Beachte, dass es elif verwendet, um nach einer zweiten Bedingung zu suchen.
- ➔ Alle Zeilen mit > enthalten Fehler. Korrigiere diese

Aufgabe

```
#!/bin/bash

fruit1=Apples
fruit2=Oranges

uif [ $1 -lt $# ]
then
  echo "This is like comparing $fruit1 and $fruit2!"
elif [ $1 -gt $2 ]
then
  echo "$fruit1 win!"
else
  echo "$fruit2 win!"
fi
```

➔ Was wird in folgenden Situationen ausgegeben?

- ./guided1.sh 3 0
- ./guided1.sh 2 4
- ./guided1.sh 0 1

Lösung

```
#!/bin/bash

fruit1=Apples
fruit2=Oranges

uif [ $1 -lt $# ]
then
  echo "This is like comparing $fruit1 and $fruit2!"
elif [ $1 -gt $2 ]
then
  echo "$fruit1 win!"
else
  echo "$fruit2 win!"
fi
```

➔ Was wird in folgenden Situationen ausgegeben?

- ./guided1.sh 3 0
  - Apples win!
- ./guided1.sh 2 4
  - Oranges win!
- ./guided1.sh 0 1
  - This is like comparing Apples and Oranges!

Aufgabe

```
if [ $1 == $number ]
then
  echo "True!"
fi
```

➔ Schreibe ein einfaches Skript, das überprüft, ob genau zwei Argumente übergeben werden. Wenn ja, drucke die Argumente in umgekehrter Reihenfolge. Betrachte dazu dieses Beispiel Der Code kann anders aussehen, sollte aber zur gleichen Ausgabe führen.

Lösung

```
#!/bin/bash

if [ $# -ne 2 ]
then
  echo "Error"
else
  echo "$2 $1"
fi
```

➔ Schreibe ein einfaches Skript, das überprüft, ob genau zwei Argumente übergeben werden. Wenn ja, drucke die Argumente in umgekehrter Reihenfolge. Betrachte dazu dieses Beispiel Der Code kann anders aussehen, sollte aber zur gleichen Ausgabe führen.

Aufgabe

```
#!/bin/bash

if [ $# -ne 2 ]
then
  echo "Error"
else
  echo "$2 $1"
fi
```

➔ Dieser Code ist korrekt, aber es handelt sich nicht um einen Zahlenvergleich. Recherchiere, um herauszufinden, worin sich dieser Code von der Verwendung von -eq unterscheidet.

Lösung

`abc == abc`	*true*
`abc == ABC`	*false*
`1 == 1`	*true*
`1+1 == 2`	*false*

➔ Dieser Code ist korrekt, aber es handelt sich nicht um einen Zahlenvergleich. Recherchiere, um herauszufinden, worin sich dieser Code von der Verwendung von -eq unterscheidet.

- Mit == werden Strings verglichen. Das heißt, wenn die Zeichen beider Variablen exakt übereinstimmen, dann ist die Bedingung wahr.
- String-Vergleiche führen zu unerwartetem Verhalten, wenn Sie auf Zahlen testen.

Aufgabe

➔ Es gibt eine Umgebungsvariable, die das aktuelle Verzeichnis ausgibt. Verwende env, um den Namen dieser Variablen zu ermitteln

Lösung

➔ Es gibt eine Umgebungsvariable, die das aktuelle Verzeichnis ausgibt. Verwende env, um den Namen dieser Variablen zu ermitteln

- PWD

Aufgabe

➔ Schreibe unter Verwendung dessen, was du in den vorherigen beiden Fragen gelernt hast, ein kurzes Skript, das ein Argument akzeptiert. Wenn ein Argument übergeben wird, überprüfe, ob dieses Argument mit dem Namen des aktuellen Verzeichnisses übereinstimmt. Wenn ja drucke ja, anderfalls drucke nein.

Aufgabe

```
#!/bin/bash

if [ "$1" == "$PWD" ]
then
  echo "yes"
else
  echo "no"
fi
```

➔ Schreibe unter Verwendung dessen, was du in den vorherigen beiden Fragen gelernt hast, ein kurzes Skript, das ein Argument akzeptiert. Wenn ein Argument übergeben wird, überprüfe, ob dieses Argument mit dem Namen des aktuellen Verzeichnisses übereinstimmt. Wenn ja drucke ja, anderfalls drucke nein.

Zusammenfassung Zwischenstand

```
#!/bin/bash

# A simple script to greet a single user.

if [ $# -eq 1 ]
then
  username=$1

  echo "Hello $username!"
else
  echo "Please enter only one argument."
fi
echo "Number of arguments: $#."
```

- ➔ Alle Skripte sollten mit einem Shebang beginnen, der den Pfad zum Interpreter definiert.
- ➔ Alle Skripte sollten Kommentare enthalten, um ihre Verwendung zu beschreiben.
- ➔ Dieses spezielle Skript arbeitet mit einem Argument, das beim Aufruf an das Skript übergeben wird.
- ➔ Dieses Skript enthält eine if-Anweisung, die die Bedingungen einer eingebauten Variablen $# testet. Diese Variable wird auf die Anzahl der Argumente gesetzt.
- ➔ Wenn die Anzahl der an das Skript übergebenen Argumente gleich 1 ist, dann wird der Wert des ersten Arguments an eine neue Variable namens username übergeben und das Skript gibt einen Gruß aus. Andernfalls erscheint eine Fehlermeldung.
- ➔ Schließlich gibt das Skript die Anzahl der Argumente aus. Dies ist für die Fehlersuche nützlich.

Exit Codes

```
$ cat -n new_script.sh

     1  #!/bin/bash
     2
     3  # A simple script to greet a single user.
     4
     5  if [ $# -eq 1 ]
     6  then
     7    username=$1
     8
     9    echo "Hello $username!"
    10  else
    11    echo "Please enter only one argument."
    12  fi
    13  echo "Number of arguments: $#."
```

```
$ echo $?
0
```

```
$ cat -n dummyfile.sh
cat: dummyfile.sh: No such file or directory
$ echo $?
1
```

- ➔ Skript hat 2 Zustände
  - ◆ Es druckt “Hello <user>”
  - ◆ Es gibt eine Fehlermeldung aus
- ➔ Überprüfe den Wert von $? mit echo
- ➔ Jede Ausführung eines Befehls gibt uns einen Exit Code
  - ◆ 0 steht für Erfolg
  - ◆ Andere Zahl steht für Fehler (je nach Befehl)
  - ◆ Fehlercode wird in $? gespeichert

Exit Codes

```
1  #!/bin/bash
2
3  # A simple script to greet a single user.
4
5  if [ $# -eq 1 ]
6  then
7    username=$1
8
9    echo "Hello $username!"
10   exit 0
11 else
12   echo "Please enter only one argument."
13   exit 1
14 fi
15 echo "Number of arguments: $#."
```

- ➔ Sind sehr nützlich beim Schreiben von Skripten
- ➔ Setze 0, um Erfolg anzuzeigen
- ➔ Setze 1, um Misserfolg anzuzeigen

```
$ ./new_script.sh Carol
Hello Carol!
$ echo $?
0
```

Exit Codes

```
1  #!/bin/bash
2
3  # A simple script to greet a single user.
4
5  if [ $# -eq 1 ]
6  then
7    username=$1
8
9    echo "Hello $username!"
10   exit 0
11 else
12   echo "Please enter only one argument."
13   exit 1
14 fi
15 echo "Number of arguments: $#."
```

Beachte, dass der Befehl echo in Zeile 15 vollständig ignoriert wurde und das Skript mit exit sofort beendet wird, so dass diese Zeile nie gefunden wird.

beim
pten
anzuzeigen
folg

```
$ ./new_script.sh Carol
Hello Carol!
$ echo $?
0
```

TECH STARTER

Behandlung mehrerer Argumente

```
#!/bin/bash

# a friendly script to greet users

if [ $# -eq 0 ]
then
  echo "Please enter at least one user to greet."
  exit 1
else
  echo "Hello $@!"
  exit 0
fi
```

```
$ ./friendly2.sh Carol Dave Henry
Hello Carol Dave Henry!
```

- ➔ Bisher kann das Skript nur einen einzigen Benutzernamen verarbeiten. Wir wollen aber eine beliebige Anzahl von Argumenten
- ➔ Positionsargumente einfach erhöhen? Dann wüssten wir aber vorher nicht, wieviele es gibt ...
  - ◆ Es gibt noch mehr eingebaute Variablen
- ➔ Umbau unseres Skripts
  - ◆ Null Argumente werfen einen Fehler
  - ◆ Beliebige Anzahl von Argumenten soll erfolgreich sein

Behandlung mehrerer Argumente

```
#!/bin/bash

# a friendly script to greet users

if [ $# -eq 0 ]
then
  echo "Please enter at least one user to greet."
  exit 1
else
  echo "Hello $@!"
  exit 0
fi
```

- ➔ $@ und $*
  - Enthalten alle an das Skript übergebene Argumente
  - Bash wird die Argumente parsen und jedes Argument trennen, wenn es auf ein Leerzeichen stößt
- ➔ Array in Bash: Elemente mit Leerzeichen getrennt

```
FILES="/usr/sbin/accept /usr/sbin/pwck /usr/sbin/chroot"
```

```
$ ./friendly2.sh Carol Dave Henry
Hello Carol Dave Henry!
```

0	1	2
Carol	Dave	Henry

TECH STARTER

Behandlung mehrerer Argumente

Wir haben also eine Liste mit mehreren Elementen. Das ist bisher nicht sehr hilfreich, da wir noch keine Möglichkeit gesehen haben, diese Elemente individuell zu behandeln

```
1  #!/bin/bash
2
3  # a friendly script to greet users
4
5  if [ $# -eq 0 ]
6  then
7    echo "Please enter at least one user to greet."
8    exit 1
9  else
10   echo "Hello $@!"
11   exit 0
12 fi
```

$@ und $*

- Enthalten alle an das Skript übergebene Argumente
- Bash wird die Argumente parsen und jedes Argument trennen, wenn es auf ein Leerzeichen stößt

➔ Array in Bash: Elemente mit Leerzeichen getrennt

```
FILES="/usr/sbin/accept /usr/sbin/pwck /usr/sbin/chroot"
```

```
$ ./friendly2.sh Carol Dave Henry
Hello Carol Dave Henry!
```

0	1	2
Carol	Dave	Henry

TECH STARTER

For-Schleifen

```
#!/bin/bash

FILES="/usr/sbin/accept /usr/sbin/pwck /usr/sbin/chroot"

for file in $FILES
do
  ls -lh $file
done
```

```
$ ./arraytest
lrwxrwxrwx 1 root root 10 Apr 24 11:02 /usr/sbin/accept -> cupsaccept
-rwxr-xr-x 1 root root 54K Mar 22 14:32 /usr/sbin/pwck
-rwxr-xr-x 1 root root 43K Jan 14 07:17 /usr/sbin/chroot
```

- ➔ Array FILES
- ➔ Brauchen Methode, um diese Variable zu “entpacken” und nacheinander auf jeden einzelnen Wert zugreifen
- ➔ Dazu gibt es 2 Variablen, auf die wir zugreifen
  - ◆ Bereich
  - ◆ Individueller Wert

For-Schleifen

```
1  #!/bin/bash
2
3  # a friendly script to greet users
4
5  if [ $# -eq 0 ]
6  then
7    echo "Please enter at least one user to greet."
8    exit 1
9  else
10   for username in $@
11   do
12     echo "Hello $username!"
13   done
14   exit 0
15 fi
```

```
$ ./friendly2.sh Carol Dave Henry
Hello Carol!
Hello Dave!
Hello Henry!
```

- ➔ Reihe von Werten in $@
- ➔ Individueller Wert ist username-Variable (kann beliebig genannt werden)

For-Schleifen

```
1  #!/bin/bash
2
3  # a friendly script to greet users
4
5  if [ $# -eq 0 ]
6  then
7    echo "Please enter at least one user to greet."
8    exit 1
9  else
10   echo -n "Hello $1"
11   shift
12   for username in $@
13   do
14     echo -n ", and $username"
15   done
16   echo "!"
17   exit 0
18 fi
```

```
$ ./friendly2.sh Carol
Hello Carol!
$ ./friendly2.sh Carol Dave Henry
Hello Carol, and Dave, and Henry!
```

- ➔ Reihe von Werten in $@
- ➔ Individueller Wert ist username-Variable (kann beliebig genannt werden)
- ➔ Können das Ganze auch in einer Zeile ausgeben lassen
  - ◆ Die Verwendung von -n mit echo unterdrückt den Zeilenumbruch nach der Ausgabe, d.h. alle Ausgaben werden auf dieselbe Zeile gedruckt, und der Zeilenumbruch folgt erst nach dem ! in Zeile 16.
  - ◆ Der Befehl shift entfernt das erste Element unseres Arrays.

Aufgabe

```
#!/bin/bash

if [ $# -lt 1 ]
then
  echo "This script requires at least 1 argument."
  exit 1
fi

echo $1 | grep "^[A-Z]*$" > /dev/null
if [ $? -ne 0 ]
then
  echo "no cake for you!"
  exit 2
fi

echo "here's your cake!"
exit 0
```

➔ Schaue dir das folgende Skript an. Wie lautet die Ausgabe der folgenden Befehle?

- ./script1.sh
- echo $?
- ./script1.sh cake
- echo $?
- ./script1.sh CAKE
- echo $?

Lösung

```
#!/bin/bash

if [ $# -lt 1 ]
then
  echo "This script requires at least 1 argument."
  exit 1
fi

echo $1 | grep "^[A-Z]*$" > /dev/null
if [ $? -ne 0 ]
then
  echo "no cake for you!"
  exit 2
fi

echo "here's your cake!"
exit 0
```

TECH STARTER

➔ Schaue dir das folgende Skript an. Wie lautet die Ausgabe der folgenden Befehle?

- ./script1.sh
  - This script requires at least 1 argument.
- echo $?
  - 1
- ./script1.sh cake
  - No cake for you!
- echo $?
  - 2
- ./script1.sh CAKE
  - Here’s your cake!
- echo $?
  - 0

Aufgabe

```
for filename in $1/*.txt
do
   cp $filename $filename.bak
done
```

➔ Was macht das folgende Skript?

Lösung

```
for filename in $1/*.txt
do
   cp $filename $filename.bak
done
```

➔ Was macht das folgende Skript?

- Dieses Skript erstellt Sicherungskopien aller Dateien, die mit .txt enden, in einem Unterverzeichnis, das im ersten Argument definiert ist.

Aufgabe

➔ Erstelle ein Skript, das beliebig viele Argumente vom Benutzer entgegennimmt und drucke nur solche Argumente, die Zahlen größer als 10 sind.

Lösung

```
#!/bin/bash

for i in $@
do
  echo $i | grep "^[0-9]*$" > /dev/null
  if [ $? -eq 0 ]
  then
    if [ $i -gt 10 ]
    then
      echo -n "$i "
    fi
  fi
done
echo ""
```

➔ Erstelle ein Skript, das beliebig viele Argumente vom Benutzer entgegennimmt und drucke nur solche Argumente, die Zahlen größer als 10 sind.