3.3 Lektion 1

Zertifikat:	Linux Essentials
Version:	1.6
Thema:	3 Die Macht der Befehlszeile
Lernziel:	3.3 Von Befehlen zum Skript
Lektion:	1 von 2

Einführung

Wir haben bisher gelernt, Befehle von der Shell aus auszuführen, aber wir können auch **Befehle in eine Datei schreiben und diese dann ausführbar machen**. Wenn die Datei ausgeführt wird, werden diese Befehle nacheinander ausgeführt. Diese ausführbaren Dateien **heißen *Skripte*** und sind ein unerlässliches Werkzeug für jeden Linux-Systemadministrator. Im Grunde können wir Bash sowohl als Programmier-sprache wie auch als Shell betrachten.

Ausgabe anzeigen

Beginnen wir mit einem Befehl, den Sie vielleicht schon in früheren Lektionen gesehen haben: `echo` gibt ein Argument in die Standardausgabe aus.

```
$ echo "Hello World!"
Hello World!
```

Nun nutzen wir die Dateiumleitung, um diesen Befehl in eine neue Datei namens `new_script` zu schreiben.

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

Die Datei `new_script` enthält nun den gleichen Befehl wie zuvor.

Ein Skript ausführbar machen

Gehen wir die notwendigen Schritte durch, damit diese Datei das tut, was wir erwarten. Der erste Gedanke eines Benutzers könnte sein, einfach den Namen des Skripts einzugeben, so wie er den Namen anderer Befehle eingibt:

```
$ new_script
/bin/bash: new_script: command not found
```

Wir wissen, dass `new_script` an unserem aktuellen Standort existiert, aber beachten Sie, dass die Fehlermeldung uns nicht sagt, dass die *Datei* nicht existiert — sie sagt uns, der ***Befehl* existiert nicht.**

Befehle und `PATH`

Wenn wir beispielsweise den Befehl `ls` in die Shell eingeben, führen wir eine Datei namens `ls` aus, die in unserem Dateisystem existiert. Sie können dies mit `which` zeigen:

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

Es würde schnell lästig werden, jedes Mal den absoluten Pfad von `ls` einzugeben, wenn wir den Inhalt eines Verzeichnisses betrachten wollen; darum hat Bash eine ***Umgebungsvariable* (*environment variable*), die alle Verzeichnisse enthält**, **in denen wir die ausführbaren Befehle** finden. Schauen Sie sich den Inhalt dieser Variable mit `echo` an:

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

An jedem dieser Orte, durch **Doppelpunkte (`:`) getrennt**, erwartet die Shell Befehle zu finden. Sie werden dort `/bin` sehen, aber es dürfte sicher sein, dass unser aktueller Standort nicht aufgeführt ist. Die Shell wird in jedem dieser Verzeichnisse nach `new_script` suchen, aber sie wird es nicht finden und daher den oben genannten Fehler anzeigen.

Es gibt **drei Lösungen für dieses Problem**: Wir können `new_script` in **eines der `PATH`-Verzeichnisse verschieben,** wir können unser **aktuelles Verzeichnis zu `PATH` hinzufügen**, oder wir ändern die **Art und Weise, wie wir das Skript aufrufen**. Die letzte Lösung ist die einfachste, denn sie verlangt lediglich, dass wir den *aktuellen Standort* angeben, wenn wir das Skript mit einem Punktstrich (`./`) aufrufen.

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

Die Fehlermeldung hat sich geändert, was darauf hindeutet, dass wir einige Fortschritte gemacht haben.

Ausführungsrechte

Die erste Untersuchung, die ein Benutzer in diesem Fall durchführen sollte, ist die Verwendung **von `ls -l`**, um sich die Datei anzusehen:

```
$ ls -l new_script
-rw-rw-r-- 1 user user 20 Apr 30 12:12 new_script
```

Wir sehen, dass die Berechtigungen für diese Datei **standardmäßig auf `664` gesetzt** sind. Wir haben für diese Datei noch **keine *Ausführungsrechte*** gesetzt.

```
$ chmod +x new_script
$ ls -l new_script
-rwxrwxr-x 1 user user 20 Apr 30 12:12 new_script
```

Dieser Befehl hat *allen* Benutzern Ausführungsrechte gewährt. Beachten Sie, dass dies ein Sicherheitsrisiko darstellen könnte, aber im Moment ist dies eine akzeptable Berechtigungsstufe.

```
$ ./new_script
Hello World!
```

Wir sind nun in der Lage, unser Skript auszuführen.

Definition des Interpreters

Wie wir gezeigt haben, konnten wir einfach Text in eine Datei schreiben, die Datei ausführbar machen und ausführen**. `new_script` ist funktional immer noch eine normale Textdatei, aber wir haben es geschafft, sie von Bash interpretieren zu lassen**. **Aber was ist, wenn sie in Perl oder Python geschrieben ist?**

Es ist sehr empfehlenswert, den **Typ des Interpreters anzugeben**, den wir in der **ersten Zeile** eines Skripts verwenden möchten. Diese Zeile wird ***BangLine*** oder häufiger ***Shebang*** genannt und zeigt dem System an, wie diese Datei ausgeführt werden soll. Da wir Bash lernen, werden wir den absoluten Pfad zu unserer ausführbaren Bash-Datei verwenden, wiederum mit `which`:

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

Unser Shebang beginnt mit einem **Hashzeichen und einem Ausrufezeichen**, gefolgt **vom absoluten Pfad** oben. Öffnen wir `new_script` in einem Texteditor und fügen den Shebang ein. Nutzen wir die Gelegenheit, einen *Kommentar* in unser Skript einzufügen, Kommentare werden vom Interpreten ignoriert und für andere Benutzer geschrieben, die Ihr Skript verstehen wollen.

```
#!/bin/bash

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

echo "Hello World!"
```

Wir werden noch eine weitere Änderung am Dateinamen vornehmen: Wir speichern diese Datei als **`new_script.sh`**. Das **Dateisuffix** **`.sh`** ändert die Ausführung der Datei in keiner Weise, es ist eine Konvention, dass Bash-Skripte mit **`.sh` oder `.bash`** gekennzeichnet werden, um sie leichter zu identifizieren, so wie **Python-Skripte** normalerweise mit dem **Suffix** **`.py`** identifiziert werden.

Häufig genutzte Texteditoren

Linux-Anwender müssen oft in einer Umgebung arbeiten, in der grafische Texteditoren nicht zur Verfügung stehen, weshalb es dringend empfohlen wird, sich zumindest mit der Bearbeitung von Textdateien über die Befehlszeile vertraut zu machen. Zwei der **gängigsten Texteditoren** sind `vi` und `nano`.

vi

`vi` ist ein ehrwürdiger Texteditor und wird standardmäßig auf fast jedem Linux-System installiert. Von `vi` gibt es einen Klon namens *vi IMproved* oder `vim`, der einige Funktionen hinzufügt, aber die Oberfläche von `vi` beibehält. Während die Arbeit mit `vi` für einen neuen Benutzer entmutigend ist, ist der Editor verbreitet und beliebt bei Benutzern, die seine vielen Funktionen kennen.

Der wichtigste Unterschied zwischen `vi` und Anwendungen wie Notepad besteht darin, dass **`vi` drei verschiedene Modi** hat: **Beim Start** werden die **Tasten `H`, `J`, `K` und `L` zum Navigieren und nicht zum Tippen verwendet**. **`I` können** Sie in diesem Navigationsmodus drücken, um in den ***Einfügemodus*** zu gelangen. Darin können Sie normal tippen. Um den Einfügemodus zu verlassen, drücken Sie **`Esc`**, um zum *Navigationsmodus* zurückzukehren. Im **Navigationsmodus** können **Sie `:` drücken, um in den *Befehlsmodus*** zu gelangen. In diesem Modus können Sie speichern, löschen, beenden oder Optionen ändern.

Während `vi` eine Lernkurve hat, erlauben die verschiedenen Modi mit der Zeit einem versierten Benutzer, effizienter zu werden als mit anderen Editoren.

nano

`nano` ist ein neueres Werkzeug, das einfacher zu bedienen ist als `vi`. `nano` hat **keine unterschiedlichen Modi;** ein Benutzer kann beim Start mit der Eingabe beginnen und verwendet `Strg`, um auf die am unteren Bildschirmrand gedruckten Werkzeuge zuzugreifen.

```
                          [ Welcome to nano.  For basic help, type Ctrl+G. ]
^G Get Help   ^O Write Out  ^W Where Is   ^K Cut Text   ^J Justify    ^C
Cur Pos    M-U Undo
^X Exit       ^R Read File  ^\ Replace    ^U Uncut Text ^T To Spell   ^_ Go
To Line M-E Redo
```

Texteditoren sind eine Frage der persönlichen Präferenz. Der Editor, den Sie verwenden, wird keinen Einfluss auf diese Lektion haben, aber sich in Zukunft mit einem oder mehreren Texteditoren vertraut zu machen, wird sich auszahlen.

Variablen

Variablen sind ein wichtiger Bestandteil jeder Programmiersprache, und Bash ist da nicht anders. Wenn Sie eine neue Sitzung vom Terminal aus starten, setzt die Shell bereits einige Variablen, wie z.B. die Variable `PATH`. Man nennt sie ***Umgebungsvariablen* (*environment variables*),** da sie in der Regel Eigenschaften unserer Shell-Umgebung definieren. Sie können Umgebungsvariablen ändern und hinzufügen, aber vorerst konzentrieren wir uns auf das Setzen von Variablen in unserem Skript.

Wir werden unser Skript wie folgt verändern:

```
#!/bin/bash

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

username=Carol

echo "Hello $username!"
```

In diesem Fall haben wir eine ***Variable* namens `username`** erstellt und haben ihr den *Wert* `Carol` zugewiesen. Beachten Sie, dass zwischen dem Variablennamen, dem Gleichheitszeichen und dem zugewiesenen Wert keine Leerzeichen stehen.

In der nächsten Zeile haben wir den Befehl `echo` mit der Variablen verwendet, aber vor dem Variablennamen steht ein **Dollarzeichen (`$`),** was wichtig ist, da es der Shell

anzeigt, dass wir `username` als Variable und nicht nur als normales Wort behandeln wollen. Indem wir `username` in unseren Befehl eingeben, geben wir an, dass wir eine *Substitution* durchführen wollen, indem wir den *Namen* einer Variablen durch den dieser Variablen zugeordneten *Wert* ersetzen.

Wenn wir das neue Skript ausführen, erhalten wir diese Ausgabe:

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

- Variablen 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++:

```
#!/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
```

Verwendung von **Anführungszeichen bei Variablen**

Nehmen wir die folgende Änderung am Wert unserer Variable `username` vor:

```
#!/bin/bash

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

username=Carol Smith

echo "Hello $username!"
```

Die Ausführung dieses **Skripts führt zu einem Fehler**:

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

Beachten Sie, dass Bash ein Interpreter ist, und als solcher *interpretiert* es unser Skript

Zeile für Zeile. In diesem Fall interpretiert es `username=Carol` korrekt, um eine Variable `username` mit dem Wert `Carol` zu setzen. Aber dann **interpretiert es das Leerzeichen als Ende dieser Zuordnung** und **`Smith` als den Namen eines Befehls**. Um auch das Leerzeichen und den Namen `Smith` als Wert unserer Variable hinzuzufügen, setzen wir doppelte Anführungszeichen (`"`) um den Namen.

```
#!/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!
```

Eine wichtige Sache, die in Bash zu beachten ist: **Doppelte und einfache Anführungszeichen (`'`) verhalten sich sehr unterschiedlich**. **Doppelte Anführungszeichen gelten als "schwach"**, weil sie es dem Interpreter ermöglichen, eine **Ersetzung** innerhalb der Anführungszeichen **durchzuführen**. **Einfache Anführungszeichen gelten als "stark", weil sie jede Ersetzung verhindern**:

```
#!/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!
```

Im zweiten Befehl `echo` wurde verhindert, dass der Interpreter `$username` durch `Carol Smith` ersetzt, so dass die Ausgabe wörtlich genommen wird.

Argumente

Sie sind bereits mit der Verwendung von Argumenten in den Linux-Kerndienstprogrammen vertraut, z.B. enthält `rm testfile` sowohl die ausführbare Datei `rm` als auch ein Argument `testfile`. Argumente können bei der Ausführung an das Skript übergeben werden und ändern das Verhalten des Skripts. Sie sind einfach zu implementieren.

```
#!/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 Variablen `$1` zu. Diese verweist auf den Wert des *ersten Arguments*.

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

Die ersten neun Argumente werden auf diese Weise behandelt. Es gibt Möglichkeiten, mehr als neun Argumente zu verarbeiten, aber das liegt außerhalb des Rahmens dieser Lektion. Wir werden ein Beispiel mit nur zwei Argumenten zeigen:

```
#!/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!
```

Bei der Verwendung von Argumenten gibt es eine wichtige Überlegung: Im obigen Beispiel gibt es **zwei Argumente `Carol` und `Dave`, die jeweils `$1` und `$2` zugeordnet** sind. **Fehlt beispielsweise das zweite Argument**, wirft die Shell keinen Fehler, der Wert von **`$2` ist einfach *null* oder gar nichts**.

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

In unserem Fall wäre es eine gute Idee, unserem Skript eine gewisse Logik beizufügen, damit verschiedene *Bedingungen* die *Ausgabe* beeinflussen. Wir beginnen mit der Einführung einer weiteren hilfreichen Variablen und gehen dann zur Erstellung von *if-Anweisungen* über.

Anzahl der Argumente zurückgeben

Während Variablen wie `$1` und `$2` den Wert von Positionsargumenten enthalten, enthält eine andere **Variable `$#` die *Anzahl der Argumente***.

```
#!/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.
```

Bedingungslogik (Conditional Logic)

Bedingungslogik ist in der Programmierung ein weites Feld und wird in dieser Lektion nicht ausführlich behandelt. Wir konzentrieren uns auf die *Syntax* bedingter Anweisungen in Bash, die sich von den meisten anderen Programmiersprachen unterscheidet.

Beginnen wir mit einem einfachen Skript, das eine Begrüßung an einen einzelnen Benutzer ausgeben soll. Bei etwas anderem als einem Benutzer, soll es eine Fehlermeldung ausgeben.

- 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.

Nun, da die Logik klar ist, wenden wir uns der *Syntax* zu, die zur Implementierung dieser Logik erforderlich ist.

```
#!/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: $#."
```

Die Bedingungslogik liegt zwischen **`if`** **und** **`fi`**. Die zu **testende Bedingung** befindet sich **zwischen eckigen Klammern `[ ]`,** und die Maßnahmen, die zu ergreifen sind, wenn die Bedingung wahr ist, werden nach `then` angezeigt. Beachten Sie die Leerzeichen zwischen den eckigen Klammern und der enthaltenen Logik. Das Weglassen dieser Leerzeichen führt zu Fehlern.

Dieses Skript gibt entweder unsere Begrüßung *oder* die Fehlermeldung aus. Aber es gibt immer die Zeile `Number of arguments` aus.

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

Beachten Sie die `if`-Anweisung: Wir haben mit **`-eq`** **einen *numerischen Vergleich*** durchgeführt. In diesem Fall testen wir, ob der **Wert von `$#` *gleich* eins** ist. Die anderen Vergleiche, die wir durchführen können, sind:

`-ne` Nicht gleich

`-gt` Größer als

`-ge` Größer oder gleich

`-lt` Kleiner als

`-le` Kleiner oder gleich