UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

INSTITUT FÜR COMPUTER-LINGUISTIK

Einführung in die Nutzung computerlinguistischer Ressourcen

Sariya Karimova

16. April 2018

Material in Teilen von L. Jehl, N. Reiter, H. Schamoni und S. Martschat

Organisatorisches

- Ziel: Vertrautheit mit Daten und Tools zur Bearbeitung von Softwareprojekten erlangen
- Integrierte praktische Übungen
- Zeiten: Montag bis Freitag, 10:00 Uhr bis 13:00 Uhr und 14:00 Uhr bis 17:00 Uhr
- Slides und Übungen verfügbar unter `http://www.cl.uni-heidelberg.de/courses/ss18/ressourcen/`
- falls Sitzung verpasst: Übungen bitte nachträglich bearbeiten und an `karimova@cl.uni-heidelberg.de` senden

Der Plan

	Montag	Dienstag	Mittwoch	Donnerstag	Freitag
Vormittag	Linux I	Korpora, Formate	Parser, Tagger	Python für ML	CQP
Nachmittag	Linux II	VCS, Makefiles	Python für NLP	Weka	Große Datenmengen

Montag: Linux I

1. Linux I
   - Terminal, Shell, Befehle
   - Prozesse, Streams und Pipes
   - Reguläre Ausdrücke

Linux I

Motivation

- viele Tools der Computerlinguistik sind für Linux entwickelt
- Linux stellt mächtige Kommandozeilentools zur Verfügung
- nützlich für automatisierte
  - Datenorganisation
  - Datenaufbereitung
  - Extraktion von Informationen

Terminal vs Shell

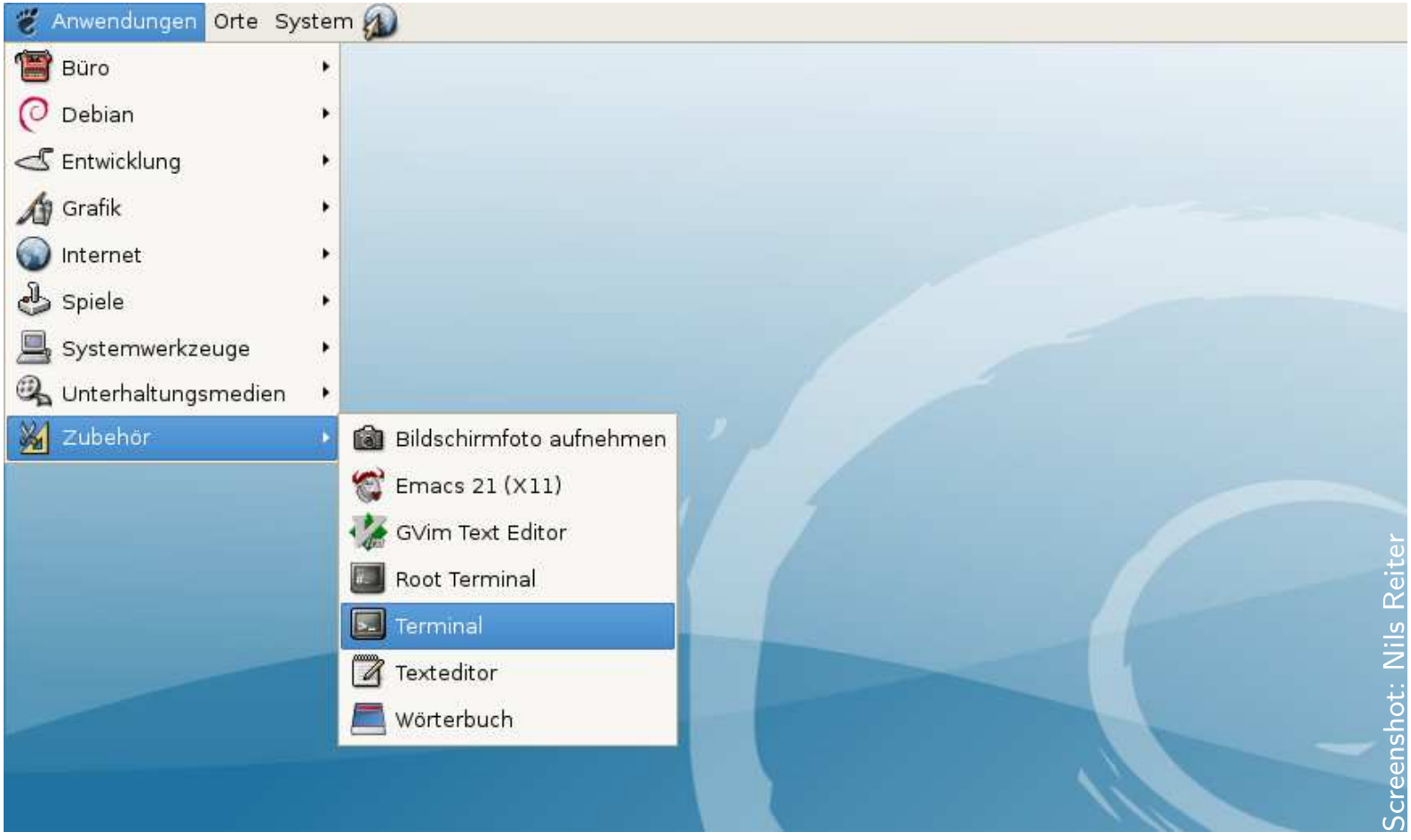
- Benutzerschnittstelle zu Computern mit Unix-Systemen (z.B. Linux, Mac OS X)
- Innerhalb des Terminals läuft eine Shell.
- Die Shell ist ein Programm, das unsere Eingaben entgegennimmt, interpretiert und evtl. andere Programme startet.
- Es gibt viele unterschiedliche Shells.
- Wir benutzen ausschließlich die Bourne-Again shell: bash.

Terminal Ein *Gerät*, das Interaktion mit einem Computer erlaubt

Shell Ein *Programm*, das auf einem Computer läuft, Befehle entegegennimmt und interpretiert

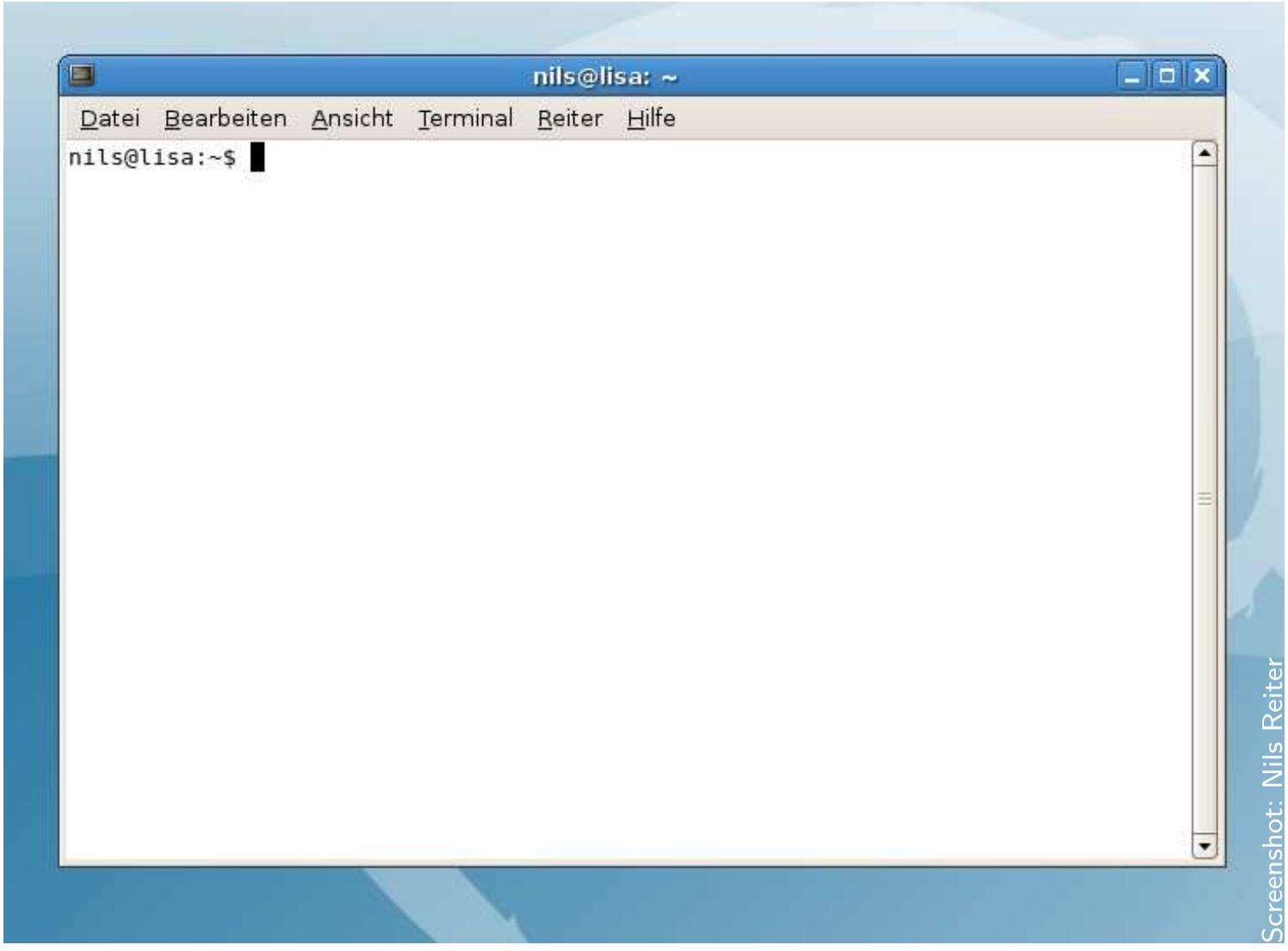
Starten und Beenden des Terminalemulators

Starten: Anwendungen → Zubehör → Terminal



Beenden: `exit` eingeben und Enter drücken

Das Gnome-Terminal



Befehle

- Befehle werden in Textform eingegeben und der Reihe nach ausgeführt.
- Befehle haben
  - einen Namen
  - Optionen (*wie der Befehl ausgeführt wird*)
  - Argumente (*worauf der Befehl angewendet wird*)
  - Optionen und Argumente sind Parameter eines Befehls
- Befehle sind eigene Programme (z.B. `less`, `wc`) oder Builtins, also Teile der Shell (z.B. `cd`, `echo`).

Beispiel

```
$ ls -l /
```

`$` Platzhalter für den Prompt

Leerzeichen

`ls` Der eigentliche Befehl

Leerzeichen

`-l` Option(en): Ändern das Verhalten des Befehls

Leerzeichen

`/` Argument(e): Datei/Verzeichnis, auf die/das der Befehl angewendet werden soll

Erst nach dem Druck auf 

wird der Befehl ausgeführt

Grundlegende Befehle

Navigieren im Verzeichnisbaum

`ls`	Zeigt Verzeichnisinhalt an
`cd`	Wechselt das aktuelle Verzeichnis
`pwd`	Gibt das aktuelle Verzeichnis aus
`mkdir`	Legt ein neues Verzeichnis an
`rmdir`	Löscht ein leeres Verzeichnis

Lange und kurze Optionen

Kurze Optionen

- Beginnen mit einem einfachen Strich
- Sind nur ein Zeichen lang: `-l`
- Mehrere können zusammengefasst werden: `-la` statt `-l -a`

Lange Optionen

- Beginnen mit zwei Strichen
- Sind länger als ein Zeichen: `--help`

- Viele Optionen gibt es sowohl in kurz als auch in lang (man page)

Lange und kurze Optionen

Kurze Optionen

- Beginnen mit einem einfachen Strich
- Sind nur ein Zeichen lang: `-l`
- Mehrere können zusammengefasst werden: `-la` statt `-l -a`

Lange Optionen

- Beginnen mit zwei Strichen
- Sind länger als ein Zeichen: `--help`

- Viele Optionen gibt es sowohl in kurz als auch in lang (man page)

Lange und kurze Optionen

Kurze Optionen

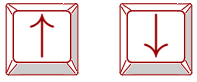
- Beginnen mit einem einfachen Strich
- Sind nur ein Zeichen lang: `-l`
- Mehrere können zusammengefasst werden: `-la` statt `-l -a`

Lange Optionen

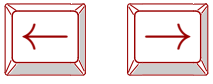
- Beginnen mit zwei Strichen
- Sind länger als ein Zeichen: `--help`

- Viele Optionen gibt es sowohl in kurz als auch in lang (man page)

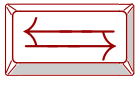
Tastenkürzel



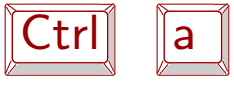
Blättern in der Befehlshistorie



Editieren des aktuellen Befehls



Automatisches Ergänzen von Datei- und Verzeichnisnamen



Springe zum Anfang der Zeile

Springe zum Ende der Zeile

Mehr Befehle

`less`	Zeigt den Inhalt von Textdateien auf dem Terminal an ( zum beenden)
`cp`	Kopiert Dateien oder Verzeichnisse
`mv`	Verschiebt Dateien oder Verzeichnisse Wird auch zum Umbenennen verwendet
`rm`	Löscht Dateien
`cat`	Gibt den Dateieinhalt im Terminal aus
`zip/unzip` / `tar`	Packt bzw. entpackt ein Archiv
`sleep`	Wartet einen angegebenen Zeitraum

- Es gibt *keine* vollständige Liste aller Befehle.
- Oft muss man viel lesen um einen passenden Befehl zu finden.

Remote arbeiten: SSH

- `ssh` stellt eine *sichere* Verbindung zu einem anderen Computer her
- Argument: `hostname`
- Optionen:
  - `-l <user>` Der Username auf dem anderen Computer
  - `-X` Erlaubt das Benutzen graphischer Programme
- Ist der Username auf beiden Computern gleich, kann er weggelassen werden (das ist am ICL der Fall!)

Beispiele

- `$ ssh ella -l karimova`
- `$ ssh karimova@ella`
- `$ ssh ella`

Remote arbeiten: SCP

- `scp` erlaubt das Kopieren von Dateien über eine sichere Verbindung
- Wo es ssh gibt, gibt es auch scp
- Wie bei cp müssen zwei Argumente genannt werden: Quelle und Ziel
- Zusätzlich zum Pfad wird ein Rechnername angegeben

Beispiele

- $ scp karimova@ella:/path/to/file .
- $ scp ~/path/to/file karimova@ella:/path/to/directory

Hilfe zur Selbsthilfe

- Man weiß nie alle Optionen auswendig
- Wichtig: Man kann Optionen nachschauen
- Viele Programme zeigen selbständig Hilfe an: **`-h`**, **`-help`** oder **`--help`**
- Viele Programme bieten eine Man-page an:

```
:~$ man <Programmname>
```

- Hilfe zur Kommandozeile insgesamt:
  `http://tldp.org/LDP/abs/html/` oder

```
:~$ man bash
```

Notation

Einzelne Befehle

- sind in **`Maschinenschrift`** gesetzt

Vollständige Eingaben

- haben einen Prompt, sind in `Maschinenschrift` und grau hinterlegt

```
:~$ Befehl
```

Prompt

```
karimova@ella:~$ _
```

karimova	Username (Benutzername)
@	
ella	Hostname (Rechnername)
:	
~	Working Directory (Aktuelles Verzeichnis)
$	
	Leerzeichen
_	Blinkender Cursor. Hier werden Befehle hingeschrieben

Dateien und Verzeichnisse

- Hierarchische Struktur
- Zu jeder Zeit gibt es ein aktuelles Verzeichnis: *working directory*
- Pfade sind absolut (beginnen mit '/') oder relativ (beginnen *nicht* mit '/')
- Jeder User hat ein *home directory*
  - Nach dem Einloggen befindet man sich im *home directory*
  - Dort kann man schreiben und lesen wie man möchte

Pfade eingeben

Platzhalter

- `.` Aktuelles Verzeichnis
- `..` Ein Verzeichnis weiter oben im Baum
- `~` Home-Verzeichnis
- `/` Oberstes Verzeichnis (root)

Wildcards

- `?` Steht für *ein* beliebiges Zeichen
  `h?llo` passt auf `hallo` oder `hello`, nicht aber auf `halllo`
- `*` Steht für beliebig viele beliebige Zeichen
  `*.txt` passt auf alle Dateinamen, die auf `.txt` enden.
  `a*` passt auf alle Dateinamen, die mit `a` beginnen.

Pfade eingeben

Platzhalter

- `.` Aktuelles Verzeichnis
- `..` Ein Verzeichnis weiter oben im Baum
- `~` Home-Verzeichnis
- `/` Oberstes Verzeichnis (root)

Wildcards

- `?` Steht für *ein* beliebiges Zeichen
  `h?llo` passt auf `hallo` oder `hello`, nicht aber auf `halllo`
- `*` Steht für beliebig viele beliebige Zeichen
  `*.txt` passt auf alle Dateinamen, die auf `.txt` enden.
  `a*` passt auf alle Dateinamen, die mit `a` beginnen.

Linux I

Linux I

Prozesse und Programme

Programm Anweisungen, die nacheinander ausgeführt werden. In Dateien gespeichert; kompiliert oder interpretiert.

Prozess Ein gestartetes Programm läuft in einem eigenen Prozess. Prozess läuft in einem „Kontext": Rechte, Working Directory, Umgebungsvariablen, Dateideskriptoren, ...

Subprozesse

- Alle Prozesse auf einem Computer sind hierarchisch angeordnet.
- Wird ein neuer Prozess gestartet, läuft er als Subprozess des Prozesses, von dem er gestartet wurde.
- Der Kontext des Elternprozesses wird übernommen.
- Der Elternprozess wartet, bis der neue Prozess beendet ist.

Prozesse

Laufende Prozesse

- **ps** zeigt laufende Prozesse, die vom momentanen Benutzer im laufenden Terminal gestartet wurden.

Beispiel (`:~$ ps -jH`)

```
  PID  PGID   SID TTY          TIME CMD
10185 10185 10185 pts/0    00:00:01 bash
10387 10387 10185 pts/0    00:00:00   ps
```

bash Die Shell selbst

ps Das ps-Kommando

Option **-e** zeigt alle laufenden Prozesse (every)

Prozesse

Laufende Prozesse

- **ps** zeigt laufende Prozesse, die vom momentanen Benutzer im laufenden Terminal gestartet wurden.

Beispiel (`:~$ ps -jH`)

```
  PID  PGID   SID TTY          TIME CMD
10185 10185 10185 pts/0    00:00:01 bash
10387 10387 10185 pts/0    00:00:00   ps
```

bash Die Shell selbst

ps Das ps-Kommando

Option **-e** zeigt alle laufenden Prozesse (every)

Subprozesse erzeugen

Beispiel (`:~$ sleep 5m`)

`sleep` startet in einem Subprozess, der den gleichen Kontext hat wie die Shell.
Solange **`sleep`** läuft, wartet die Shell.

Beispiel (`:~$ sleep 5m &`)

Ein & am Ende des Kommandos schiebt den Prozess in den Hintergrund.
Die Shell wartet dann nicht, bis der Prozess beendet ist.

Subprozesse erzeugen

Beispiel (`:~$ sleep 5m`)

`sleep` startet in einem Subprozess, der den gleichen Kontext hat wie die Shell.
Solange **`sleep`** läuft, wartet die Shell.

Beispiel (`:~$ sleep 5m &`)

Ein & am Ende des Kommandos schiebt den Prozess in den Hintergrund.
Die Shell wartet dann nicht, bis der Prozess beendet ist.

Subprozesse managen

- Das Kommando `jobs` zeigt eine Liste aller Prozesse im Hintergrund an
- Mit `fg` und `bg` lassen sich Prozesse in den Hintergrund schieben oder in den Vordergrund holen
- Ein laufender Prozess kann mit Ctrl - z pausiert und mit Ctrl - c abgebrochen werden
- Achtung: damit ein pausierter Prozess weiterläuft, muss er mit `bg` in den Hintergrund geschoben werden

Subprozesse managen

- Das Kommando `jobs` zeigt eine Liste aller Prozesse im Hintergrund an
- Mit `fg` und `bg` lassen sich Prozesse in den Hintergrund schieben oder in den Vordergrund holen
- Ein laufender Prozess kann mit Ctrl - z pausiert und mit Ctrl - c abgebrochen werden
- Achtung: damit ein pausierter Prozess weiterläuft, muss er mit `bg` in den Hintergrund geschoben werden

Remote arbeiten

- Wird ein Prozess beendet, werden auch alle Subprozesse beendet
- Prozesse werden beendet, wenn die SSH-Verbindung abbricht (z.B. durch Schließen des Terminalfensters)
- Um Prozesse weiterlaufen zu lassen (etwa: die ganze Nacht / über das Wochenende), muss man Prozesse *vorher* richtig starten, zum Beispiel mit **`screen`** (`https://www.gnu.org/software/screen/`)

screen

- Erlaubt *persistente* Sitzungen
- Zwischenschicht zwischen Terminal und Rechner

Beispiel

```
$ ssh ella
ella$ screen
ella$ # do something
ella$ C-a d
[detached]
ella$ # do something else
ella$ screen -r
```

Es gibt Alternativen zu `screen`, z.B. `tmux`
(`https://github.com/tmux/tmux/wiki`)

screen

- Erlaubt *persistente* Sitzungen
- Zwischenschicht zwischen Terminal und Rechner

Beispiel

```
$ ssh ella
ella$ screen
ella$ # do something
ella$ C-a d
[detached]
ella$ # do something else
ella$ screen -r
```

Es gibt Alternativen zu `screen`, z.B. `tmux`
(`https://github.com/tmux/tmux/wiki`)

screen

- Erlaubt *persistente* Sitzungen
- Zwischenschicht zwischen Terminal und Rechner

Beispiel

```
$ ssh ella
ella$ screen
ella$ # do something
ella$ C-a d
[detached]
ella$ # do something else
ella$ screen -r
```

Es gibt Alternativen zu `screen`, z.B. `tmux`
(`https://github.com/tmux/tmux/wiki`)

Linux I

Ein- und Ausgabe

- Stream/Strom: Ein Kanal, in dem zeichenweise gelesen und geschrieben wird
- Drei Streams werden automatisch für jeden Prozess geöffnet:

  `STDOUT` Standardausgabe-Kanal – im Normalfall das Terminal

  `STDIN` Standardeingabe-Kanal, von dem der Prozess lesen kann – im Normalfall sind das Tastatureingaben

  `STDERR` Standard-Fehlerausgabe – Fehlermeldungen können entweder angezeigt oder gespeichert werden (oder beides)

Standardstreams

Beispiel (Terminal)

`STDOUT` Das Fenster am Bildschirm

`STDERR` Das Fenster am Bildschirm, allerdings ungepuffert
gepuffert=Ausgabe wird erst angezeigt, wenn Newline ausgegeben wird oder exit/fflush aufgerufen wurde -> d.h. stderr wird sofort ausgegeben!

`STDIN` Die Tastatur

Prozesse, die im Terminal gestartet werden, „erben" diese Eigenschaften, wenn sie nicht explizit umgeleitet werden

Streams umleiten I

Ausgabe umleiten

Mit dem Zeichen $>$ am Ende eines Kommandos kann man die (Standard-)Ausgabe des Kommandos in eine Datei umleiten

Beispiel (`:~$ ps > processes.txt`)

Keine Ausgabe mehr im Terminal, stattdessen stehen die Informationen jetzt in der Datei **`processes.txt`**, wovon man sich mit **`less`** überzeugen kann.

Streams umleiten II

Eingabe umleiten

Mit dem Zeichen $<$ am Ende eines Kommandos wird die Standardeingabe des Kommandos aus einer Datei gefüllt

Beispiel (`:~$ sort < processes.txt`)

```
  26331   pts/9    00:00:00   bash
  31920   pts/9    00:00:00   ps
    PID     TTY        TIME   CMD
```

Streams umleiten III

Umleitungen

`>`	`STDOUT` umleiten, alte Datei wird gelöscht
`>>`	`STDOUT` umleiten, an alte Datei wird angehängt
`<`	`STDIN` umleiten
`2>`	`STDERR` umleiten
`&>`	`STDOUT` und `STDERR` umleiten
`2>&1`	`STDERR` an `STDOUT` senden

Pipe

Beispiel

```
:~$ ps > processes.txt
```

```
:~$ sort < processes.txt
```

Beispiel

```
:~$ ps | sort
```

Pipe

- Pipe | kombiniert verschiedene Befehle
- Prozess 2 bekommt STDOUT von Prozess 1 als STDIN

Pipe

Beispiel

```
:~$ ps > processes.txt
```

```
:~$ sort < processes.txt
```

Beispiel

```
:~$ ps | sort
```

Pipe

- Pipe | kombiniert verschiedene Befehle
- Prozess 2 bekommt STDOUT von Prozess 1 als STDIN

Linux I

Escaping

" und '

- Zeichenketten (*Strings*) werden mit Anführungszeichen " oder Apostrophen ' begrenzt
- Um Anführungszeichen oder Apostrophen als Zeichen zu verwenden, müssen sie *escaped* werden
- Escaping wird in Linux/Unix mit einem Backslash \ gemacht

Beispiele (Escaping)

- "My sister's friend's investments" ✓
- 'My sister\'s friend\'s investments' ✓
- 'My sister's friend's investments' X

Escaping

" und '

- Zeichenketten (*Strings*) werden mit Anführungszeichen " oder Apostrophen ' begrenzt
- Um Anführungszeichen oder Apostrophen als Zeichen zu verwenden, müssen sie *escaped* werden
- Escaping wird in Linux/Unix mit einem Backslash \ gemacht

Beispiele (Escaping)

- "My sister's friend's investments" ✓
- 'My sister\'s friend\'s investments' ✓
- 'My sister's friend's investments' X

Escaping

" und '

- Zeichenketten (*Strings*) werden mit Anführungszeichen " oder Apostrophen ' begrenzt
- Um Anführungszeichen oder Apostrophen als Zeichen zu verwenden, müssen sie *escaped* werden
- Escaping wird in Linux/Unix mit einem Backslash \ gemacht

Beispiele (Escaping)

- "My sister's friend's investments" ✓
- 'My sister\'s friend\'s investments' ✓
- 'My sister's friend's investments' X

Sonderzeichen

Nicht-druckbare Zeichen

- Bestimmte Zeichen des normalen Zeichensatzes sind *nicht druckbar*.
- Dazu gehören zum Beispiel Zeilenumbrüche, Wagenrückläufe, Tabulatoren oder die Backspace-Taste.
- Mit diesen Zeichen kann dennoch normal gearbeitet werden – Escaping macht es möglich.
- Für jedes dieser Steuerzeichen ist ein Escape Code der Form \<zeichen> definiert

Escape Codes

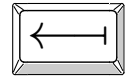
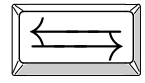
Escape Code	Name	Tastatursymbol
\b	Backspace	
\c	Control	
\f	Form feed	
\n	New line, return	
\r	Carriage return	
\t	Tab	
\\	Backslash	

Tabelle: Ausgewählte Escape Codes

Zeilenumbrüche

DOS/Windows	Carriage Return, New line	\r \n
Unix (Linux, Mac OS X)	New line	\n
Mac OS	Carriage Return	\r

Tabelle: Zeilenende nach Betriebssystem

Umwandlung

`dos2unix` , **`unix2dos`** , **`tr`** , **`sed`**

Editoren

- Text-Editoren unterstützen manuelle Texteingabe
- In unserem Kontext: Dokumentation, Quelltext, Daten, ...
- Zwei große Editoren: `Emacs` und `vi`
- Sie sollten sich mit einem von beiden beschäftigen.

Features

- Syntax-Highlighting für *alle* Programmiersprachen/Formate
- Suchfunktion (reguläre Ausdrücke!), Suchen und Ersetzen
- Copy-Cut-Paste, Undo
- Universelle Verfügbarkeit

Editoren

- Text-Editoren unterstützen manuelle Texteingabe
- In unserem Kontext: Dokumentation, Quelltext, Daten, ...
- Zwei große Editoren: `Emacs` und `vi`
- Sie sollten sich mit einem von beiden beschäftigen.

Features

- Syntax-Highlighting für *alle* Programmiersprachen/Formate
- Suchfunktion (reguläre Ausdrücke!), Suchen und Ersetzen
- Copy-Cut-Paste, Undo
- Universelle Verfügbarkeit

Linux I

Reguläre Ausdrücke

- Beschreiben eine Menge von Zeichenfolgen
- Ist eine gegebene Zeichenfolge Teil dieser Menge, *matcht* der Ausdruck auf die Zeichenfolge.

Anwendungsbeispiele

- Entfernen von Zeichen und Zeichenketten aus Texten
- Umwandeln von Zeichen in andere Zeichen
- Herausfinden, ob Zeichenketten ein bestimmtes Format haben

Reguläre Ausdrücke

- Beschreiben eine Menge von Zeichenfolgen
- Ist eine gegebene Zeichenfolge Teil dieser Menge, *matcht* der Ausdruck auf die Zeichenfolge.

Anwendungsbeispiele

- Entfernen von Zeichen und Zeichenketten aus Texten
- Umwandeln von Zeichen in andere Zeichen
- Herausfinden, ob Zeichenketten ein bestimmtes Format haben

Varianten

- Keine einheitliche Syntax für reguläre Ausdrücke
- Insbesondere Kommandozeilen-Tools nutzen oft verschiedene Varianten
- Wir schauen uns *POSIX* genauer an, da viele Kommandozeilentools POSIX oder eine Abwandlung von POSIX verwenden
- POSIX ist ein UNIX-Standard, der bestimmte grundlegende Funktionen und Eigenschaften von UNIX-Systemen definiert.

Linux I

Basic Regular Expressions

- Zeichen matchen auf sich selbst
- `.` matcht ein (!) beliebiges Zeichen
- `^` matcht den Anfang der Zeichenkette oder der Zeile
- `$` matcht das Ende der Zeichenkette oder der Zeile
- `*` matcht das vorherige Zeichen 0 oder mehr Male
- `\{m, n\}` matcht das vorherige Zeichen mindestens m und maximal n mal

BRE – Bracket Expression

- `[ ]` matcht jedes einzelne Zeichen, das zwischen den Klammern steht (bracket expression)
- `[^ ]` matcht jedes einzelne Zeichen, das *nicht* zwischen den Klammern steht

Beispiele

- Einzelne Zeichen: `[aeiou]`
- Nicht einzelne Zeichen: `[^a-d]`
- Bereiche: `[a-d0-3]`
- Einfacher Bindestrich am Anfang oder Ende: `[-a-z]`

BRE – Bracket Expression

- `[ ]` matcht jedes einzelne Zeichen, das zwischen den Klammern steht (bracket expression)
- `[^ ]` matcht jedes einzelne Zeichen, das *nicht* zwischen den Klammern steht

Beispiele

- Einzelne Zeichen: `[aeiou]`
- Nicht einzelne Zeichen: `[^a-d]`
- Bereiche: `[a-d0-3]`
- Einfacher Bindestrich am Anfang oder Ende: `[-a-z]`

BRE – Backreferences

- `\( \)` Ein Block für Backreferences
- `\n` matcht den n-ten Block ($1 \leq n \leq 9$)

Beispiele

- Zwei gleiche Buchstaben hintereinander: `\(.\)\1`
- Weiter entfernte Referenz: `\(.\)h\1` (aha, oho, uhu, ...)
- Maximal neun Blöcke

BRE – Backreferences

- `\( \)` Ein Block für Backreferences
- `\n` matcht den n-ten Block ($1 \leq n \leq 9$)

Beispiele

- Zwei gleiche Buchstaben hintereinander: `\(.\)\1`
- Weiter entfernte Referenz: `\(.\)h\1` (aha, oho, uhu, ...)
- Maximal neun Blöcke

BRE – Escaping

- Um ein Zeichen zu matchen, das im Ausdruck eine Bedeutung hat, muss es *escaped* werden
- Dazu benutzt man den Backslash: `\`
- Folgende Zeichen müssen escaped werden: `[`, `]`, `.`, `*`, `^`, `$`, `\`.
- Runde und geschweifte Klammern werden *nicht* escaped, da diese ja escaped verwendet werden.

Zeichenklassen

`[[:alnum:]]`	`[a-zA-Z0-9]`
`[[:alpha:]]`	`[a-zA-Z]`
`[[:digit:]]`	`[0-9]`
`[[:lower:]]`	`[a-z]`
`[[:punct:]]`	Punctuation and Symbols
`[[:space:]]`	`[\t\r\n\v\f]`
`[[:upper:]]`	`[A-Z]`
`[[:word:]]`	`[a-zA-Z0-9_]`

Tabelle: Zeichenklassen

Extended Regular Expressions

- Erweiterung zu BRE
- Meta-characters (`\(, \), \{ \}`) werden nicht mehr escaped.
  `\(` matcht jetzt "("anstatt einen Block aufzumachen
- `?` matcht das vorige Zeichen 0 oder 1 mal
- `+` matcht das vorige Zeichen 1 oder mehr male
- `a|b` matcht "a" oder "b"
- Bei vielen Kommandozeilentools werden Extended REs mit der Option `-E` aktiviert

Linux I

Matching

`grep`

- Mit **`grep`** kann man Dateien nach Vorkommen von Zeichenketten durchsuchen
- In der Standardeinstellung werden die Zeilen, in welchen die gesuchten Zeichenketten vorkommen, ausgegeben
- Die Zeichenketten können durch reguläre Ausdrücke definiert werden

Beispiel (`:~$ grep '^$' file.txt`)

Findet alle leeren Zeilen in `file.txt`

Ersetzung

`sed`

- Werkzeug, um Datenströme zu bearbeiten
- Insbesondere für Ersetzungsregeln mit regulären Ausdrücken nützlich
- Mit dem Ausdruck **`s/a/b/g`** wird jedes Vorkomen von a durch b ersetzt
- Modifikatoren können angehängt werden
- Statt / sind auch andere Zeichen möglich, z.B. `s#/ein/pfad/#/ein/anderer/pfad#g`

Beispiel (`:~$ sed 's/groß/klein/g' file.txt`)

Ersetzt jede in **`file.txt`** vorkommende Zeichenfolge „groß" durch die Zeichenfolge „klein"

Ersetzung – Beispiele

Beispiele

- `$ sed 's/[[:space:]]/\n/g' file.txt`
- `$ sed 's/\([[:space:]]\)[[:punct:]]/\1/g'`
- `$ sed 's/\([[:lower:]]\)\([[:upper:]]\)/\1 \2/g'`

Ersetzung – Beispiele

Beispiele

- `$ sed 's/[[:space:]]/\n/g' file.txt`
  Ersetzt jedes space-Zeichen durch einen Zeilenumbruch. Jedes „Wort” steht dann auf einer eigenen Zeile
- `$ sed 's/\([[:space:]]\)[[:punct:]]/\1/g'`
- `$ sed 's/\([[:lower:]]\)\([[:upper:]]\)/\1 \2/g'`

Ersetzung – Beispiele

Beispiele

- `$ sed 's/[[:space:]]/\n/g' file.txt`

- **`$ sed 's/\([[:space:]]\)[[:punct:]]/\1/g'`**

- `$ sed 's/\([[:lower:]]\)\([[:upper:]]\)/\1 \2/g'`

Ersetzung – Beispiele

Beispiele

- `$ sed 's/[[:space:]]/\n/g' file.txt`

- **`$ sed 's/\([[:space:]]\)[[:punct:]]/\1/g'`**
  Ersetzt Space gefolgt von Punctuation-Zeichen durch Space.
- `$ sed 's/\([[:lower:]]\)\([[:upper:]]\)/\1 \2/g'`

Ersetzung – Beispiele

Beispiele

- `$ sed 's/[[:space:]]/\n/g' file.txt`
- `$ sed 's/\([[:space:]]\)[[:punct:]]/\1/g'`
- **`$ sed 's/\([[:lower:]]\)\([[:upper:]]\)/\1 \2/g'`**

Ersetzung – Beispiele

Beispiele

- `$ sed 's/[[:space:]]/\n/g' file.txt`
- `$ sed 's/\([[:space:]]\)[[:punct:]]/\1/g'`
- **`$ sed 's/\([[:lower:]]\)\([[:upper:]]\)/\1 \2/g'`**
  Ersetzt CamelCase-Ausdrücke durch Camel Case

Übung 1