Werkzeuge für das wissenschaftliche Arbeiten

Python for Machine Learning and Data Science

Magnus Bender
bender@ifis.uni-luebeck.de
Wintersemester 2022/23

Inhaltsübersicht

1. Programmiersprache Python
   a) *Einführung, Erste Schritte*
   b) *Grundlagen*
   c) *Fortgeschritten*
2. Auszeichnungssprachen
   a) *LaTeX, Markdown*
3. Benutzeroberflächen und Entwicklungsumgebungen
   a) **Jupyter Notebooks lokal und in der Cloud (Google Colab)**
4. Versionsverwaltung
   a) Git, GitHub
5. Wissenschaftliches Rechnen
   a) NumPy, SciPy
6. Datenverarbeitung und -visualisierung
   a) Pandas, matplotlib, NLTK
7. Machine Learning (scikit-learn)
   a) Grundlegende Ansätze (Datensätze, Auswertung)
   b) Einfache Verfahren (Clustering, …)
8. DeepLearning
   a) TensorFlow, PyTorch, HuggingFace Transformers

Themen

I. Projektaufgabe 1

    1. Lösungsvorschlag

II. Benutzeroberflächen und Entwicklungsumgebungen

    1. Jupyter Notebooks

    2. Grundlagen (Bash-)Terminal

III. Reguläre Ausdrücke

Projektaufgabe 1

„Textbasierter Taschenrechner“

- Vorstellung möglicher Lösungen

(werden nicht in Moodle hochgeladen!)

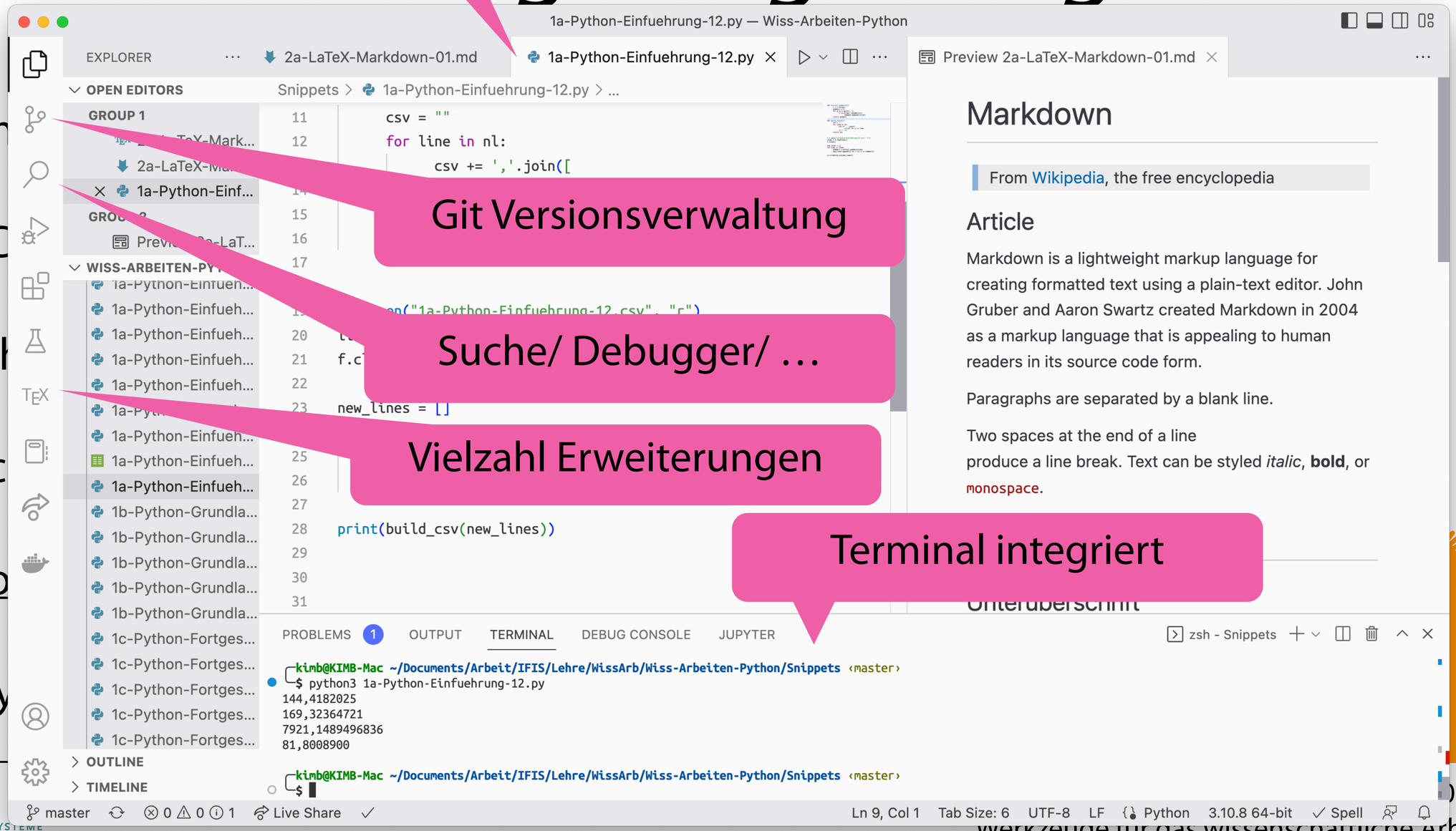
II.
Benutzeroberflächen und Entwicklungsumgebungen

1. Jupyter Notebooks – lokal und in der Cloud

Entwicklungsumgebungen

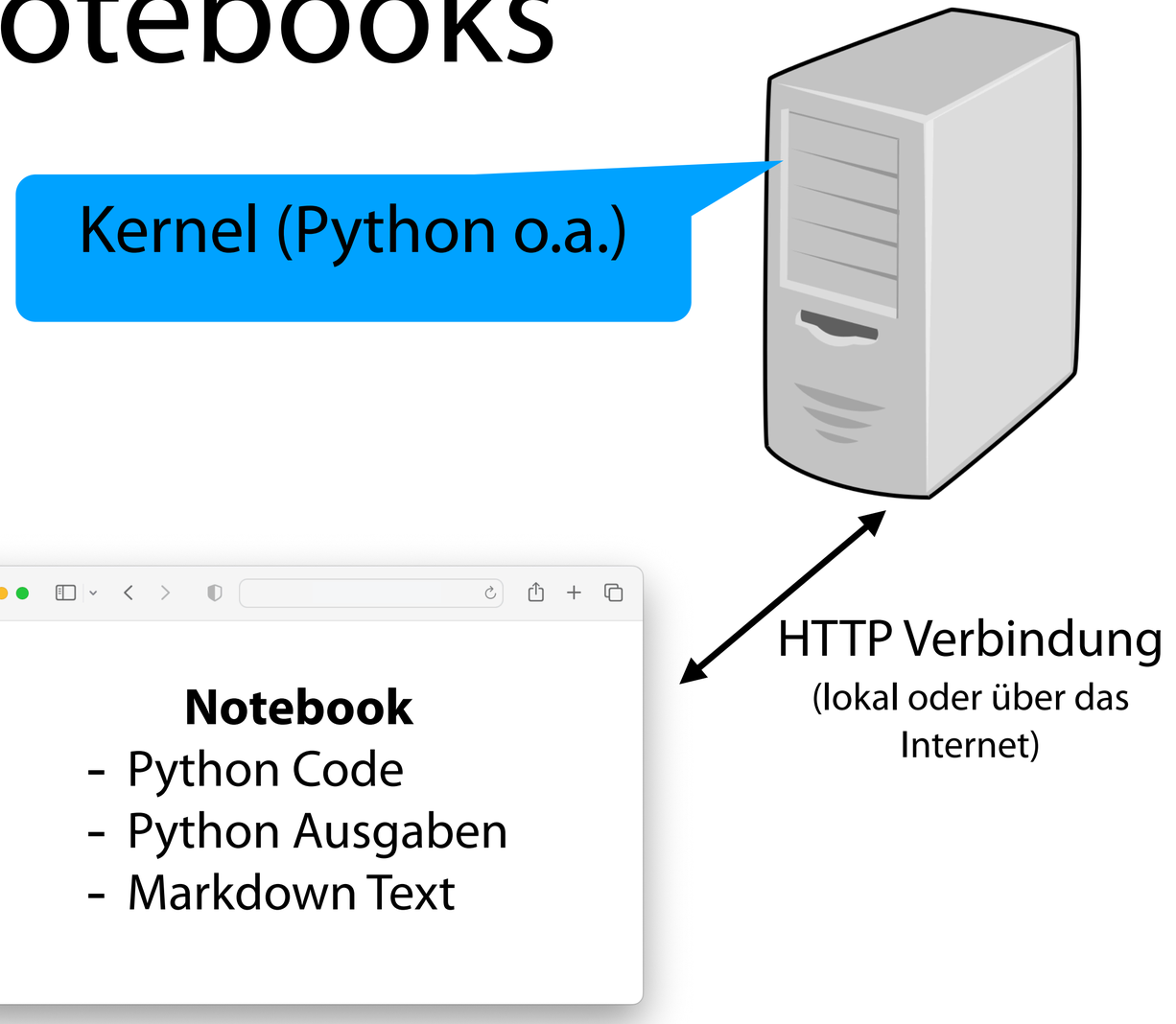
Editortabs mit verschiedenen Inhalten

- Vielzah
  - VS C
  - PyCh
- Entwic
  - vsco
  - Jupy



Jupyter Notebooks

- Darstellung im Browser (aber u.a. auch in VS Code)
- Installation lokal oder Nutzung in der Cloud
- Kombination von Programm und Text (Markdown)



Notebook Beispiel

Ein Notebook besteht aus mehreren Zellen, jede Zelle kann einen anderen Inhaltstypen haben:

- Python Code
- Markdown Text

Und es werden z.B. auch Formeln unterstützt!

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{n^2+n}{2}$$

In [1]:
```
import sys
sys.version
```

Out[1]:
```
'3.10.6 (main, Nov  2 2022, 18:53:38) [GCC 11.3.0]'
```

Einen Codeblock kann man mittels `Shift+Enter` ausführen, die Ausgabe erscheint dann direkt darunter (auch ohne `print()`). Übrigens, die Eingabe von Markdown beendet man auch mit `Shift+Enter`.

In [2]:
```
def summe(n):
    s = 0
    for i in range(n+1):
        s += i
    return s
```

Eine Funktion kann einfach definiert werden und ist dann im *laufenden* Kernel verfügbar, also in allen weiteren Codeblöcken definiert.

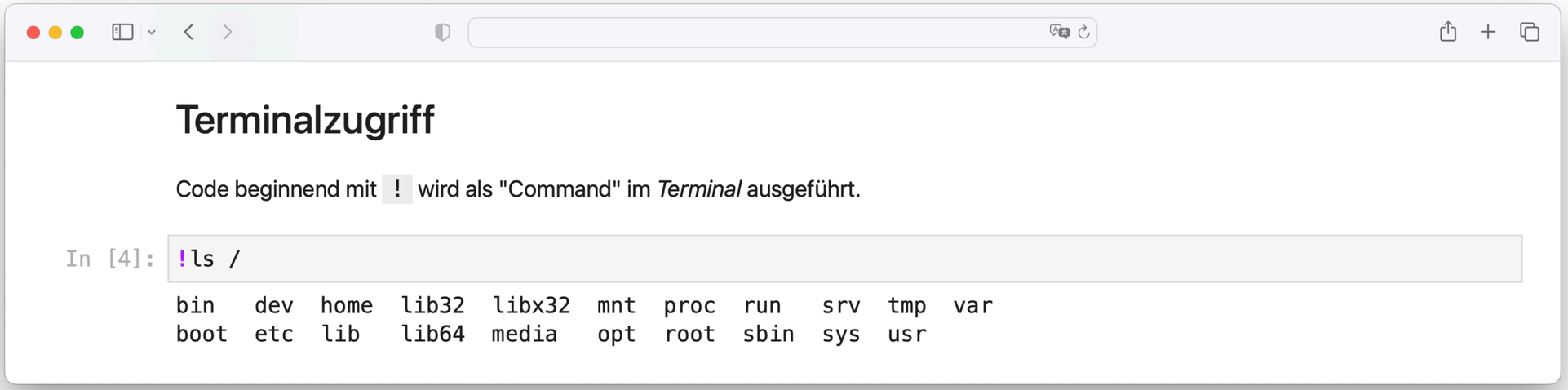
In [3]:
```
"Korrekt" if summe(10) == (10**2+10)/2 else "Fehlerhaft"
```

Out[3]:
```
'Korrekt'
```

Jupyter in der Cloud

- Beispiel: Google Colab(oratory)
  - Kostenfreie Nutzung
  - Zugriff auf GPUs

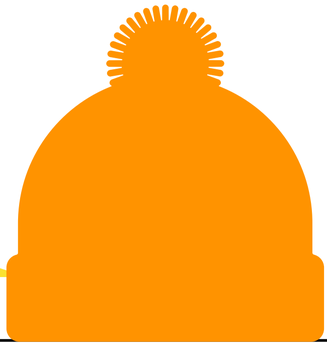
Praktisch bei der Entwicklung von Code und Erklärung bei gleichzeitiger Anzeige von Ausgaben.



II.
Benutzeroberflächen und Entwicklungsumgebungen

1. Die Shell und das Terminal

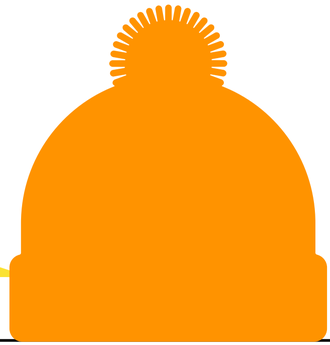
Wir hatten zuvor immer Beispiele mit Befehlen für das Terminal – aber haben das Terminal nie betrachtet.



Unixoides Betriebssystem

- Grundzüge
  - „Alles ist eine Datei“
  - „Unix-Shell“
  - Benutzerverwaltung (Admin = `root`)
  - Grafikserver X11
  - …
- Beispiele
  - Linux, macOS, BSD
  - Auf Windows Linux über die WSL (Windows Subsystem for Linux) möglich

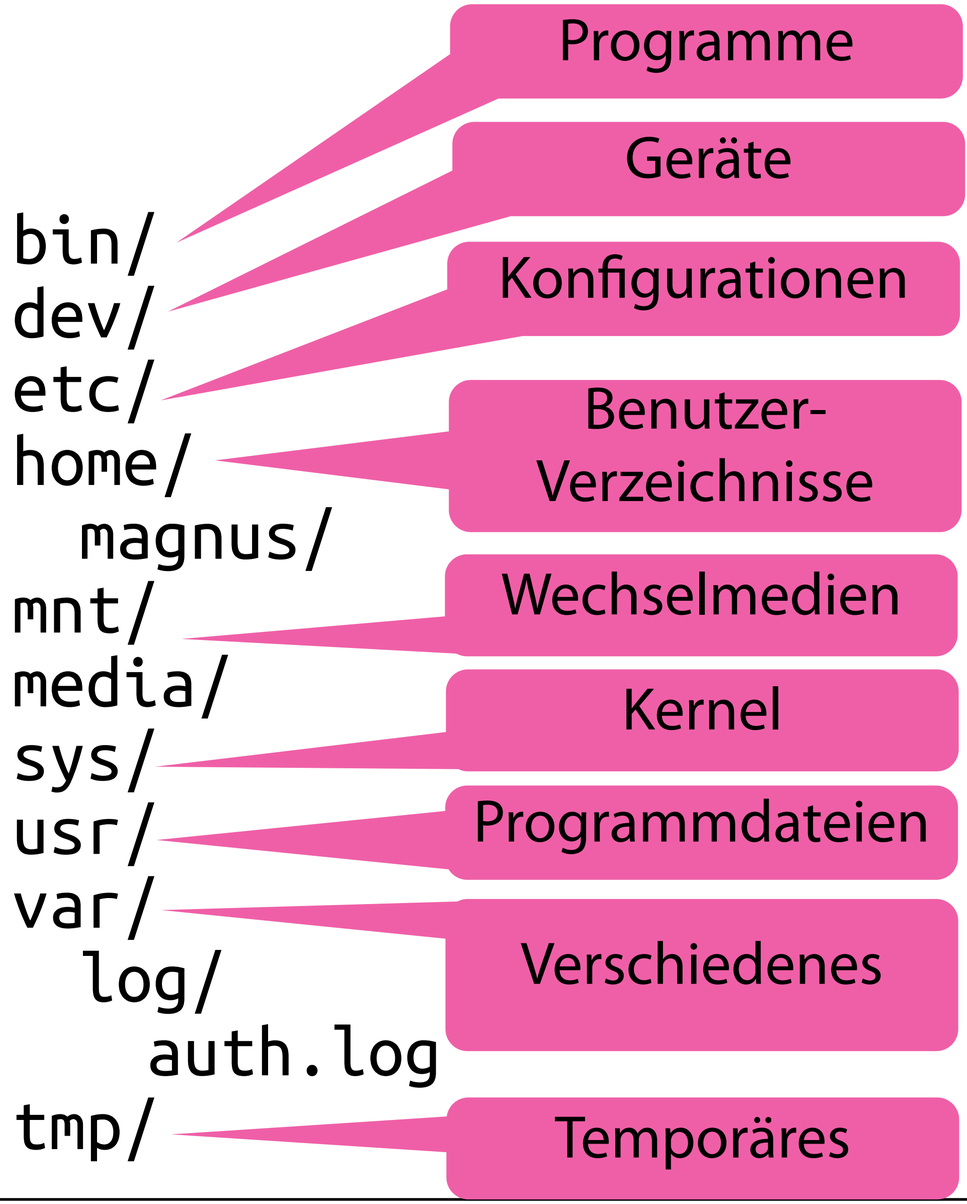
Wir beziehen uns im folgenden auf eine „Unix-Shell“



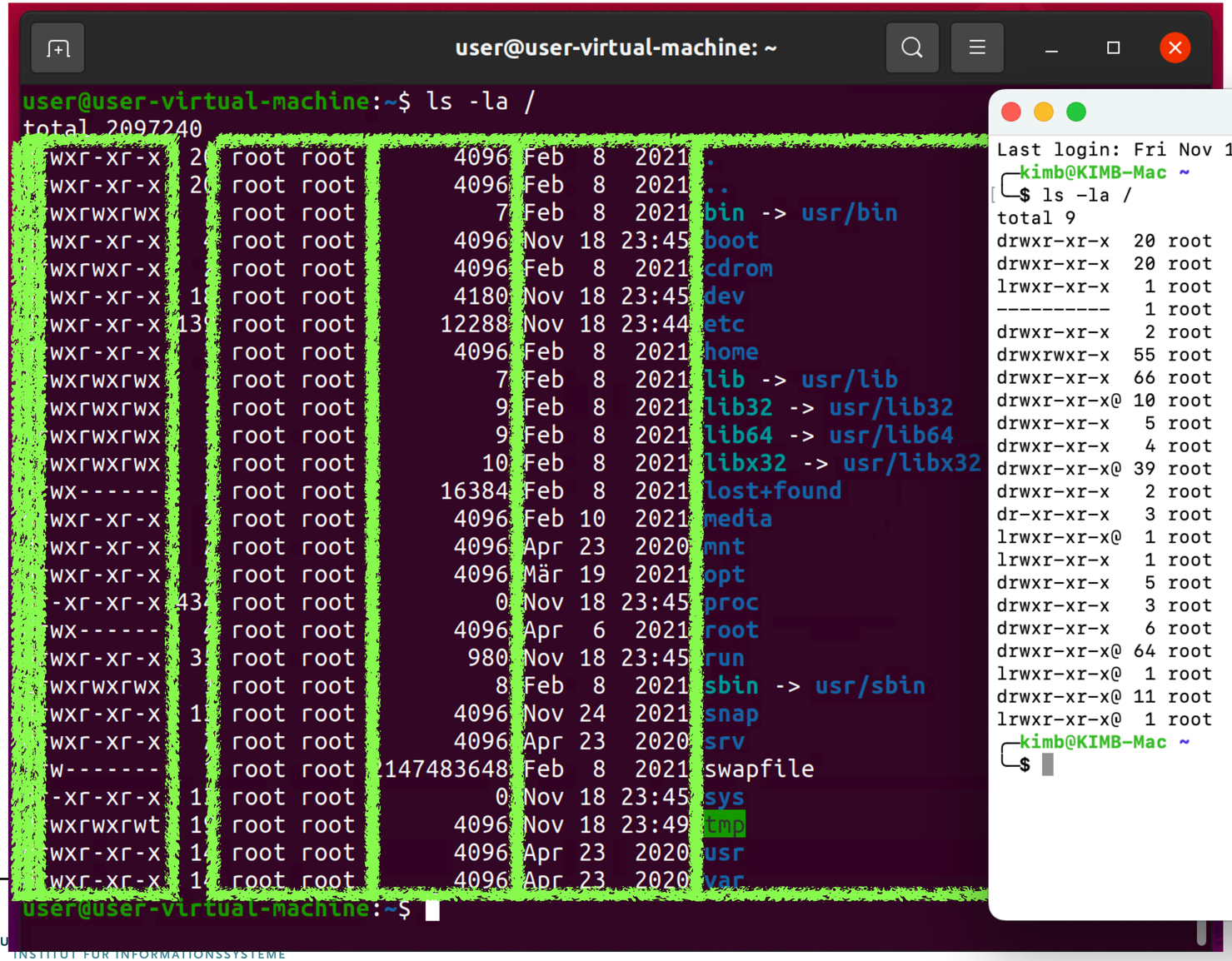
Pfade

- „/“ Wurzelpfad
- „.“ referenziert das aktuelle Verzeichnis
- „..“ referenziert das übergeordnete Verzeichnis
- „~“ referenziert das Home-Verzeichnis des aktuellen Users

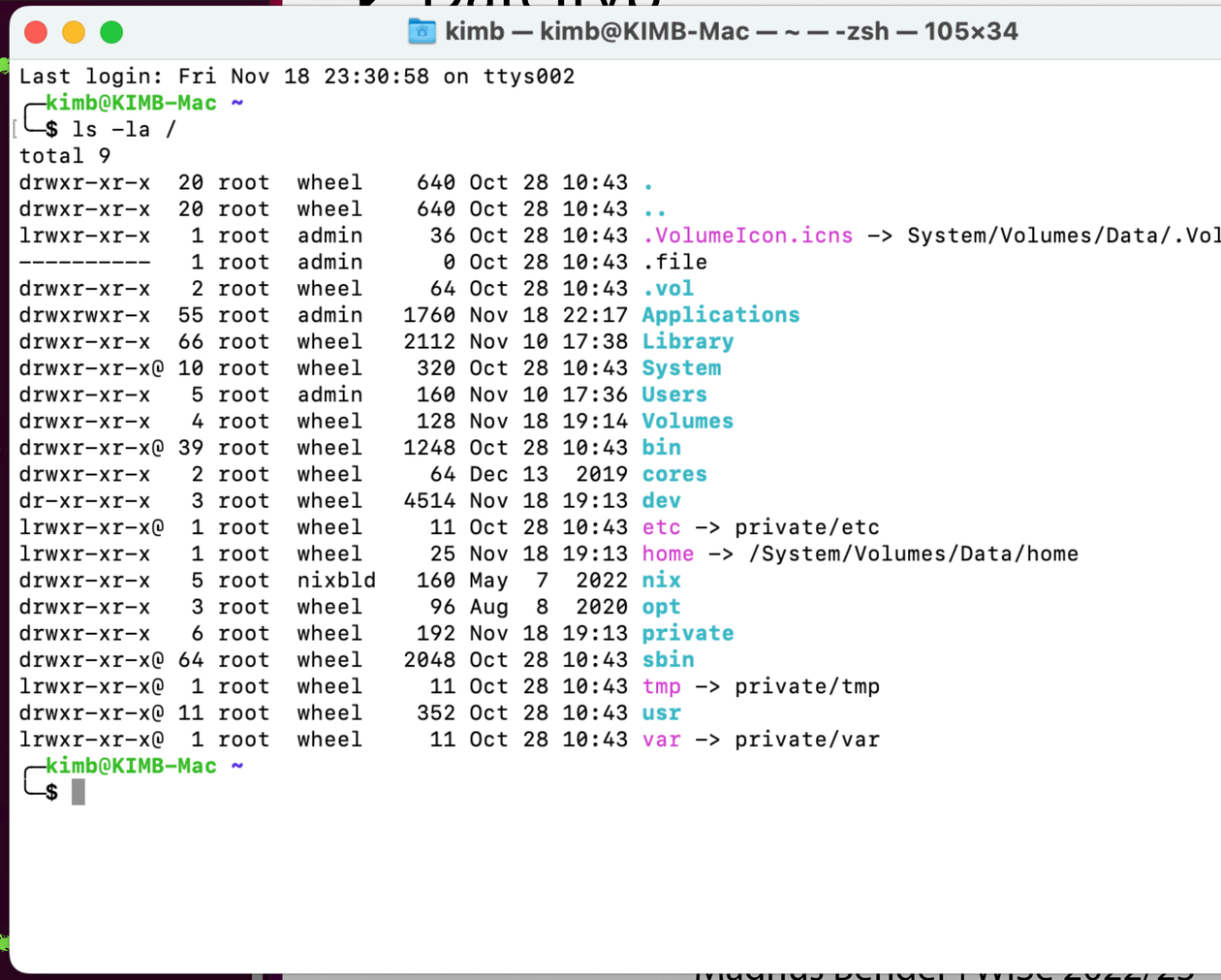
- Verstecke (Datei-)Namen beginnen mit „.“



Dateisystem



→ Dateityp



„Command“

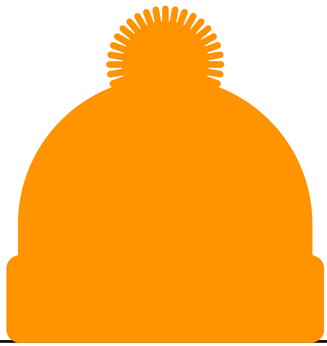
- Programmaufruf über das Terminal

Leerzeichen als Trenner

```
programm [[-p <wert>] --parameter <wert>] argument
```

  - Programmname
  - Optionale Parameter
  - Argumente

Häufig haben Parameter eine kurze (`-`) und eine Lange (`--`) Bezeichnung.



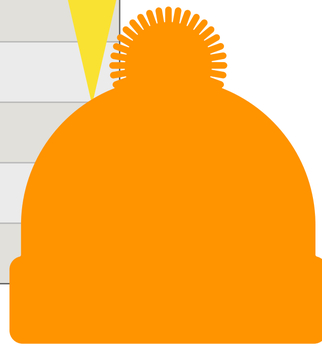
Bei kurzen Parametern kann man meist statt „`programm -a -b`“ auch „`programm -ab`“ ausführen.

Nützliche Befehle I

Navigation im Dateisystem	`cd`	`cd ..`	In einen anderen Ordner wechseln
	`ls`	`ls .`	Inhalte eines Ordners auflisten
	`pwd`	`pwd`	Aktuellen Pfad anzeigen
	`mc`	`mc .`	Midnight Commander (GUI-artiger Explorer)
	`find`	`find . -name "*.txt" -type f`	Suchen und finden von Dateien
	`grep`	`grep "hi" ./file.txt`	Suchen und finden in Dateien
			rdners
			rdners
			ns Terminal)
			len einer Datei
			en einer Datei
Dateien	`touch`	`to h ./new.txt`	Erstellen einer leeren Datei
	`mcedit`	`mc it ./file.txt`	Bearbeiten einer Datei (GUI-artig, von `mc`)
	`nano`	`nan ./file.txt`	Bearbeiten einer Datei (Quasi-Standard)
	`vim`	`vim ./file.txt`	Bearbeiten einer Datei
	`rm`	`rm ./file.txt`	Löschen einer Datei (oder auch Ordnern!)

Mit „`rm -rf /`“ würde man das ganze Dateisystem rekursiv löschen (sofern man die Berechtigung hat).

„How to exit Vim?“

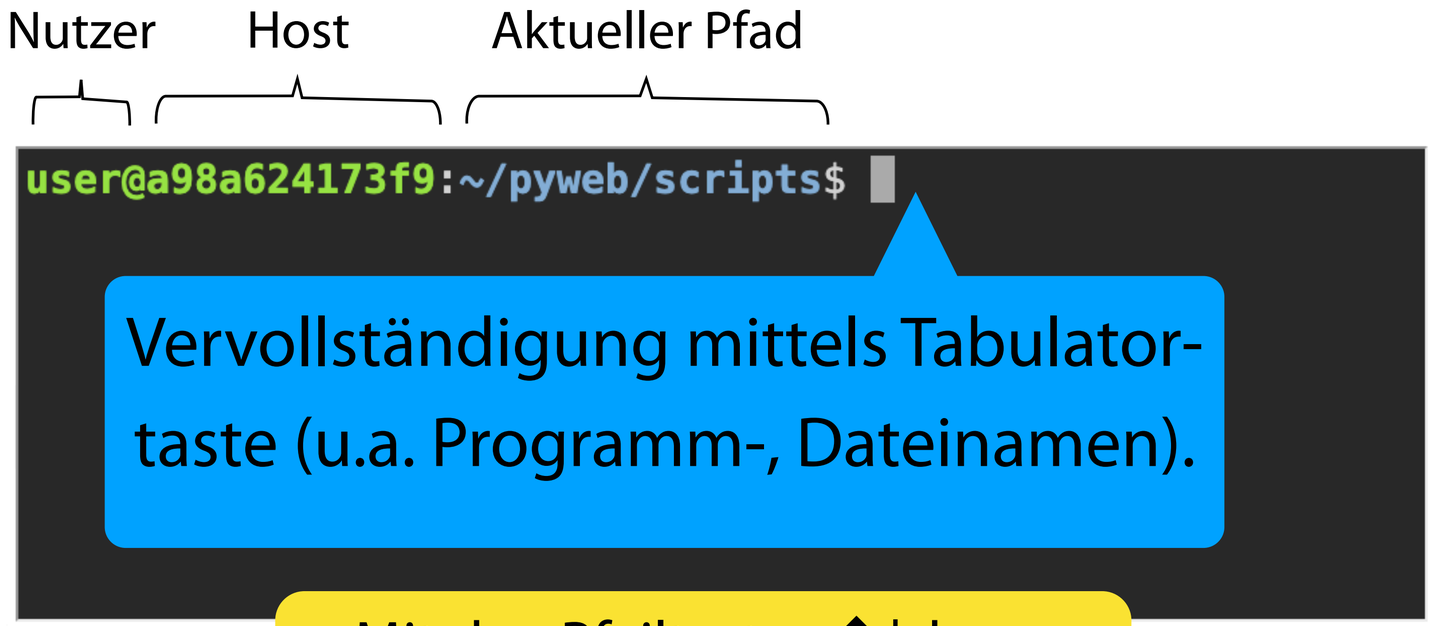


Nützliche Befehle II

Systemüberwachung	`top`	Zeigt aktuell laufende Prozesse	
	`htop`	Task-Manager (GUI-artig)	
	`free`	Zeigt Speicherbelegung an	
	`df -h`	Zeigt Festplatten und Belegung an	
Hilfe	`man`	`man ls`	Öffnet Hilfeseite
	`whereis`	`whereis python`	Zeigt Pfad zu Programm
	`whoami`	Bestimmt aktuellen Nutzernamen	
Netzwerk	`wget`	`wget http://example.com/file.zip`	
	`curl`	`curl -I http://example.com`	

Das Terminal

- Führt eine Shell aus
- Ausführungsumgebung für Commands
- Viele Designs, ähnlicher Aufbau



Symbole >&|>>&

- Um- und weiterleiten

```
echo "Hallo Welt" > ./file.txt
echo "Tschuess"   >> ./file.txt

echo "1,2,3"    | grep "2"
cat ./file.txt | python ./process.py
```

Leitet Ausgabe auf Datei um
(> überschreibt und >> fügt an).

file.txt

```
Hallo Welt
Tschuess
```

Pipe, hier: STDOUT des ersten Programms in STDIN des zweiten

- Prozesse

```
sleep 1 && echo "Hi"   &   echo "Hi2"
```

```
Hi2
Hi
```

Zwei Commands nacheinander mittels && trennen, gleichzeitig mittels &

- Glob

```
cat *.txt     file1.txt ~~file2.tex~~ a.txt
cat ?.txt     ~~file1.txt file2.tex~~ a.txt
```

Platzhalter für einen oder mehrere Zeichen in Dateinamen.

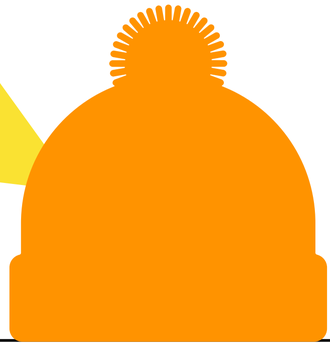
III.
Reguläre Ausdrücke

Ja, die kennt man aus TI, hier beziehen wir uns aber auf die typische Umsetzung in Programmiersprachen, *Perl Compatible Regular Expressions (PCRE)*

Reguläre Ausdrücke

- Leistungsstarke und *einfache* Art Zeichenketten zu verarbeiten
- In Python durch das Paket `re` umgesetzt.
- Nicht nur in Python unterstützt, auch z.B. in Java mit `String.matches()` oder in PHP mit `preg_match()`

„Mit dem Tool https://regexr.com/ kann man seine eigenen RegEx schnell erstellen und ausprobieren.“



Idee: RegEx

- (Teil-)Zeichenkette mit bestimmten Muster erkennen
  - „Hello“ oder „Hallo“ → `"H(e|a)llo"`
- Teile extrahieren
  - „Ratzeburger Allee 124“ → `".* Allee (\d+)"`
    - Capturing Group → `"124"`

Warum wird hier das Escape-Symbol benutzt?

Syntax I: RegEx

Anfang und Ende	`^`	Anfang der Zeichenkette
	`$`	Ende der Zeichenkette
Escape	`\`	Escape-Symbol
Zeichenmengen	`.`	Ein beliebiges Zeichen (keine neue Zeile)
	`[A-Z], [abdc], [0-9] oder [a-z0-9]`	Menge von Zeichen
	`[^A-z], [^+\-()]`	Negation einer Menge von Zeichen
	`\w, \d, \s`	Zeichen, Ziffer oder Leerzeichen (alle Arten)
	`\W, \D, \S`	Nicht Zeichen, Ziffer oder Leerzeichen (alle Arten)
Wiederholungen	`*`	Beliebig oft des Zeichens/ Gruppe zuvor
	`+`	Mindest ein Mal das Zeichen/ Gruppe zuvor
	`?`	Ein oder kein Mal das Zeichen/ Gruppe zuvor
	`{m}, {m,n}`	`m` Mal bzw. `m-n` Mal das Zeichen/ Gruppe zuvor

RegEx: Anfang, Ende, Zeichen

```
import re

re.match(r'^https?://www.example.com/?', "https://www.example.com/")
#  ✔
re.match(r'^https?://www.example.com/?', "http://www.example.com/")
#  ✔

re.match(r'^https?://www.example.com/?', "http://wwwOexampleOcom/")
#  ✔ (Aber das soll ja nicht so!)
re.match(r'^https?://www\.example\.com/?', "http://wwwOexampleOcom/")
#  ✗

re.match(r'^https?://www\.example\.com/?', "http://www.example.com/hallo")
#  ✔ (Aber das soll ja nicht so!)
re.match(r'^https?://www\.example\.com/?$', "http://www.example.com/hallo")
#  ✗
```

RegEx: Wiederholungen und Mengen

```
pattern = r'[^@]@[a-z]\.?uni-luebeck\.de'
re.match(pattern, "bender@ifis.uni-luebeck.de")
#  ✗
re.match(r'[^@]+@[a-z]+\.?uni-luebeck\.de', "bender@ifis.uni-luebeck.de")
#  ✔

re.match(r'[^@]+@[a-z]+\.?uni-luebeck\.de', "m.bender@uni-luebeck.de")
#  ✗
re.match(r'[^@]+@[a-z]*\.?uni-luebeck\.de', "m.bender@uni-luebeck.de")
#  ✔
re.match(r'\w+@\w*\.?uni-luebeck\.de', "bender@ifis.uni-luebeck.de")
#  ✔

re.match(r'\w+@\w*\.?uni-luebeck\.de', "m.bender@uni-luebeck.de")
#  ✗ (Achtung, \w beinhaltet kein Symbole!)

re.match(r'\w+@\w*\.?uni-luebeck\.de', "bender@FAKEuni-luebeck.de")
#  ✔ (Sollte aber nicht)
```

Bei E-Mail-Adressen gibt es auch einige „abenteuerliche“ Formen (RFC 3696), daher besser eine Bibliothek nutzen.

Syntax II: RegEx

Oder	`... | ...`	Passt, falls einer der beiden RegEx ... passt
Gruppe	`( ... )`	Capturing Group für RegEx ...
	`(?: ... )`	Non-Capturing Group RegEx ...
	`(?! ... )`	Passt, falls der RegEx ... nicht passt
	usw.	Es gibt noch viele weitere Gruppen
Weitere Zeichen	`\n`	Zeilenumbruch (neue Zeile)
	`\r`	Carriage return
	`\t`	Tabulator

RegEx: Gruppen

```python
import re

m = re.match(r'^H(e|a)llo$', "Hello")
print(m.groups()) # ('e',)

m = re.search(r'H(e|a)llo', "Hallo Magnus, Hello Mr. Bender,")
print(m.groups()) # ('a',)

m = re.findall(r'H(e|a)llo', "Hallo Magnus, Hello Mr. Bender,")
print(m)          # ['a', 'e']

m = re.findall(r'(H(e|a)llo)', "Hallo Magnus, Hello Mr. Bender,")
print(m)          # [('Hallo', 'a'), ('Hello', 'e')]

m = re.findall(r'H(?:e|a)llo ([^,]+)', "Hallo Magnus, Hello Mr. Bender,")
print(m)          # ['Magnus', 'Mr. Bender']

mail = re.compile(r'[^@]+@(?:[a-z]*\.)?uni-luebeck\.de')
print(mail.match("m.bender@uni-luebeck.de"))    # ✔
print(mail.match("bender@ifis.uni-luebeck.de")) # ✔
print(mail.match("bender@FAKEuni-luebeck.de"))  # ✗
```

RegEx: Gruppen und Ersetzungen

Mehrzeilige Strings können mit """ erstellt werden.

```
import re

program = """
class Zug:
  def __ini__(self, a, b, c):
    pass
  def _internal(self):
    pass
class Lokomotive():
"""

m = re.search(r'class ([a-zA-Z_][0-z]*)(?:\(\))?:', program)
print(m.group(0), m.group(1))

print(re.findall(r'class ([a-zA-Z_][0-z]*)(?:\(\))?:', program))
print(re.findall(r'class ([a-zA-Z_][0-z]*)(\(\))?:', program))

print(re.sub(
  r'def __ini__\((self[^)]*)\):',
  r'def __init__(\1):',
  program
))
```

Sucht erstes Vorkommen

Sucht alle Vorkommen

Nimmt man für den zweiten Parameter "", dann löscht man alles, auf das der RegEx passt.

Suchen und ersetzen, beim Erstzen können Gruppen eingesetzt werden.

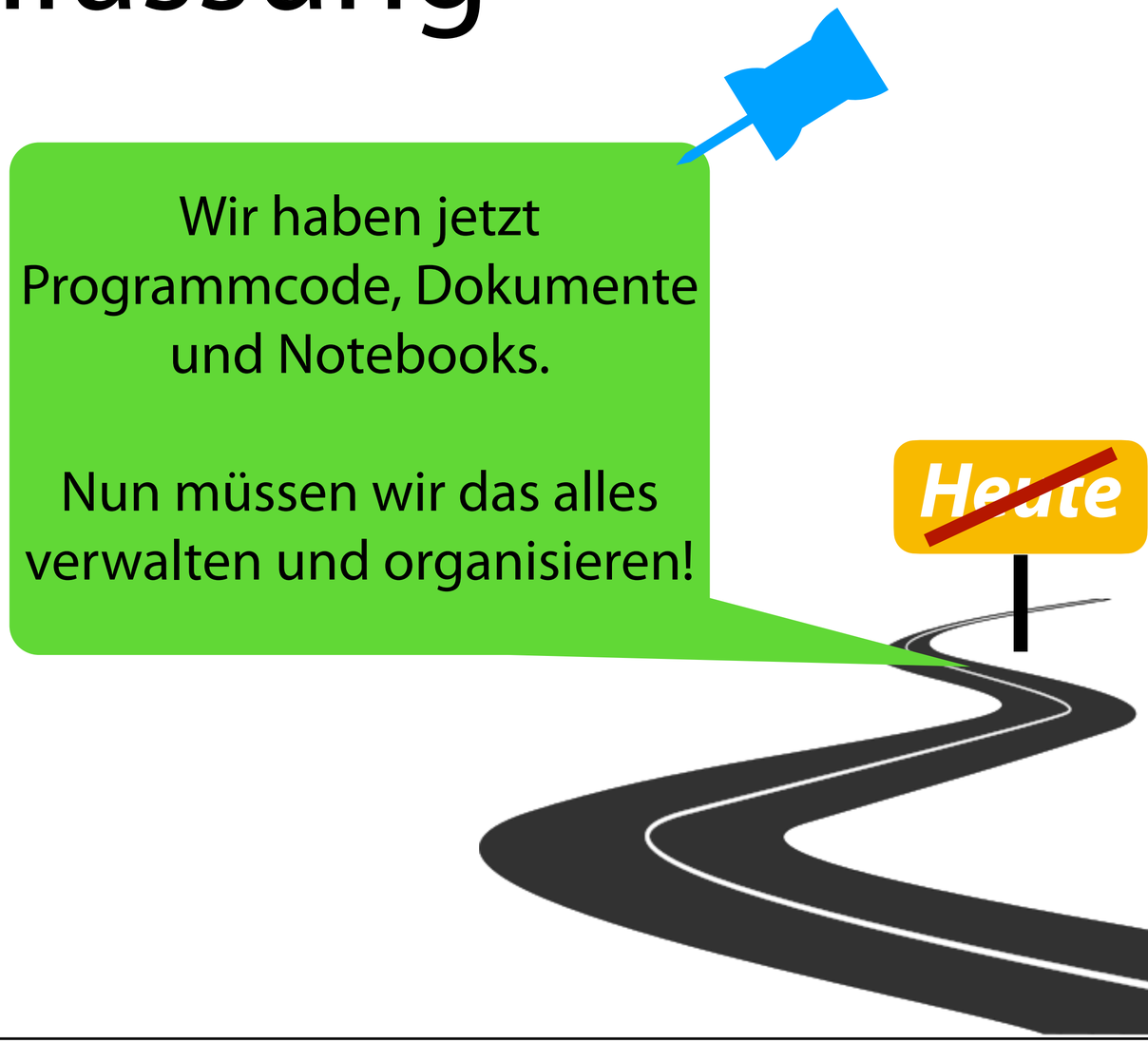
```
$> python3 name.py

  class Zug: Zug


  ['Zug', 'Lokomotive']
  [('Zug', ''), ('Lokomotive', '()')]

…
def __init__(self, a, b, c):
```

Zusammenfassung

- Lösungsmöglichkeiten Aufgabe 1

- Notebooks lokal und in der Cloud
- Grundlagen von Bash

- Reguläre Ausdrücke



Inhaltsübersicht

1. Programmiersprache Python
   a) *Einführung, Erste Schritte*
   b) *Grundlagen*
   c) *Fortgeschritten*
2. Auszeichnungssprachen
   a) *LaTeX, Markdown*
3. Benutzeroberflächen und Entwicklungsumgebungen
   a) *Jupyter Notebooks lokal und in der Cloud (Google Colab)*
4. Versionsverwaltung
   a) Git, GitHub
5. Wissenschaftliches Rechnen
   a) NumPy, SciPy
6. Datenverarbeitung und -visualisierung
   a) Pandas, matplotlib, NLTK
7. Machine Learning (scikit-learn)
   a) Grundlegende Ansätze (Datensätze, Auswertung)
   b) Einfache Verfahren (Clustering, …)
8. DeepLearning
   a) TensorFlow, PyTorch, HuggingFace Transformers