Automatisierung mit **make**

PeP et al. Toolbox Workshop

PEP ET AL. E.V.
PHYSIKSTUDIERENDE UND
EHEMALIGE PHYSIKSTUDIERENDE
DER TU DORTMUND

2015

Problem:

- → kurz vor Abgabe noch neue Korrekturen einpflegen
  - → Tippfehler korrigieren, Plots bearbeiten
- → TEX ausführen, ausdrucken
- → vergessen, Plots neu zu erstellen

Lösung: Make!

- → sieht, welche Dateien geändert wurden
- → berechnet nötige Operationen
- → führt Python-Skript aus, führt TEX aus

Motivation

- → Automatisierung verhindert Fehler
- → Dient als Dokumentation
- → Reproduzierbarkeit unverzichtbar in der Wissenschaft
- → Idealfall: Eingabe von `make` erstellt komplettes Protokoll/Paper aus Daten
- → Spart Zeit, da nur nötige Operationen ausgeführt werden

Makefile

- Datei heißt `Makefile`, keine Endung!
  - bei Windows Dateiendungen einschalten!
    (`http://support.microsoft.com/kb/865219/de`)
- besteht aus Rules:

Rule

```
target: prerequisites
    recipe
```

`target` Datei(en), die von dieser Rule erzeugt werden

`prerequisites` Dateien, von denen diese Rule abhängt

`recipe` Befehle: `prerequisites` → `target` (mit Tab eingerückt)

Einfachstes Beispiel

```
plot.pdf: plot.py data.txt
    python plot.py
```

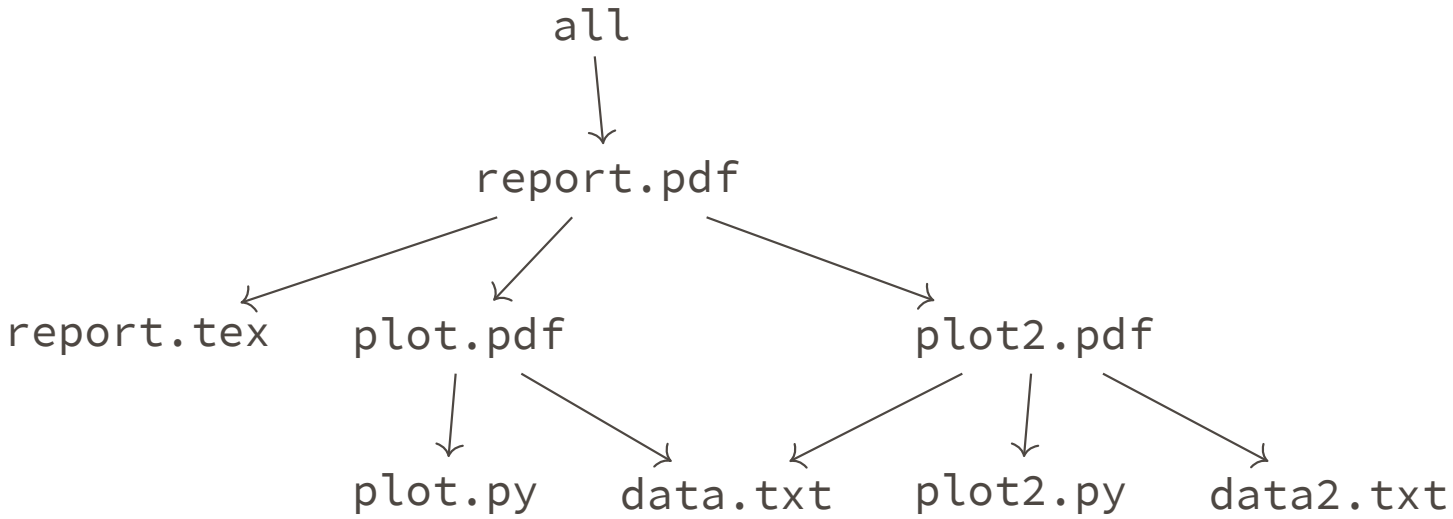
Beispiel

```
all: report.pdf  # convention

plot.pdf: plot.py data.txt
    python plot.py

report.pdf: report.tex
    lualatex report.tex

report.pdf: plot.pdf  # add prerequisite
```

`make` eingeben:

→ `all` braucht `report.pdf`
  → `report.pdf` braucht `plot.pdf`
    → `python plot.py`
  → `lualatex report.tex`

Funktionsweise



- Abhängigkeiten bilden einen DAG (directed acyclic graph)
- Dateien werden neu erstellt, falls sie nicht existieren oder älter als ihre Prerequisites sind
- Prerequisites werden zuerst erstellt
- top-down Vorgehen

Argumente für `make`

`make` *`target`*	statt des ersten in der `Makefile` genannten Targets (meist `all`) nur `target` erstellen
`make -n`	dry run: Befehle anzeigen aber nicht ausführen
`make -d`	debug: anzeigen, warum `make` sich so entschieden hat
`make -p`	Datenbank aller Abhängigkeiten ausgeben

→ Nützlich, wenn man einen Plot bearbeitet: `make plot.pdf`

make clean

Es ist eine (nützliche) Konvention, dass `make clean` alles vom `Makefile` erstellte löscht.

```
clean:
    rm plot.pdf report.pdf
```

Das Projekt sollte dann so aussehen wie vor dem ersten Ausführen von `make`.

`build`-Ordner: Projekt sauber halten

```
all: build/report.tex

build/plot.pdf: plot.py data.txt | build
    python plot.py  # savefig('build/plot.pdf')

build/report.pdf: report.tex build/plot.pdf | build
    lualatex --output-directory=build report.tex

build:
    mkdir -p build

clean:
    rm -rf build

.PHONY: all clean
```

- → `|` `build` ist ein order-only Prerequisite: Alter wird ignoriert
- → Targets, die bei `.PHONY` genannt werden, entsprechen nicht Dateien (guter Stil)

Expert

Mehrere unabhängige Auswertungen: könnte man sie parallel ausführen?
Ja! `make -j4` (4 Prozesse gleichzeitig)

```
plot1.pdf plot2.pdf: plot.py data.txt
    python plot.py
```

Problem: `make` führt `plot.py` gleichzeitig zweimal aus

Lösung: manuell synchronisieren

```
plot1.pdf: plot.py data.txt
    python plot.py

plot2.pdf: plot1.pdf
```

Wenn man `plot2.pdf` aber nicht `plot1.pdf` löscht, kann `make` nicht mehr `plot2.pdf` erstellen.