

Objektorientierung und Funktionale Programmierung

Einführung

Kristof Van Laerhoven
kvl@eti.uni-siegen.de

Michael Moeller
Michael.moeller@uni-siegen.de

Andreas Hoffmann
Andreas.Hoffmann@uni-siegen.de
uni-siegen.de



Funktionale Programmierung

Wir sollten unsere Programme (soweit es geht) durch *pure functions* aufbauen, d.h. durch Funktionen die

- 1) *nur von ihren Eingabeargumenten abhängen* und
- 2) *keine Nebeneffekte haben! Ihr einziger Effekt ist die Rückgabe von Werten.*

Wir üben nun das Schreiben von funktionalem Code, erstmal von pure functions, am Beispiel unseres Goofspiels

- Nebeneffekte vermeiden, z.B. keine globale Variablen in Funktionen ändern, alle Abhängigkeiten offenlegen

Also kurz: **Pure functions verwenden**

- Immutable Variablen verwenden
- Erstmal nur alles für den Spielverlauf wichtige sammeln – die Dokumentation (mit print Befehlen) abkoppeln

(Logik des Programms und Dokumentation trennen!)

- Alles, was als „state“ des Programms wichtig ist, in einer „state“ Variablen sammeln

Goofspiel funktionaler machen

Nun können wir noch den einzigen wirklich notwendigen Nebeneffekt, einen print Befehl, vom Rest des Programms trennen, indem wir uns alle states merken! Hierzu generieren wir ein tuple mit allen states, an das wir in jeder Runde einen Eintrag anhängen

```
n = 13 # Anzahl der Karten
anfangsstate = {"preiskarten" : h.kartenspiel_erzeugen(n), \
                "karten_spieler1" : h.kartenspiel_erzeugen(n), \
                "karten_spieler2" : h.kartenspiel_erzeugen(n), \
                "punkte_spieler1" : 0, \
                "punkte_spieler2" : 0, \
                "strategieSpieler1" : h.spiele_die_gleiche_karte_wie_preiskarte, \
                "strategieSpieler2" : h.spiele_zufällige_karte, \
                "karte1" : None, \
                "karte2" : None, \
                "preiskarte" : None \
               }

states = (anfangsstate,) # Tuple mit dem einen Eintrag "anfangsstate". Dieses soll
# Das Spiel selbst:
while (len(states[-1]["preiskarten"])>0): # Wenn Liste der Preiskarten leer, dann
    states = states + (h.spielrunde(states[-1]),)
```

Der einzige Nebeneffekt steckt dann in

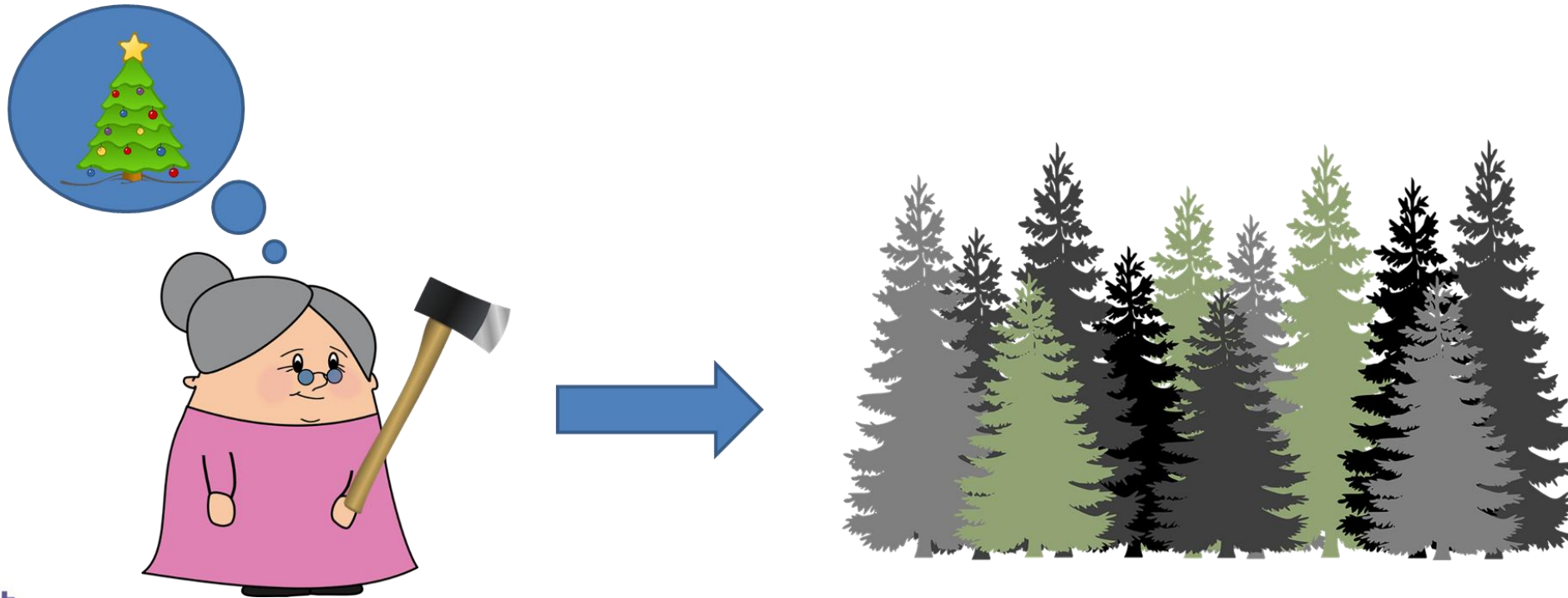


```
# Dokumentation des Spielverlaufs
if alles_kommentieren:
    print(h.spieldokumentation(states, h.dokumentation_einer_spielrunde, h.auswertung))
```

Um unser Goofspiel “noch funktionaler“ zu machen, benötigen wir zunächst ein weiteres Konzept: Die **Rekursion**

Verrückte Eigenschaft: Man kann in Python (und fast jeder anderen Sprache) eine Funktion schreiben, die sich selbst aufruft

Ich habe auf youtube gesucht, wie man Rekursion erklären kann. Hier das Beispiel, welches <https://www.youtube.com/watch?v=weTpjhDnLnc> folgt.





Wie fällt man einen Baum der Dicke a ?

Wiederhole „HACK“ (und subtrahiere 1 von der Dicke) bis die Dicke auf 0 ist. Rufe: „Baum fällt!“

Wichtig ist, dass bei der Rekursion stets auf eine geeignete Abbruchbedingung geachtet wird damit die Rekursion immer terminiert!

Was geschieht hier?

```
def hacke_baum(dicke, level=0):  
    if dicke==0:  
        print("\t"*level+"Baum fällt!")  
    else:  
        print("\t"*level+"HACK!")  
        hacke_baum(dicke-1, level+1)  
        print("\t"*level+"Man kann sogar hier  
schreiben! Level", level)  
  
hacke_baum(4)
```

```
HACK!  
    HACK!  
        HACK!  
            HACK!  
                Baum fällt!  
                Man kann sogar hier schreiben! Level 3  
            Man kann sogar hier schreiben! Level 2  
        Man kann sogar hier schreiben! Level 1  
    Man kann sogar hier schreiben! Level 0
```

Es bildet sich ein **stack (Stapel)** von Funktionsaufrufen, der nacheinander abgearbeitet wird!

Rekursion ist oft eine Alternative zu Schleifen (**for** oder **while**).

Vorteile:

- Einige Programme können mit Rekursion einfacher, eleganter und mit weniger Code gelöst werden.
- Leichter und kürzerer Code vermeidet Fehler und ist einfacher zu pflegen
- Rekursion kann oft die Mutation von Variablen verhindern und ist daher für die funktionale Programmierung sehr interessant!

Nachteile:

- Oft weniger Laufzeit effizient (Funktionsaufrufe weniger effizient; stack an Aufrufen)
- Oft speicherhungriger, weil die Daten aller rekursiver Aufrufe auf dem Stack gespeichert werden müssen.

Beispiel: Berechne $n!$ mittels einer for-Schleife und rekursiv!

```
def fakultät_for(n):  
    val = 1  
    for i in range(1,n+1):  
        val *= i  
    return val  
  
def fakultät_rekur(n):  
    if n<=1:  
        return 1  
    else:  
        return n*fakultät_rekur(n-1)
```

Abstrakte Idee:

```
# Das eigentliche Spiel
abbruchkriterium = lambda states: not states[-1]["preiskarten"]
states = help.spiel_rekursiv((state,), abbruchkriterium, help.spielrunde)
```

```
def spiel_rekursiv(states, abbruchkriterium, spielrunde):
    if abbruchkriterium(states):
        return states
    else:
        return spiel_rekursiv(states + (spielrunde(states[-1]), ))
```

Wenn wir dann noch eine Dokumentation des gesamten Spielverlaufs aus den States als pure Function schreiben besteht das Hauptprogramm praktisch nur noch aus einem print-Befehl

Datentypen: Dictionary + Tuple, z.B. state des Spiels als Dictionary, Tuple von states um das ganze Spiel zu dokumentieren

Das Zeichen `\` um im Editor zur besseren Lesbarkeit einen Zeilenumbruch machen zu können

Die Addition von Tupeln; führt zu einem neuen Tupel, dass beide konkateniert

Mit `random.Random(seed)` einen Zufallsgenerator erzeugen, `setstate()`, `getstate()`

Tab in einem String `\t`

Konzept: Rekursion