

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



Unser Goofspiel ist nun nur noch

```
## Zustand "state" des Programms
state = {"zufallsgenerator": random.Random(0), \
        "spieler1_kartenspiel" : help.kartenspiel(N), \
        "spieler2_kartenspiel" : help.kartenspiel(N), \
        "spieler1_karte" : None, \
        "spieler2_karte" : None, \
        "preiskarten" : help.kartenspiel(N), \
        "aktuelle_preiskarte" : None, \
        "spieler1_strategie" : help.spiele_zufällige_karte, \
        "spieler2_strategie" : help.spiele_den_wert_der_preiskarte, \
        "spieler1_punkte" : 0, \
        "spieler2_punkte" : 0}

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

Aufgabe: Umgekehrte Zeichenkette mit Rekursion

Schreibe eine **rekursive Funktion** `umkehren(text)`, die eine Zeichenkette **rückwärts** ausgibt. Verwende **keine Schleifen**, sondern **ausschließlich Rekursion**.

Beispiel:

python

 Kopieren

 Bearbeiten

```
umkehren("hallo") → "ollah"
```

```
umkehren("abc") → "cba"
```

Aufgabe: Summe aller Zahlen in einer verschachtelten Liste

Schreibe eine rekursive Funktion `verschachtelte_summe(liste)`, die alle Zahlen in einer Liste aufsummiert, auch wenn die Liste verschachtelte Unterlisten enthält.

Beispiel:

python

 Kopieren

 Bearbeiten

```
verschachtelte_summe([1, 2, [3, 4], [5, [6]], 7])
```

→ 28

! Hinweise:

- Die Liste kann beliebig tief verschachtelt sein.
- Die Funktion soll **rekursiv durch alle Unterlisten** gehen und Zahlen aufsummieren.
- Du kannst `isinstance(x, list)` verwenden, um zu prüfen, ob ein Element eine Liste ist.

Dinge noch „funktionaler“ machen

Im Goofspiel folgt nur noch die Dokumentation. Hier gibt es immernoch eine **for**-Schleife, die etwas an **doku** anhängt.

```
#### Dokumentation
doku = ""
for state in states[1:]:
    doku += help.rundendokumentation(state) + "\n"
doku += help.auswertung_am_ende(state["spieler1_punkte"], state["spieler2_punkte"])

# Nebeneffekt: Drucke die gesamte Spieldokumentation
print(doku)
```

Können wir dies schöner / funktionaler schreiben?

Hilfreich: Dinge aus einer Liste oder einem Tupel (oder einem anderen iterierbaren Objekt)

- einzeln weiterverarbeiten,
- auf ein Element reduzieren,
- oder auswählen.

→
map
reduce
filter

Den Befehl **map** haben wir früher bereits einmal kennengelernt. Er nimmt eine Funktion und ein iterierbares Objekt (z.B. Liste oder Tupel) entgegen und wendet die Funktion auf jedes Element des iterierbaren Objektes an. Genauer, bekommen wir einen Iterator zurück über welchen wir iterieren können oder Methoden wie **__next__()** aufrufen.

Beispiel: Multipliziere jede Zahl in einer Liste mit einer zufälligen ganzen Zahl zwischen 0 und 9.

map kann auch eine Funktion, die mehrere Eingabeargumente erwartet, als erstes Argument bekommen, wenn danach so viele (gleichlange) iterierbare Objekte folgen, wie die Funktion Eingabeargumente hat. In diesem Fall wird

```
“map(f, liste1, liste2, ..., liste_m)
= [f(liste1[0], liste2[0], liste_m[0]), ..., f(liste1[-1], liste2[-1], liste_m[-1])]“
```

gebildet.

Beispiel: Definiere eine Funktion **vektoraddition**, welche map verwendet um zwei Listen elementweise zu addieren.

Der Befehl **reduce** nimmt auch eine Funktion und iterierbares Objekt (z.B. Liste oder Tupel) entgegen. Die Funktion muss zwei Elemente in dem iterierbaren Objekt auf eines abbilden. Reduziert nun durch sukzessives Anwenden der Funktion das gesamte iterierbare Objekt auf ein Element:

“reduce(f, [x1,x2,x3, ..., xn]) = f(...f(f(f(x1,x2),x3), ..., xn))“

Beispiel: Durchschnitt von Zahlen bilden.

Beispiel: Berechne das „Birthday Paradox“ (aus der ersten Vorlesung) in Python.

Beispiel: Eine Liste von Strings zu einem einzigen String (mit Absätzen zwischen den Einträgen) machen.

Beispiel: Besonders kompakte Version von **spieldokumentation**

```
def spieldokumentation (states, rundendokumentation, schlussdokumentation):  
    return reduce(lambda x1, x2: x1+'\n'+x2, map(rundendokumentation, states[1:])) \  
        + '\n' +schlussdokumentation(states[-1])
```

Beachte: Der Funktion **reduce** kann zusätzlich ein Startwert gegeben werden.

filter nimmt eine Funktion und ein iterierbares Objekt (z.B. Liste oder Tupel) entgegen. Die Funktion nimmt ein Element unseres iterierbaren Objektes als Eingabe und evaluiert zu einem boolschen Wert (also **True** oder **False**). Die Rückgabe von Filter ist ein Iterator, der nur diejenigen Elemente enthält, auf denen die Funktion zu **True** evaluiert wurde.

Beispiel: Finde alle Worte mit mehr als vier Buchstaben in einem gegebenen Satz.

Beispiel: Gegeben zwei Listen, finde die Summe derjenigen Zahlen in der ersten Liste, die vom gleichen Element der zweiten Liste ohne Rest geteilt werden.

Beispiel: Schreibe folgende Funktion aus dem Goofspiel unter Verwendung von „filter“ um:

```
def kartenspiel_ohne_karte(karte : int, kartenspiel : tuple)->tuple:  
    neues_kartenspiel = list(kartenspiel)  
    neues_kartenspiel.remove(karte)  
    return tuple(neues_kartenspiel)
```

Befehle **map**, **filter**, **reduce**

Ein paar abschließende generelle Dinge

Um bessere Programmierer*innen zu werden: Üben, üben, üben!

Schreiben sie Code, den sie und anderen nach Monaten noch verstehen! Z.B. mit Kommentaren und sinnvoller Benamung. Lernen sie (selbstständig) Error/Exception Handling. Lernen sie in ihrer IDE Debugging!

Machen sie sich mit einem Versionierungstool wie z.B. Git vertraut!

Schauen sie sich um welche Module/Bibliotheken/Repositories es in ihrem Interessensgebiet in Python gibt.

Lernen sie den Code anderer Leute zu lesen, zu verstehen und zu editieren. Merken sie sich interessante Ansätze/Stile/Befehle.

Sobald sie anfangen mehrere zu installierenden Module zu verwenden, machen sie sich mit Environments vertraut!

Falls sie irgendetwas Richtung Datenanalyse (oder sonstigen mathematischen Berechnung) mit Python machen, sollten sie unbedingt numpy und matplotlib kennenlernen.

Überlegen sie sich die Art der von ihnen geschriebenen Programme – macht es Sinn die Abhängigkeit von states (Zuständen) und eine implizite Manipulation von Eingaben zu vermeiden?

Danke für Ihre Aufmerksamkeit!

Welche Fragen sind offen geblieben?

Es gibt viele wertvolle Tools für kompakte (funktionale) Programmierung, über die wir nicht sprechen konnten.

Ein Anfang ist das Lesen von <https://docs.python.org/3/howto/functional.html>

Einen letzten Aspekt in dieser Vorlesung zum Thema funktionale Programmierung ist das sogenannte **Currying**

Die Idee ist es abstrakte Funktionen schreiben zu können, die von vielen Parametern abhängen und diese dann durch das *Binden* bestimmter Parameter an fixe Werte spezialisieren zu können.

Eine Beispiel: Aus einer Funktion, die die Summe zweier Zahlen bildet, durch fixieren eines Argumentes auf den Wert 2 eine neue Funktion **zwei_weiter** erschaffen.

```
def summe(x,y):  
    return x+y  
  
zwei_weiter = lambda y : summe(2,y)  
  
print(zwei_weiter(0))
```

Abstrakt mathematisch kann man dies so verstehen, dass eine Funktion mit zwei Eingabeargumenten

$$f : A_1 \times A_2 \rightarrow B$$

auch als eine Funktion verstanden werden kann, die einen Parameter auf eine Funktion in einem weiteren Parameter abbildet

$$f_{\text{curry}} : A_1 \rightarrow (A_2 \rightarrow B)$$

Es gilt also

$$f_{\text{curry}}(a_1)(a_2) = f(a_1, a_2)$$

aber man kann auch $f_{\text{curry}}(a_1)$ alleine auswerten (und bekommt dann eine Funktion die noch einen Parameter als Eingabe erwartet).

Dieses Konzept lässt sich auf beliebig viele Parameter erweitern, also für Funktionen

$$f : A_1 \times A_2 \times \dots \times A_n \rightarrow B$$

Einige funktionale Programmiersprachen (z.B. Haskell) sind stark auf die Verwendung solcher Konzepte ausgelegt. (Kein Wunder, der Name kommt vom Mathematiker Haskell Brooks Curry, 1900 – 1982).

Mit einem Decorator kann man eine Funktion erweitern oder verändern, indem man sie in eine Funktion höherer Ordnung hüllt (wrapt).

Z.B. für jede Funktion, die einen String zurückgibt, könnte folgendes interessant sein

```
✓ def grossbuchstaben(funktion):  
  ✓ | def wrapper(*args, **kwargs):  
    |     ergebnis = funktion(*args, **kwargs)  
    |     return ergebnis.upper()  
  | return wrapper
```

In Python genügt es nun, vor die Definition eine Funktion mit einem @-Zeichen den obigen Wrapper zu schreiben

```
@grossbuchstaben  
def begruessung(name):  
    | return f"Hallo, {name}!"
```

Dies führt zu einer Ausgabe in Großbuchstaben!

Damit Metadaten (Funktionsname, docstring, etc.) trotz Dekorator noch von der ursprünglichen Funktion erhalten bleiben, schreibt man noch ein

```
@functools.wraps(funktion)
```

mit in die Definition der umhüllenden Funktion („grossbuchstaben“ in unserem Beispiel). Hierfür muss man `functools` importieren.

Beispiel: Schreibe einen Dekorator `logge_aufruf`, der vor jedem Funktionsaufruf den Funktionsnamen und die übergebenen Argumente auf der Konsole ausgibt

```
def logge_aufruf(funktion):  
    @functools.wraps(funktion)  
    def wrapper(*args, **kwargs):  
        print(f"[LOG] {funktion.__name__} aufgerufen mit {args}, {kwargs}")  
        return funktion(*args, **kwargs)  
    return wrapper
```