

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



Letzte Vorlesung:

- Wie definiert man Funktionen?
- Dann: Funktionen können in Variablen gespeichert werden
- Dann: Funktionen können Funktionen als Eingabeargumente haben (*Funktionen höherer Ordnung*)

Schreiben sie eine abstrakte Funktion `auf_jedes_element(funktion, liste)` die eine Funktion und eine Liste entgegennimmt, und auf jedes Element der `liste` die Funktion `funktion` anwendet.

Testen sie ihre Funktion an zwei Beispielen:

- Quadrieren sie alle Zahle von 0 bis 9
- Fügen sie an eine Liste von Strings einen Absatz zu jedem String hinzu.

Hierfür sehr praktisch: Der Befehl

`lambda x1, x2, ... : aus Eingaben bestimmter Rückgabewert`

Erlaubt es kurze Funktionen schnell (in einer Zeile) zu schreiben ohne dass man ein `def` benötigt.

Vervollständigen Sie folgendes Lernspiel für Kinder der dritten Klasse: Es werden zwei zufällige Zahlen von 0 bis 9 und eine zufällige Rechenoperation aus {+,-,*} gezogen. Das Ergebnis der Rechenoperation mit der Zahl wird zurückgegeben. Die Kinder werden gefragt, welche Operation verwendet wurde. Verwenden Sie die lambda Notation um Funktionen zu definieren.

```
Import (Lücke 0)
addition = (Lücke 1)          # definiere eine Funktion, die zwei Zahlen addiert
multiplikation = (Lücke 2)    # definiere eine Funktion, die zwei Zahlen multipliziert
subtraktion = (Lücke 3)       # definiere eine Funktion, die zwei Zahlen subtrahiert
zufallszahl = (Lücke 4)       # definiere eine Funktion, die einen zufällige natürliche Zahl
                               # zwischen 0 und 9 liefert

liste = [addition, multiplikatio, subtraktion]
x = zufallszahl()
y = zufallszahl()

op = (Lücke 5) # Wähle ein zufälliges Element aus „liste“

print(„Das Ergebnis von {} und {} ist {}. Welche Operation wurde durchgeführt?“.format(x,y,(Lücke 6)))
```

Zurück zum Goofspiel und zum Verständnis einiger Programmierkonzepte.

Interessante Beobachtung: Unsere aktuelle Funktion „mischen“ hat ein Kartenspiel als Eingabe, hat aber keine Rückgabe. Trotzdem wird das Kartenspiel offensichtlich durch die Funktion geändert!

Dies liegt daran, dass „kartenspiel“ vom Typ „list“ ist und es sich dabei um eine **veränderbare (mutable)** Variable handelt. Ähnliche Konstruktionen für **unveränderbare (immutable)** Variable funktionieren nicht:

```
def teste_aenderung(a,b):  
    a = 5  
    return a+b  
  
x = 1  
y = 1  
print(teste_aenderung(x,y))  
print(x)
```

Unveränderbar (immutable) in Python:

int, float, bool, string, unicode, tuple

Veränderbar (mutable) in Python:

list, dict, set, selbstgeschriebene Klassen

Das Verwenden (von allen Variablen) und Manipulieren (von mutable Variablen) funktioniert auch, wenn diese definiert, aber nicht explizit das Argument einer Funktion sind

```
a = 4

def meine_funktion(x):
    return x+a

print(meine_funktion(2))
```

Hier wird eine Funktion definiert die weder Eingabe noch Ausgabe hat, aber den ersten Eintrag

in ,liste' manipuliert

```
liste = [1,2,3]

def meine_manipulative_funktion():
    liste[0] = 5

meine_manipulative_funktion()
print(liste)
```

Wir nennen überall bekannte Variablen (**a** und **liste**) in den vorherigen Beispielen **globale Variablen**. Dies geschieht in Abgrenzung zu **lokalen Variablen**, die nur innerhalb einer Funktion bekannt sind.

```
a = 1
def meine_funktion():
    b = 4
    return a+b
meine_funktion()
print(b)  Dies erzeugt einen Fehler – b ist nur lokal bekannt
```

```
NameError: name 'b' is not defined
```

```
a = 1
def meine_funktion():
    b = a + 1
    print(b)
```

Funktioniert! **a** ist globale Variable, **b** ist lokal und wird auch lokal mittels **print** gedruckt.

```
a = 1
def meine_funktion():
    a = 5

meine_funktion()
print(a)
```

Funktioniert und gibt **1** auf der Kommandozeile aus! **a** ist sowohl globale Variable als auch lokale Variable mit einem anderen Wert, da **a** eine Zahl und damit unveränderbar (immutable) ist. Innerhalb von **meine_funktion()** hat **a** den Wert 5. Außerhalb der Funktion hat und behält **a** den Wert 1.

```
a = [1]
def meine_funktion():
    a = [5]

meine_funktion()
print(a)
```

Funktioniert und gibt **[1]** auf der Kommandozeile aus! **a** ist sowohl globale Variable als auch lokale Variable mit einem anderen Wert. Innerhalb von **meine_funktion()** hat **a** den Wert **[5]**. Außerhalb der Funktion hat und behält **a** den Wert **[1]**.

```
a = [1]
def meine_funktion():
    a[0] = 5

meine_funktion()
print(a)
```

Funktioniert und gibt **[5]** auf der Kommandozeile aus! **a** ist eine globale veränderbare Variable. Veränderungen von **a** sind innerhalb von Funktionen möglich.

```
a = 1
def meine_funktion():
    a = a + 1

meine_funktion()
```

Funktioniert nicht: Der Interpreter versteht a als lokale Variable, die aber noch nicht initialisiert ist

```
a = 1
def meine_funktion(a):
    a = a + 1

meine_funktion(a)
print(a)
```

Funktioniert und gibt 1 aus: Da a unveränderbar ist, gelten die Änderungen innerhalb von **meine_funktion** nur lokal.

Mit dem Schlüsselwort **global** ist es möglich den Geltungsbereich einer Variablen zu erweitern. Folgender Code funktioniert und gibt 2 aus

```
a = 1
def meine_funktion():
    global a
    a = a + 1

meine_funktion()
print(a)
```

```
def meine_fun():
    global a
    a = 2

meine_fun()
print(a)
```

Wenn sie jetzt verwirrt sind, dann ist **Funktionale Programmierung** nun gut motiviert!

Idee: 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.**

1) wird auch oft als das Vermeiden der Abhängigkeit von sogenannten *states (Zuständen)* beschrieben!

- Pure functions hängen von nichts anderem ab, als von den Eingabeargumenten.
- Sie greifen weder auf globale Variablen noch auf irgendwelche anderen externen Daten zu.
- Eine Funktion die nur von ihren Eingabeargumenten abhängt, wird auch *referentially transparent* genannt.

Idee: Funktionen im Programm sollten sich eher so verhalten wie mathematische Funktionen.

Wenn jemand in der Mathematik

$$y = f(x)$$

schreibt, dann geht man auch davon aus, dass f von keinen anderen Dingen als x abhängt und dass x anschließend keinen anderen Wert hat.

Eine ganz formale Definition von Funktionaler Programmierung gibt es nicht. Das Vermeiden von Nebeneffekten und Abhängigkeiten von states sind die Grundpfeiler. Darüber hinaus gibt es viele Werkzeuge, die oft mit Funktionaler Programmierung in Verbindung gebracht werden, hierfür aber nicht zwingend notwendig sind.

Interessante Vorträge zum Schauen:

- Richard Feldman, The Essence of Functional Programming, Functional Conf 2022
<https://www.youtube.com/watch?v=l0ruvPCQh9I>
- Richard Feldman, Functional Programming for Pragmatists, GOTO 2021
<https://www.youtube.com/watch?v=3n17wHe5wEw>
- Russ Olsen, Functional Programming in 40 Minutes, GOTO 2018,
<https://www.youtube.com/watch?v=0if71HOyVjY>
- Rafal Dittwald, Solving Problems the Clojure Way
https://www.youtube.com/watch?v=vK1DazRK_a0

(Hierher habe ich die Goofspiel Idee – Teile der weiteren Vorlesung orientieren sich daran)

Um funktionale Programmierung besonders gut umsetzen zu können, sind einige Dinge charakteristisch, z.B.

- *Pure Functions* (wie besprochen)
- *Funktionen höherer Ordnung* (= Funktionen die Funktionen als Eingabe bekommen und/oder Funktionen als Rückgabewert haben können)
- *Immutable (unveränderbare) Datenstrukturen* (da diese keine Gefahr haben an unterschiedlichen Stellen geändert zu werden)
- *Anonyme Funktionen* (= die Möglichkeit kurze Funktionen einfach in einer Zeile zu erstellen)
- *Rekursion* (Funktionen, die sich selbst aufrufen; noch zu besprechen)
- *Funktionen auf Iteratoren* (noch zu besprechen, aber grundsätzlich bei Befehlen wie `for element in liste` schon gesehen. Hierauf können wir besonders gut higher order functions anwenden. Insbesondere `map`, `filter`, und `reduce`.)
- *Currying* (die Möglichkeit eine von mehreren Parametern abhängige Funktion als Verkettung von Funktionen zu schreiben, die nur von einem Parameter abhängen und eine Funktion zurückgeben; noch zu besprechen)

1. Einfacheres Finden von Fehlern!

```
y = meine_pure_function(x)
```

Wenn y nicht das gewünschte Ergebnis beinhaltet, liegt der Fehler in der Funktion `meine_pure_function` (vorausgesetzt ihre Eingabe `x` stimmt)

Gibt es Funktionen mit Nebeneffekten, so kann (ab Erstellung einer veränderbaren Variable) jede(!) nachfolgende Funktion ein falsches Ergebnis verursachen!

Gibt es Funktionen, die nicht nur von ihren Eingabeargumenten abhängen, muss beim Zurückverfolgen von falschen Ergebnissen nicht nur das Eingabeargument, sondern jede(!) globale Variable mit überprüft werden.

Mal ein paar Beispiele, dass dies schon bei wenigen Zeilen Code Konzentration erfordert:
<https://edu.onlinetd.de/>. Für viele Tausend Zeilen kann dies äußerst schwierig werden.

Ein kleines Quiz zu dem gelernten.

2. Wiederverwenden von Funktionen

```
y = meine_pure_function(x)
```

Pure functions können gefahrlos wiederverwendet werden! Wenn Funktionen von globalen Variablen oder anderen äußeren Umständen abhängen, müssen diese ggf. zum Anwendungsfall passen.

3. Parallelisierung

Pure functions können immer gefahrlos parallelisiert werden! Die beiden Zeilen

```
y = f(x)  
z = f(x)
```

können gleichzeitig ausgeführt werden! Wenn Funktionen Nebeneffekte haben und von globalen Variablen oder anderen äußeren Umständen abhängen, kann dies zu unkontrolliertem Verhalten führen.

4. Teilweise kompakterer/kürzerer Code

5. Für einige Probleme hohe Effizienz durch *Memoisation* (Zwischenspeichern von Rückgabewerten)



```
a = 0  
def f(x):  
    global a  
    a = a + 1  
    return x+a  
  
x = 5  
  
y = f(x)  
z = f(x)
```

Lesen und Schreiben in einer Datenbank kann ein typisches komplexeres Problem sein

6. Verzögerte Auswertung (Lazy Evaluations)

Auswertungen müssen erst dann gemacht werden, wenn sie auch gebraucht werden

```
1 import math
2 import time
3
4 def wurzel(zahl):
5     return math.sqrt(zahl)
```

```
zahlen = list(range(2,10**7))
startzeit = time.time()
for i,zahl in enumerate(zahlen):
    zahlen[i] = wurzel(zahl)
print(zahlen[0])
endzeit = time.time()

print('Dies dauerte {:.2f}
Sekunden'.format(endzeit-startzeit))
```

```
1.4142135623730951
Dies dauerte 4.32 Sekunden
```

```
zahlen = list(range(2,10**7))
startzeit = time.time()
zahlen = map(wurzel,zahlen)
print(zahlen.__next__())
endzeit = time.time()

print('Dies dauerte {:.2f}
Sekunden'.format(endzeit-startzeit))
```

```
1.4142135623730951
Dies dauerte 0.00 Sekunden
```

Was haben wir im Beispiel neu gelernt?

Das Modul `time` zusammen mit dem Befehl `time()` der es ermöglicht, durch Differenzen die Laufzeit von Programmen zu messen.

Die Funktion höherer Ordnung `map`, die eine Funktion sowie ein iterierbares Objekt entgegennimmt, und ein map Objekt zurückgibt, welches selbst iterierbar ist und durch Anwenden der gegebenen Funktion auf jedes einzelne Element des iterierbaren Objektes entsteht.

Iterierbare Objekte sind uns bislang hauptsächlich im Beispiel der Liste and der Range bekannt, aber auch Dictionaries und Tupel sind uns (bei Funktionen mit variabler Anzahl an Argumenten) schon begegnet.

Iterierbare Objekte erlauben die Schreibweise `for element in obj` in Python. Außerdem haben sie alle die Methode `__next__()` sowie einen internen ‚Zählerstand‘. Mit `__next__()` bekommt man das nächste Element als Rückgabe und erhöht den internen Zähler, sodass bei einem weiteren `__next__()` wieder das nächste Element zurückgegeben wird.

```
tupel = (1,2,3)
liste = [1,2,3]
ran = range(1,4,1)
menge = {1, 2, 3}
dic = {'eins':1, 'zwei':2, 'drei':3}

for element in ran:
    print(element)
```

`lambda x1, x2, ... : aus Eingaben bestimmter Rückgabewert`

Variablen global machen `global`

`copy()`

Viele Konzepte: Mutable, immutable, pure functions, funktionale Programmierung, Vorteile hiervon

`import time, time()`

`import math, math.sqrt()`

Auf jedes Element eine Funktion anwenden: `map(funktion, iterierbares Objekt)`

Das nächste Element in einem iterierbaren Objekt `__next__()`

Kurz gesehen: Tupel, Menge, Dictionary