FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Konzepte moderner Programmiersprachen (KmPS)

(Wahlfach, Modulnummer 55685)

Wintersemester 2025/2026

Prof. Dr. Andreas Claßen
(zusammen mit Prof. Dr. Heinrich Faßbender)
Fachbereich 5 Elektrotechnik und Informationstechnik
FH Aachen

Mutability & Immutable Data

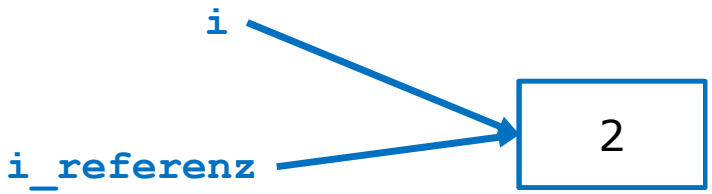
Mutability

… Wertänderung, wobei Objektidentität und Verweise auf den Wert unverändert.
"In-place Wertänderung"

Kritisch bei "shared data": Mutation ohne sichtbare ändernde Operationen.

```
int i = 2;
int& i_referenz = i;

i_referenz = 10;
```



Wert von `i` „mutiert", obwohl auf `i` keine ändernde Operation ausgeführt wurde!

Immutable Data

… darf nicht geändert werden, sobald Wert initial "erzeugt".

Datenwert nach dem Anlegen also read-only.

Unterschied zur "Konstante":
Immutable Data: initialer Wert ggfs. erst durch Berechnungen zur Laufzeit.

Immutable Data und Variablenbindung

… in Programmiersprachen unterschiedlich interpretiert.

Ist nur der Datenwert immutable (d.h. nicht veränderbar) … — *immutable data*
… oder auch die Bindung der Variable an den Wert? — *immutable variable/binding (inkl. immutable data)*

Python (Tupel sind dort immutable data):

```
t = (1, 2, 3)
t[0] = 88     // error: 'tuple' object does not support item assignment
t = (4, 5, 6) // erlaubt: Bindung an einen anderen Wert => Änderung
```

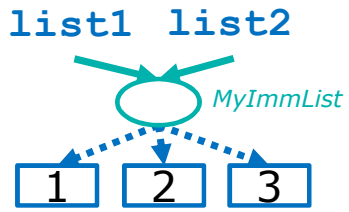
Rust (Variablendefinition immutable, falls nicht als `mut` gekennzeichnet):

```
let iv1 = vec![1,2,3];
iv1 = vec![4,5,6];   // error: cannot assign twice to immutable variable `v1`
iv1[0] = 88;         // error: cannot borrow `v1` as mutable, as it is not declared as mutable
```

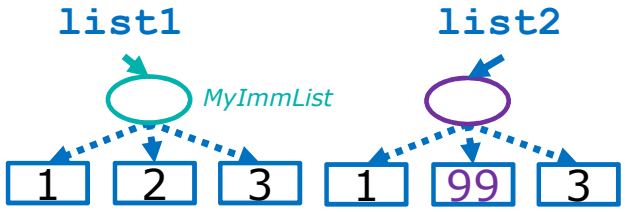
Immutable Data

… ändernde Operationen entweder nicht erlaubt oder müssen Kopie des Werts angelegen (Originalwert bleibt unverändert; Kopie ist neuer, unabhängiger Wert …).
D.h. bei „geteilten Verweisen" ein Verweis nachher auf neues, nicht geteiltes Objekt.

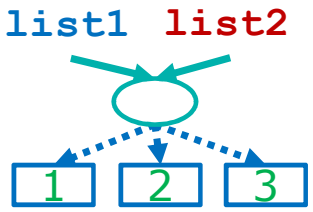
```
list1 = MyImmutableList([1,2,3])
list2 = list1
print(list1, id(list1), list2, id(list2))
# => [1, 2, 3] 50250384 [1, 2, 3] 50250384
```



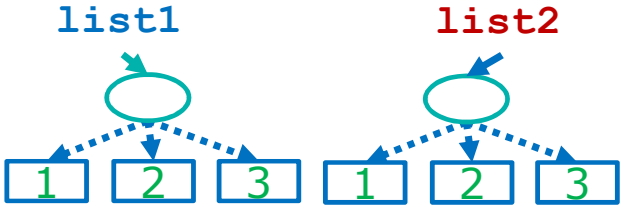

```
# Dann: …
list2 = list2.set(1, 99)
print(list1, id(list1), list2, id(list2))
# => [1, 2, 3] 50250384 [1, 99, 3] 50251600
```



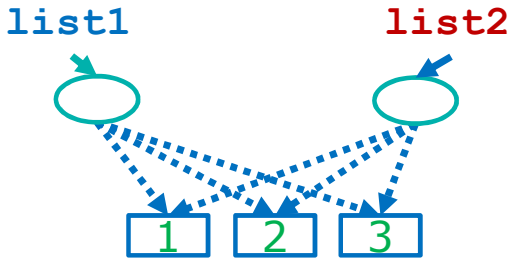
Datenkopien: Deep versus Shallow Copy



Deep Copy:



Shallow Copy:



Um die Seiteneffekte bei Mutable Data zu vermeiden, ist bei Container-Datenstrukturen meist Deep Copying erforderlich.

Datenkopien: Deep versus Shallow Copy

Aber nicht immer ergibt Deep Copy das gewünschte Ergebnis:

```
class Person: pass

peter = Person()
peter.name = "Peter" ; peter.alter = 30

class Auto: pass

vw = Auto()
vw.fabrikat = "VW" ; vw.person = peter
privat_auto = vw ; firmen_auto = vw

import copy
firmen_auto = copy.deepcopy(firmen_auto)
firmen_auto.fabrikat = "Audi" # Deep Copy wurde gemacht, damit diese Änderung nicht das Privatauto betrifft

firmen_auto.person.alter = 40

print(peter.alter, privat_auto.person.alter, firmen_auto.person.alter)
# => (30, 30, 40)  # gewolltes "Sharing" der Person (ein Besitzer für beide Autos) wurde durch Deep-Copy aufgehoben!
```

Mutability ist problematisch …

… z.B. mutierende Key-Werte in Hashes/Dictionaries:

```
# Beispiel:

hash1 = {}
a = [1, 2]
hash1[a] = "value1"
puts hash1      # => { [1, 2] => "value1" }

a << 3          # ... "append" mutiert das Array (Arrays sind mutable in Ruby ...)
puts a          # => [1, 2, 3] , die Liste ist also wie erwartet [1,2,3]
puts hash1[a]   # => nil Im Container gibt es keinen Eintrag für [1,2,3]
                # Der Container hat (scheinbar) immer noch mit dem Key-Wert [1,2]
puts hash1      # => { [1, 2, 3] => "value1" } !!! Schlimmer noch: Der
                # Container hat die Mutation nicht bemerkt und arbeitet
                # mit einem anderen Indexwert als dem, den er anzeigt!!!!
hash1[ [1,2,3] ] = "value2"
puts hash1      # => { [1, 2, 3] => "value1" , [1, 2, 3] => "value2"}
                # Das ist eigentlich unmöglich. Key-Wert darf nicht doppelt sein!
```

Mutability ist problematisch …

… z.B. mutierende Key-Werte in Hashes/Dictionaries.

```
# Python erlaubt keine Mutable Datentypen als Dictionary Keys ...

d = { [1,2]: "value1" }

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'


# Tupel sind hingegen immutable in Python ...

d = { (1,2): "value1" }

print(d)
# => {(1, 2): 'value1'}
```

Mutability ist problematisch …

Default-Parameterwerte von Funktionen werden in Python nur einmal angelegt & ausgewertet!

Mutation dann häufig problematisch.

```
def fun( arg1 = {} ):
    print("Wert des Parameters: " + str(arg1))
    arg1["key"] = "value"

fun() ;  fun()
```

… ergibt …

```
# => Wert des Parameters: {}
# => Wert des Parameters: {'key': 'value'}
```

Mutable versus Immutable Datentypen

… in Python, Ruby, C++.

	Python	**Ruby**	**C++** ***(via Referenzen)***	***Funktionale Prg*** ***z.B. Clojure, Erlang***
Integer	Immutable	Immutable	Mutable	Immutable
String	Immutable	Mutable	Mutable	Immutable
List/Array	Mutable *(Tupel: immutable)*	Mutable	Mutable	Immutable
Dict/Hash	Mutable *(immutable keys)*	Mutable	Mutable	Immutable
Objekt/ Klasse	Mutable	Mutable	Mutable	Immutable