ETH zürich

Übungslektion 3 – Comprehension & Numpy

Informatik II

4. / 5. März 2025

Heutiges Programm

Wiederholung von Kursinhalten

Numpy

Rekursion

Lektionsübung

Hausaufgaben

1. Wiederholung von Kursinhalten

Listen-Abstraktion

Erstellen einer Liste aus einer Funktion und einem Behälter.

```
l = [f(x) for x in c]
```

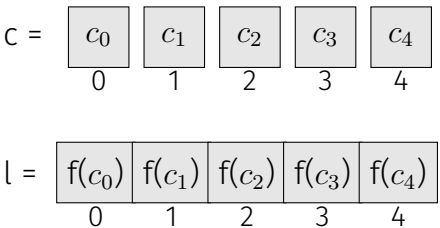
Äquivalenter **C++**-Code:

```
std::vector<double> l(c.size());
for(int i=0;i<c.size();i++)
{
      l[i] = f(c[i]);
}
```

Listen-Abstraktion

Erstellen einer Liste aus einer Funktion und einem Behälter.

```
l = [f(x) for x in c]
```



code bsp doubler

x ** 0,5

Listen-Abstraktion: Quiz

Was ist die Ausgabe des folgenden Codes? range(2,7) = (2,3,4,5,6)

```
[x**2 for x in range(2,7)]
```

[4, 8, 16, 25, 36]

Wie kann man die folgende Liste mittels Listen-Abstraktion generieren?

```
[1, 2, 4, 8, 16, 32, 64, 128]
```

potenzen

2^1 2^3 2^6 2^7

2**x for x in range(8)

Gefilterte Listen-Abstraktion

```
l = [f(x) for x in c if b(x)]
```

Äquivalenter **C++**-Code:

```
std::vector<double> l;
for(int i=0;i<c.size();i++)
{
    if(b(x[i]))
    {
        l.push_back(f(c[i]));
    }
}
```

Gefilterte Listen-Abstraktion

```
l = [f(x) for x in c if b(x)]
```

condition

c =

c_0	c_1	c_2	c_3	c_4
0	1	2	3	4

b(x):

T	T	F	T	F
0	1	2	3	4

l =

f(c_0)	f(c_1)	f(c_3)
0	1	2

code bsp:

x >= 2

doubles if x > 2

Gefilterte Listen-Abstraktion: Quiz

Was ist die Ausgabe des folgenden Codes?

```
[x**3 for x in range(6) if x%2==1]
```

[1, 27, 125]

Wie kann man die folgenden Listen mittels gefilterter Listen-Abstraktion generieren?

```
[25, 16, 9, 4, 4, 9, 16, 25]
```

Sammlungen

- **set**

```
s = {1, 6, 2, 7}
```

C++-Äquivalente: `std::set`, `std::unordered_set`

- **dictionary (dict)** *key:value*

```
d = {1:3, 6:2, 2:6, 7:5}
```

key value

C++-Äquivalente: `std::map`, `std::unordered_map`

Dict-Operationen

Dictionary besteht aus Paaren `key:val`

```
d = {"Banana":2.4, "Apple":3.2, "Orange":3.6}
```

- Item zugreifen

```
d[key]
```

- Item hinzufügen

```
d[key] = val
```

- Item modifizieren

```
d[key] = val
```

- Key suchen

```
key in d
```

- Key löschen

```
del d[key]
```

zeigen an Vscode

Dict-Iteration

```
d = {"Banana":2.4, "Apple":3.2, "Orange":3.6}
```

- Über keys

```
for key in d.keys():
    print(key)
```

```
Banana
Apple
Orange
```

- Über values

```
for val in d.values():
    print(val)
```

```
2.4
3.2
3.6
```

- Über Paare

```
for key, val in d.items():
    print(str(key)+" "+str(val))
```

```
Banana 2.4
Apple 3.2
Orange 3.6
```

Dict mittels zwei Listen

Liste `k` von keys, `v` von values:

```
d = dict(zip(k,v))
```

Beispiel:

```
stadt = ["Zurich", "Basel", "Bern"]
plz = [8000, 3000, 4000]
d = dict(zip(stadt,plz))
```


```
{'Zurich': 8000, 'Basel': 3000, 'Bern': 4000}
```

Dict: Quiz

Gehe von den folgenden zwei Listen aus:

```
brand = ["Lindt", "Cailler", "Frey"]
cost = [3.2, 2.5, 2.0]
```

Wie sieht dict `d` nach jedem Schritt aus?

```
d = dict(zip(brand,cost))
d["Halba"] = 1.9
d["Frey"] = 2.1
del d["Cailler"]
```

Dict-Abstraktion

Erstellen eines Dict aus einer Sammlung und zwei Funktionen

```
d = {f(x):g(x) for x in c}
```

Äquivalenter **C++**-Code:

```
std::unordered_map<double, double> d;
for(int i=0;i<c.size();i++)
{
    d[f(c[i])] = g(c[i]);
}
```

Dict-Abstraktion

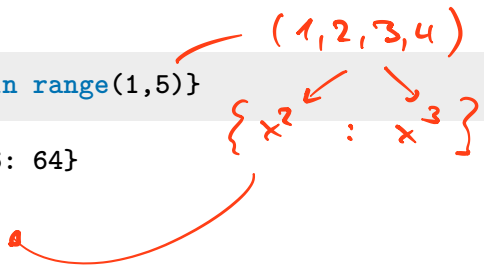
Erstellen eines Dict aus einer Sammlung und zwei Funktionen

```
d = {f(x):g(x) for x in c}
```

Beispiel:

```
{(x**2):(x**3) for x in range(1,5)}
```

```
{1: 1, 4: 8, 9: 27, 16: 64}
```



Dict-Abstraktion

Mit Filter `b(x)`

```
d = {f(x):g(x) for x in c if b(x)}
```

condition

Aus zwei Sammlungen `cx, cy`

```
d = {f(x):g(y) for x, y in zip(cx, cy)}
```

Aus einem anderen Dict `d0`

```
d = {f(k):g(v) for k, v in d0.items()}
```

Aliasing

- Aliasing tritt auf, wenn der Wert einer Variablen einer anderen Variablen zugewiesen wird.
- Variablen sind nur Namen, die Verweise auf den tatsächlichen Wert speichern.
- In Python ist alles ein Zeiger!

```
first_variable = "PYTHON"
print("Value of first:", first_variable)
print("Reference of first:", id(first_variable))
```

```
Value of first: PYTHON
Reference of first: 4349862704
```

```
second_variable = first_variable # making an alias
print("Value of second:", second_variable)
print("Reference of second:", id(second_variable))
```

```
Value of second: PYTHON
Reference of second: 4349862704
```

Aliasing

- Das Ändern eines Wertes führt zu einer Änderung des Zeigers.

```
second_variable = 42 # changing the value
print("Value of first:", first_variable)
print("Reference of first:", id(first_variable))
print("Value of second:", second_variable)
print("Reference of second:", id(second_variable))
```

```
Value of first: PYTHON
Reference of first: 4349862704
Value of second: 42
Reference of second: 4308446800
```

- Änderungen an Elementen innerhalb einer Liste führen **nicht** zu einer Änderung des Zeigers.

```
l1 = ["a", "b", "c"]
l2 = l1
l1[1] = "d"
print(l2)
l2[1] = "e"
print(l1)
```

```
['a', 'd', 'c']
['a', 'e', 'c']
```

Aliasing von Funktionen

- Aliasing gilt auch für Funktionen. Mittels Aliasing kann man bestehenden Funktionen neue Namen zuweisen.

```
def fun(name):
  print(f"Hello {name}, welcome to Informatik II!!!")

cheer = fun
print("The id of fun():", id(fun))
print("The id of cheer():", id(cheer))

fun('everyone')
cheer('students')
```

```
The id of fun(): 4408778960
The id of cheer(): 4408778960
Hello everyone, welcome to Informatik II!!!
Hello students, welcome to Informatik II!!!
```

Aliasing von Funktionen

- Aliasing kann auch auf Funktionen von Objekten angewendet werden.

```
class Test:
  def __init__(self):
    self._name = "original name"
  def name(self):
    return self._name

# create object of Test class
test = Test()

# create reference to name and its alias
name_fn = test.name
name_fn_ref = name_fn

print(name_fn())

test._name = "modified name"

print(name_fn_ref())
```

```
original name
modified name
```

Aliasing: Quiz

- Was wird auf dem Bildschirm ausgegeben?

```
alist = [4, 2, 8, 6, 5]
blist = alist
blist[3] = 999
print(alist)
```

```
A. [4, 2, 8, 6, 5]
B. [4, 2, 8, 999, 5]
```

- Welche Vergleiche geben in Anbetracht der folgenden Listen 'True' aus? (Wähle alle, die zutreffen).

```
list1=[1,100,1000]
list2=[1,100,1000]
list3=list1
```

```
A. print(list1 == list2)
B. print(list1 is list2)
C. print(list1 is list3)
D. print(list2 is not list3)
E. print(list2 != list3)
```

2. Numpy

Erstellung von Numpy Arrays

```
import numpy as np
```

Erstellen eines Numpy Array aus einer Sequenz.

```
a = np.array([1, 2, 3, 4])
b = np.array(range(5, 0, -1)) #[5, 4, 3, 2, 1]
c = np.array([[1, 2], [3, 4]]) # two dimensional array
```

Erstellen eines Numpy Array mit **`arange()`**. Ähnlich wie bei **`np.array(range())`**.

```
a = np.arange(1, 5, 2) # [1, 3]
b = np.arange(1, 5) # step = 1, [1, 2, 3, 4]
c = np.arange(5) # start = 0, step = 1, [0, 1, 2, 3, 4]
```

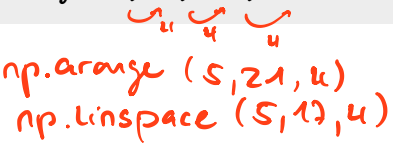
Erstellung von Numpy Arrays & Quiz

Erstellen Sie ein Numpy-Array mit `linspace()`. Stopp ist **inklusive**.

```
a = np.linspace(2, 10, 5) # [2., 4., 6., 8., 10.]
a = np.linspace(2, 100) # num = 50, [2., 4., 6., ..., 100.]
```

Quiz: Wie kann man das folgende Array mit Hilfe von `arange()` bzw. `linspace()` erzeugen?

```
array([5, 9, 13, 17])
```



Numpy Array Operationen

Print

```
print(np.array([2, 2, 6, 8])) #[2 2 6 8]
```

Länge und Größe

```
a = np.array([1, 2, 3, 4, 5, 6]) # one dimension
len(a) # 6
a.size # 6
b = np.array([[1, 2], [3, 4], [5, 6]]) # two dimensions
len(b) # 3
len(b[0]) # 2
b.size # 6
```

Numpy Array Operationen

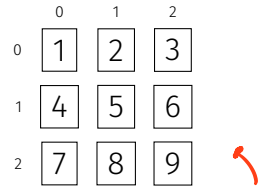
Zugriff auf ein Element

```
a = np.array([1, 2, 3, 4, 5, 6]) # one dimension
a[2] # 3
a[-2] # a[-2] = a[-2 + len(a)] = a[4] = 5
```

```
b = np.array([[1, 2], [3, 4], [5, 6]]) # two dimensions
b[1] # array([3, 4])
b[1, 0] # b[1, 0] = b[1][0] = 3
```

Gefilterte Listen-Abstraktion

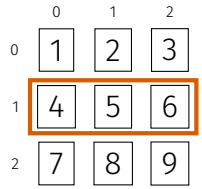
Zerschneiden




```
A = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

Gefilterte Listen-Abstraktion

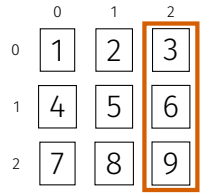
Zerschneiden




```
A = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
A[1] # = A[1,:]  = array([4, 5, 6]) (Zeile)
```

Gefilterte Listen-Abstraktion

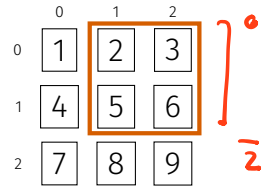
Zerschneiden




```
A = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
A[1] # = A[1,:]  = array([4, 5, 6]) (Zeile)
A[:, 2] # array([3, 6, 9]) (Spalte)
```

Gefilterte Listen-Abstraktion

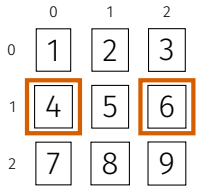
Zerschneiden




```
A = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
A[1] # = A[1,:]  = array([4, 5, 6]) (Zeile)
A[:, 2] # array([3, 6, 9]) (Spalte)
A[0:2, 1:3] # array([[2, 3], [5, 6]])
```

Gefilterte Listen-Abstraktion

Zerschneiden



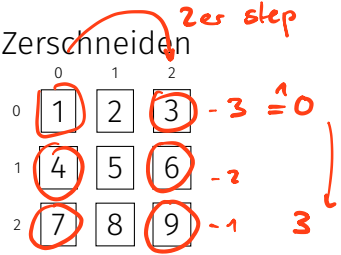

```
A = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
A[1] # = A[1,:]  = array([4, 5, 6]) (Zeile)
A[:, 2] # array([3, 6, 9]) (Spalte)
A[0:2, 1:3] # array([[2, 3], [5, 6]])
A[1:2, ::2] # = A[1:2, 0:3:2] = [[4, 6]]
```

A[1, ::2] ← zweiter schritt

↑ reihe 1

Gefilterte Listen-Abstraktion

Zerschneiden




```
A = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
A[1] # = A[1,:]  = array([4, 5, 6]) (Zeile)
A[:, 2] # array([3, 6, 9]) (Spalte)
A[0:2, 1:3] # array([[2, 3], [5, 6]])
A[1:2, ::2] # = A[1:2, 0:3:2] = [[4, 6]]
```

Quiz: `A[-3:3, ::2] = ?`

0→3 alle, 2weise wie array([[1,3],[4,6],[7,9]])

Numpy Array Statistiken

Numpy Array Statistiken

```
a = np.array([5, 6, 7, 8, 1, 2, 3, 4])
a.min() # 1
a.max() # 8
np.mean(a) # 4.5
a.sum() # 36
np.std(a) # standard deviation, 2.291
```

Numpy Array Operationen

Elementweise Operationen

```
A = np.array([[1, 2, 3], [4, 5, 6]])
A + 1 # [[2, 3, 4], [5, 6, 7]]
A * 3 # [[3, 6, 9], [12, 15, 18]]
A ** 2 # [[1, 4, 9], [16, 25, 36]]
np.sin(A) # [[sin(1), sin(2), sin(3)], [sin(4), sin(5), sin(6)]]
B = np.array([[4, 5, 6], [7, 8, 9]])
A + B # [[5, 7, 9], [11, 13, 15]]
A * B # [[4, 10, 18], [28, 40, 54]]
```

Matrix Multiplikation

```
A @ np.array([[1, 4], [3, 4], [4,6]]) # [[19, 30], [43, 72]]
```

Numpy Array Filterung & Quiz

```
a = np.arange(1, 10)

f = a % 3 == 0

a[f]
```

a:	1	2	3	4	5	6	7	8	9
f:	F	F	T	F	F	T	F	F	T
a[f]:	3	6	9						

Quiz

```
a[a**2 < 20].sum() = ?
```

condition

$\{1^2, 2^2, 3^2, 4^2, 5^2 \ldots\}$

$[1, 2, 3, 4].sum() = 10$

3. Rekursion

Rekursion

- Eine Funktion wird als **rekursiv** bezeichnet, wenn sie sich selbst aufruft.
- Die Idee ist es ein großes Problem in **kleinere sich wiederholende Teile** desselben Problems aufzuteilen.
- Jeder rekursive Algorithmus beinhaltet mindestens 2 Fälle:
  - **Basisfall**: Ein einfaches Problem, das direkt beantwortet werden kann.
  - **Rekursiver Fall**: Ein komplexeres Auftreten des Problems, das nicht direkt beantwortet werden kann.
- Einige rekursive Algorithmen haben mehr als einen Basis- oder rekursiven Fall, aber alle haben mindestens einen von beiden.

Rekursion: Beispiel

- Betrachte die folgende Funktion, um eine Zeile mit *-Zeichen auszugeben:

```
def printStars(n):
  for _ in range(n):
    print("*", end = ' ')
  print()

printStars(5)
```

```
* * * * *
```

- Schreibe eine rekursive Version dieser Funktion (ohne Schleifen zu verwenden).

Rekursion: Beispiel Basisfall

- Was ist der Basisfall?

```
def printStars(n):
  """base case; just print one star"""
  if n == 1:
    print("*")
  else:
    ...

printStars(5)
```

Rekursion: Beispiel Weitere Fälle

- Umgang mit weiteren Fällen, ohne Schleifen zu verwenden (auf eine schlechte Weise):

```
def printStars(n):
  """base case; just print one star"""
  if n == 1:
    print("*")
  elif n == 2:
    print("*", end = ' ')
    printStars(1)
  elif n == 3:
    print("*", end = ' ')
    printStars(2)
  elif n == 4:
    print("*", end = ' ')
    printStars(3)
  else:
    ...

printStars(5)
```

Rekursion: Beispiel Rekursion Richtig Verwenden

- Zusammenfassen der rekursiven Fälle zu einem einzigen Fall:

```
def printStars(n):
  """base case; just print one star"""
  if n == 1:
    print("*")
  else:
    """recursive case"""
    print("*", end = ' ')
    printStars(n - 1)

printStars(5)
```

- Die obere Funktion geht davon aus, dass der kleinste Wert 1 ist, aber was wenn wir wollen dass der kleinste wert 0 ist?

Rekursion: Beispiel "Rekursion Zen"

- **Recursion Zen**: Die Kunst, die besten Fälle für einen rekursiven Algorithmus richtig zu identifizieren und elegant zu programmieren.

```
def printStars(n):
  """base case; just end the line of output"""
  if n == 0:
    print()
  else:
    """recursive case; print one more star"""
    print("*", end = ' ')
    printStars(n - 1)

printStars(5)
```

Rekursion: Beispiel Fakultät

- Die Fakultät einer Zahl ist das Produkt aller Zahlen von 1 bis zu dieser Zahl. Zum Beispiel ist die Fakultät von 6 (auch bezeichnet als 6!) 1*2*3*4*5*6 = 720.
- Beispiel einer rekursiven Funktion zum Ermitteln der Fakultät einer Zahl:

```
def factorial(x):
  if x == 1:
    """base case"""
    return 1
  else:
    """recursive case"""
    return (x * factorial(x-1))

num = 3
print("The factorial of", num, "is", factorial(num))
```

```
The factorial of 3 is 6
```

```
factorial(3)          # 1st call with 3
3 * factorial(2)      # 2nd call with 2
3 * 2 * factorial(1)  # 3rd call with 1
3 * 2 * 1             # return from 3rd call as number=1
3 * 2                 # return from 2nd call
6                     # return from 1st call
```

Rekursion: Vor- und Nachteile

Vorteile

- Rekursive Funktionen lassen den Code **sauber und elegant** aussehen.
- Komplexe Aufgaben können durch Rekursion in **einfachere Teilprobleme** zerlegt werden.

Nachteile

- Rekursive Aufrufe sind meistens teuer (**ineffizient**), da sie viel Speicher und Zeit beanspruchen.
- Rekursive Funktionen sind **schwer zu debuggen**, da es manchmal schwierig ist, der Logik hinter der Rekursion zu folgen.

Rekursion: Stack Overflow

- Jede rekursive Funktion muss eine **Grundbedingung** haben, die die Rekursion stoppt, sonst ruft sich die Funktion endlos selbst auf.
- Der Python-Interpreter **begrenzt** die Rekursionstiefe, um unendliche Rekursionen zu vermeiden die zu einem Stack Overflow führen.
- Standardmäßig beträgt die maximale Rekursionstiefe **1000**. Wird die Grenze überschritten, führt dies zu einem `RecursionError`.

```
def recursor():
    recursor()
recursor()
```

```
Cell In[1], line 2, in recursor()
      1 def recursor():
----> 2     recursor()

RecursionError: maximum recursion depth exceeded
```

Rekursion: Quiz

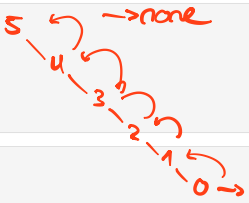
- In welcher Datenstruktur werden Rekursionsaufrufe im Speicher abgelegt?

```
A. Heap
B. Stack
C. Tree
```

- Was ist die Ausgabe des unten angegebenen Codes?

```
def pprint(n):
  if n == 0:
    return
  else:
    return pprint(n-1)

print(pprint(5))
```




```
A. 5
B. 5 4 3 2 1
C. None
D. RecursionError
```

4. Lektionsübung

Lektionsübung: Einheitsmatrix

Eine **Einheitsmatrix** oder **Identitätsmatrix** ist eine quadratische Matrix, deren Elemente auf der Hauptdiagonale eins und überall sonst null sind.

Schreibe ein Python Programm welches:

- Die Größe n der Einheitsmatrix als Input nimmt und die Matrix als verschachtelte Liste ausgibt.
- Die Matrix muß nicht weiter formatiert sein. Jedoch muß man auf jedes Element der Matrix I mittels $I[r][c]$ zugreifen können, wo r und c die Inidces der Reihe und Spalte sind.

Hinweise: Listen können mit einer Konstanten multipliziert werden. Außerdem, können List Comprehensions und Python's Ternäroperator zu einer kompakteren Lösung führen.

Lektionsübung: Numpy Array

Code Expert — Numpy Array Slicing & Masking

Schreibe ein Python Programm welches:

- Erzeugen Sie ein zweidimensionales Feld (Aufgabe 1).
- Schneiden Sie das Array und berechnen Sie seine statistischen Ergebnisse (Aufgabe 2, 3).
- Filtern Sie das Array und berechnen Sie seine statistischen Ergebnisse (Aufgabe 4-6).

Siehe detaillierte Aufgabenbeschreibung in Code Expert.

5. Hausaufgaben

Übung 2: Python II

Auf `https://expert.ethz.ch/mycourses/SS25/mavt2/exercises`

Exercise 2: Python II

- String Reverse
- Skalarprodukt
- Suchen
- Dict Comprehension
- List Comprehension

Fällig bis Montag, 10.03.2025, 20:00 MEZ

KEINE HARDCODIERUNG

Fragen?