Informatik II Woche 3

List & Dict Comprehension, Aliasing, Numpy, Rekursion

Website: https://n.ethz.ch/~kvaratharaja/

Die Slides basieren auf den offiziellen Übungsslides der Kurswebsite: https://lec.inf.ethz.ch/mavt/informatik2/2025/

Heute

1. List / List Comprehension
2. Dict / Dict Comprehension
3. Aliasing
4. Numpy
5. Rekursion
6. In-class Exercises
7. Hausaufgaben

Collections

set

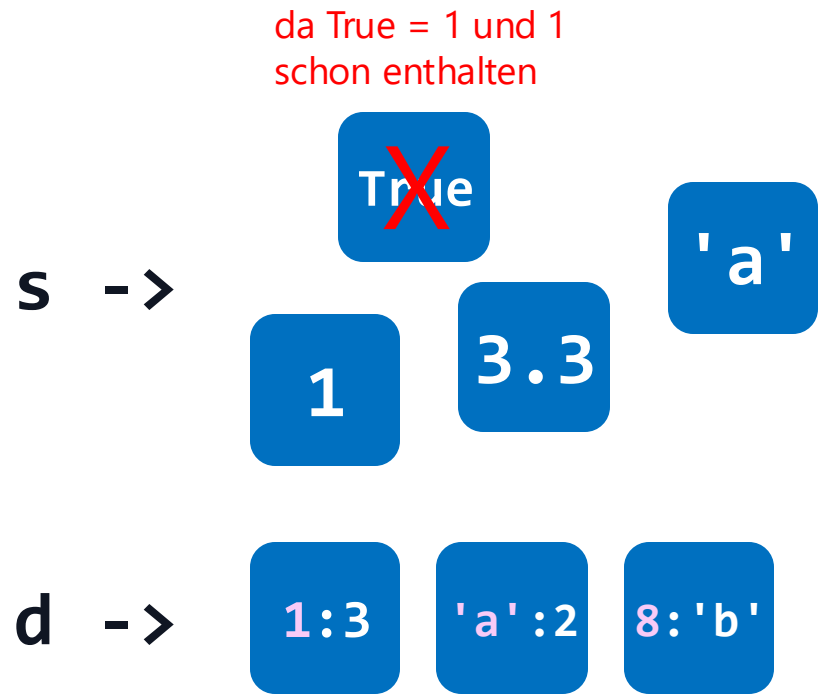
```
s = {1, True, 'a', 3.3}
```

C++ equivalents: `std::set`, `std::unordered_set`

dictionary (dict) key:value

```
d = {1:3, 'a':2, 8:'b'}
```

C++ equivalents: `std::map`, `std::unordered_map`



1. Lists Recap

List Operations

```
l = [5, 7, 2]
l[2] = 8
print(l) #output: [5,7,8]
```

Element ändern

```
l[i] = val
```

Liste umkehren

```
l.reverse()
```

Element am Ende anhängen

```
l.append(val)
```

Liste mit k Elementen mit Wert val erstellen

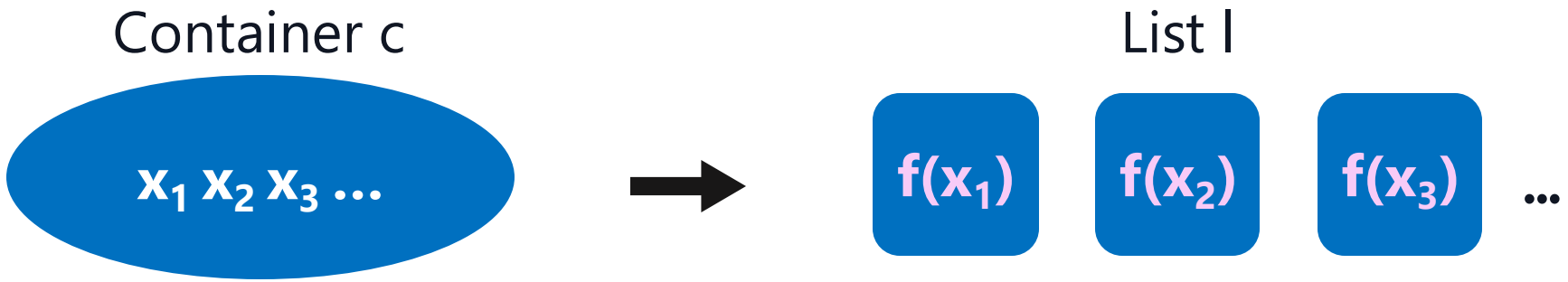
```
l = [val] * k
```

Element entfernen

```
del l[i]
```

List-Comprehension

c kann String, Tuple, range, list, set, dict sein




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

equivalent: l = list(f(x) for x in c if g(c))

List-Comprehension

Erstellen einer Liste aus einer Funktion und aus einem Container

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

List-Comprehension Quiz

Was ist die Ausgabe des folgenden Codes?

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

[4,9,16,25,36]

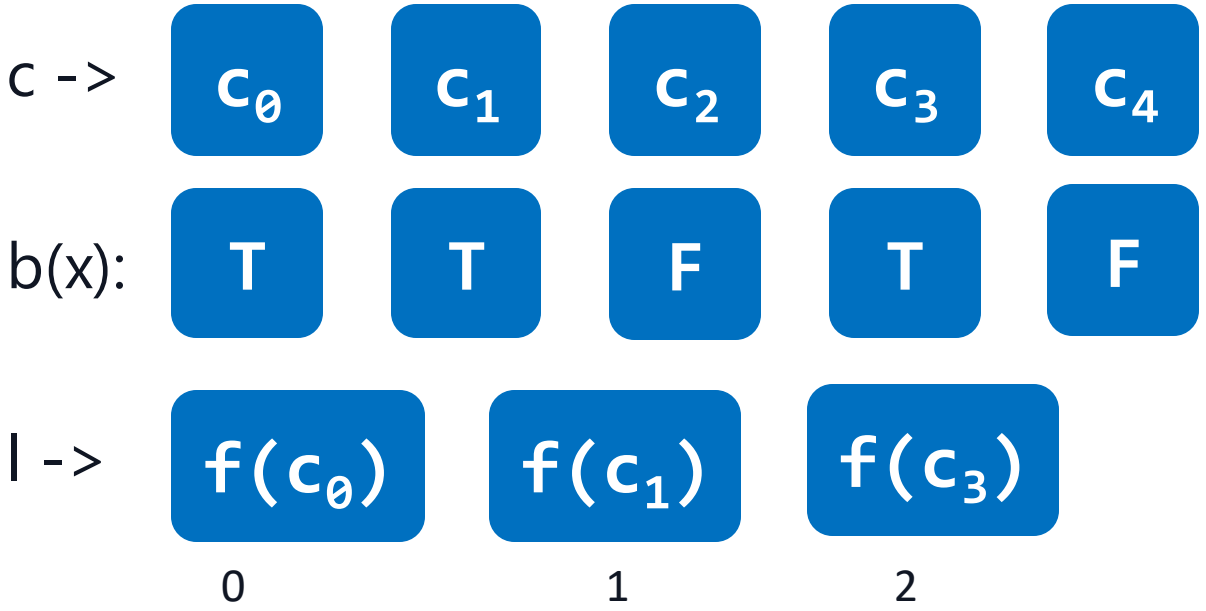
Wie kann man die folgende Liste mittels List-Comprehension generieren?

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

[2**x for x in range(8)]

Gefilterte List-Comprehension

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



Gefilterte List-Comprehension 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 folgende Liste mittels gefilterter List-Comprehension generieren?

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

[x**2 for x in range(-5,6) if x**2 >1] # or x**2 > 2, x**2 >3

Gefilterte List-Comprehension Quiz

Gegeben sind 2 Listen, erstelle eine Liste, deren Elemente die Elemente beider Listen paarweise addiert. Die Liste soll nur Elemente strikt grösser als 10 enthalten

```
l1 = [5, 1, 8, 10]
```

```
l2 = [4, 10, 4, 0]
```

```
l3 = [11, 12]
```

l3 = [x + y for x,y in zip(l1,l2) if x + y > 10]

2. Dicts Recap

Dicts

- “Ein Dictionairy ist ein Container, bei dem man auf Werte mit einem Key (meistens ein String) zugreift"

```
dictionary = {
    key1: value1,
    key2: value2,
    key3: value3
}

dictionary[key1] #value1
```

Dict Operations

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

- Auf einen Wert zugreifen

```
d[key]
```

d["Apple"]

- Element hinzufügen

```
d[key] = value
```

d["Pineapple"] = 5.4

- Wert updaten

```
d[key] = value
```

d["Apple"] = 4.7

- Key enthalten? (True/False)

```
key in d
```

"Apple" in d
->gibt True zurück

- Element bei Key löschen

```
del d[key]
```

del d["Orange"]

Dict-Iterationen

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

- Über die Keys iterieren:

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

```
Banana
Apple
Orange
```

- Über die Werte iterieren:

```
for value in d.values()
    print(value)
```

```
2.4
3.2
3.6
```

- Über Paare:

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

```
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))
```

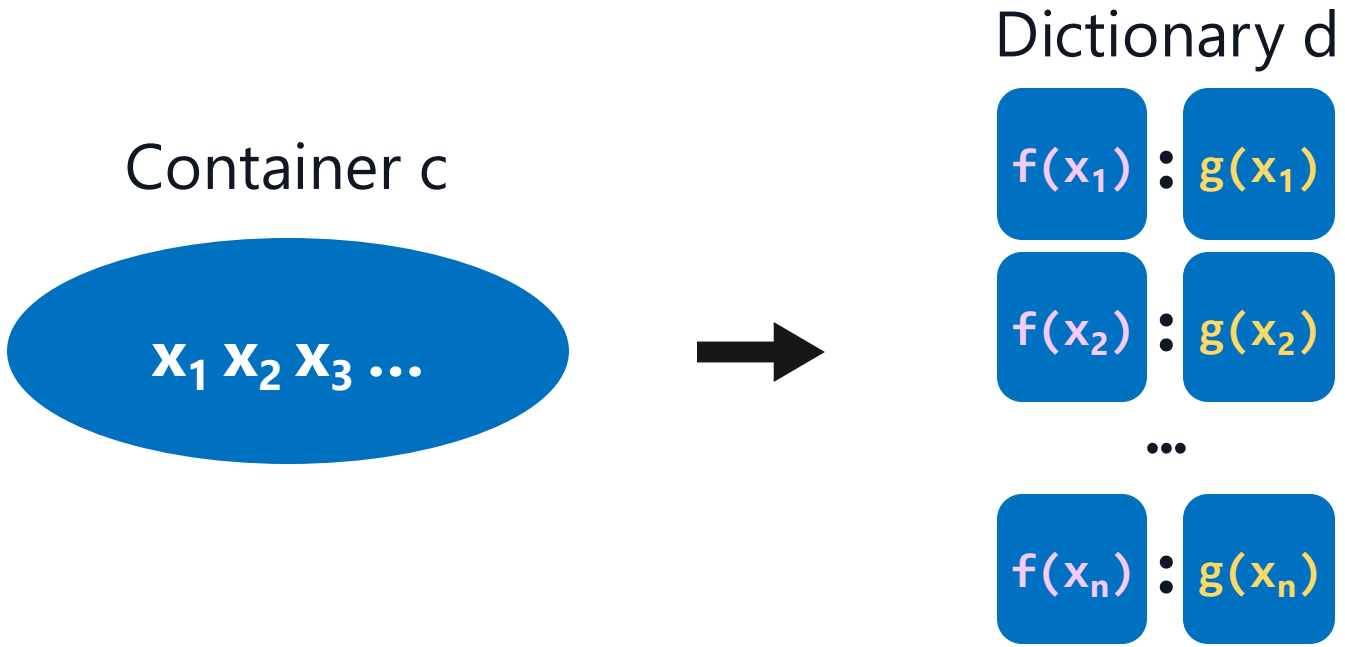
```
{"Zurich": 8000, "Basel": 3000, "Bern": 4000}
```

Dict: Quiz

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

{'Lindt': 3.2, 'Cailler': 2.5, 'Frey': 2.0}
{'Lindt': 3.2, 'Cailler': 2.5, 'Frey': 2.0, 'Halba': 1.9}
{'Lindt': 3.2, 'Cailler': 2.5, 'Frey': 2.1, 'Halba': 1.9}
{'Lindt': 3.2, 'Frey': 2.1, 'Halba': 1.9}

Dict-Comprehension



equivalent: d = dict(f(x):g(x) for x in c if h(c))

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

d = { f(x):g(y) for x, y in c if h(x,y) }
```

c ist meist ein zip

Dict-Comprehension

Erstellen eines Dict aus einem Container 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-Comprehension

Mit Filter b(x)

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

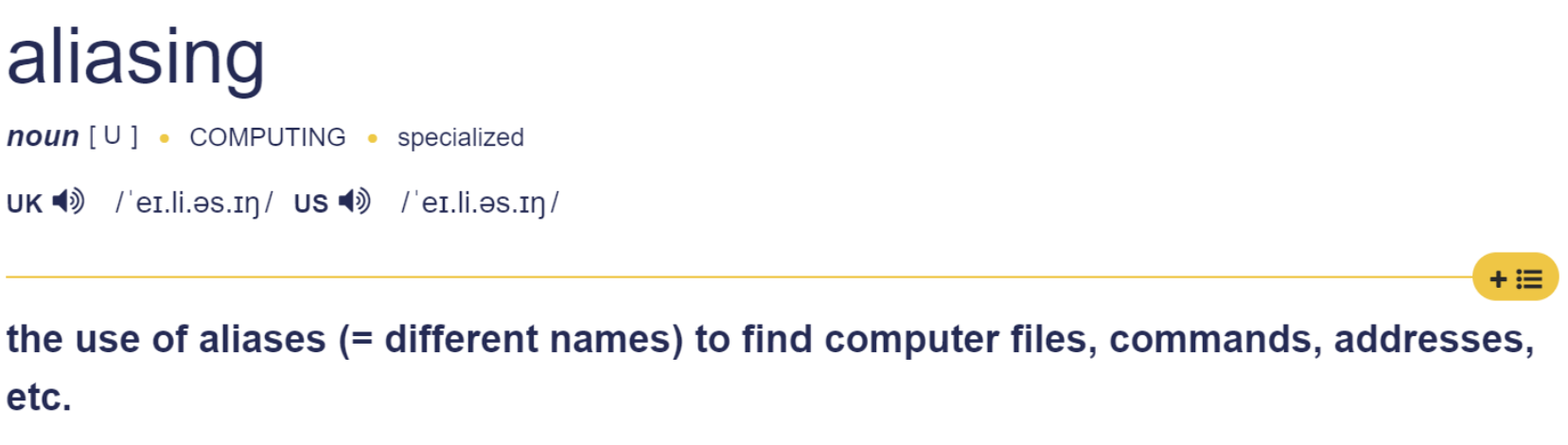
Aus zwei Collections cx, cy

```
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()}
```

3. Aliasing


aliasing
noun [U] • COMPUTING • specialized
UK /ˈeɪ.li.əs.ɪŋ/ US /ˈeɪ.li.əs.ɪŋ/
the use of aliases (= different names) to find computer files, commands, addresses, etc.

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 Pointer!**

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

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

Console Output:

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

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

Aliasing

- Das Ändern eines Wertes führt zu einer Änderung des Pointers

```
first_variable = "PYTHON"
second_variable = first_variable #making an alias
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))
```

Console Output:

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

List Aliasing

- Das Ändern von Werten innerhalb einer Liste führt **nicht** zu einer Änderung des Pointers:

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

Console Output:

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

Function Aliasing

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

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

cheer = fun
print("The id of fun():", id(fun))
print("The id of cheer():", id(cheer))
#create reference to name and its alias
fun('everyone')
cheer('students')
```

Console Output:

```
The id of fun(): 4408778960
The id of cheer(): 4408778960

Hello everyone, welcome to Info II!!!
Hello students, welcome to Info II!!!
```

Function Aliasing

- 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())
```

Console Output:

```
original name

modified name
```

Aliasing: Quiz 1

- What is the output?

```
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]`

Aliasing: Quiz 2

- 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)`

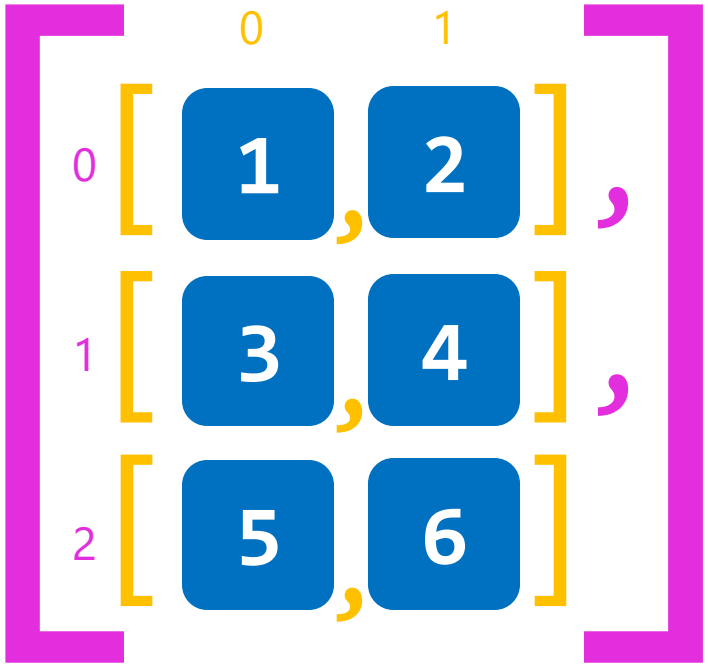
C. `print(list1 is list3)`

D. `print(list1 is not list2)`

E. `print(list2 != list2)`

4. Numpy

Numpy array




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

Numpy Array vs Lists

Python Lists	Numpy Arrays
Datentyp von Listelementen beliebig	Datentyp von Arrayelementen fix
keine fixe Grösse (non contiguous)	fixe Grösse (contiguous)
Eigentlich 1D (oder Liste in Liste)	Multi-Dimensional
keine built-in Element-wise operations	built-in Element-wise operations
langsamer (Performance)	schneller (Performance)

Numpy Array Erstellen

- Numpy Package importieren

```
import numpy as np
```

- Numpy array mithilfe einer Sequenz erstellen

```
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
```

- Numpy array mit `arange()` erstellen. Äquivalent zu `np.array(range())`

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

Numpy Array Erstellen - Linspace

- Ein Numpy array mit `linspace()` erstellen. Num ist die Anzahl an Array Elementen. Stop ist **inklusiv**!

```
np.linspace(start, stop, num)
np.linspace(start, stop) #num = 50 (default)
step = (stop - start) / (num - 1)
```

- Beispiel:

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

Numpy Array Erstellen – Quiz 1

- Zusammengefasst:

```
np.linspace(start, stop, num)
np.linspace(start, stop) #num = 50 (default)
step = (stop - start) / (num - 1)
```

```
np.arange(start, stop, step)
np.arange(start, stop) #step = 1 (default)
np.arange(stop) #start = 0, step = 1 (default)
```

- How do you generate the following array using `arange()` and `linspace()`?

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

np.arange(5,21,4), np.linspace(5,17,4)

Numpy Array Operations

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

- Print

```
print(a)
print(b)
```

Console Output:

```
[1 2 3 4 5 6]
[[1 2]
 [3 4]
 [5 6]]
```

- Length

```
len(a) #6
len(b) #3
```

- Size

```
a.size #6
b.size #6
```

- Shape

```
a.shape #(6,)
b.shape #(3,2)
```

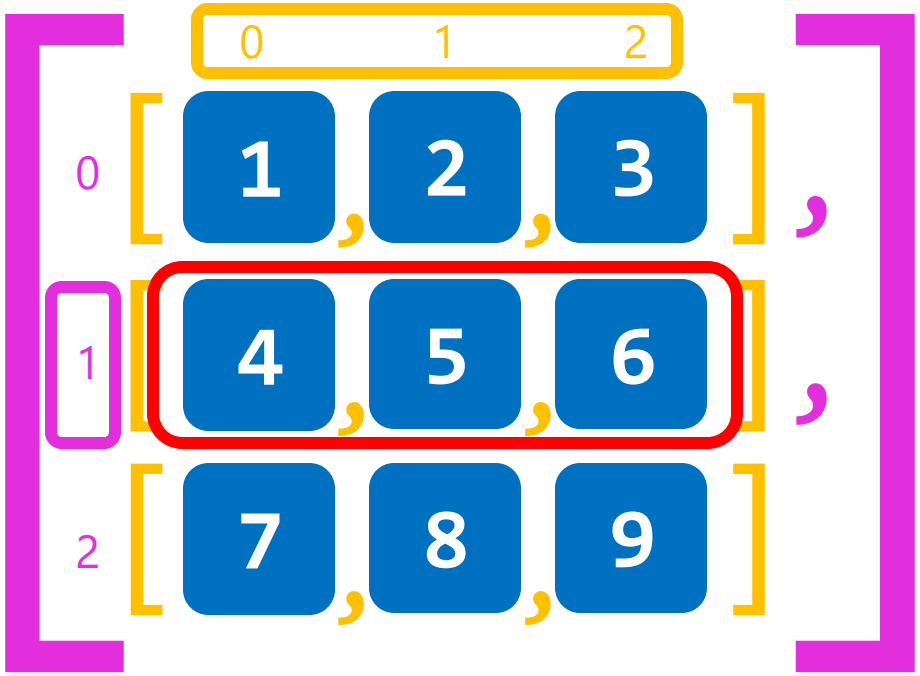
- Access element

```
a[2] #3
b[1,0] #3
b[1] #array([3,4])
b[:,1] #array([2, 4, 6])
```

- Transpose

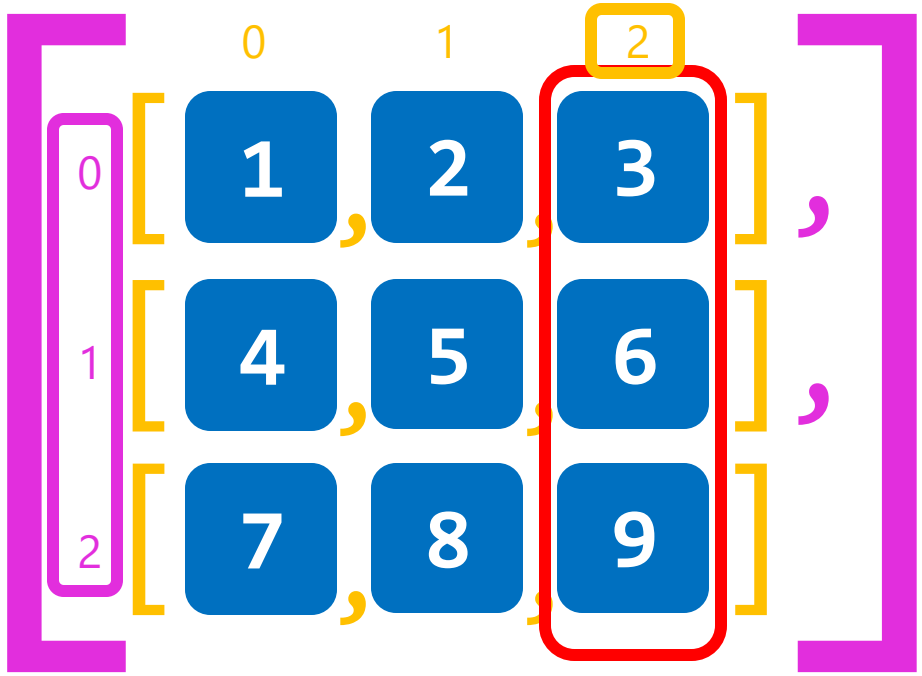
```
b.T
#b.shape = (2,3)
```

Numpy Array Operations




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

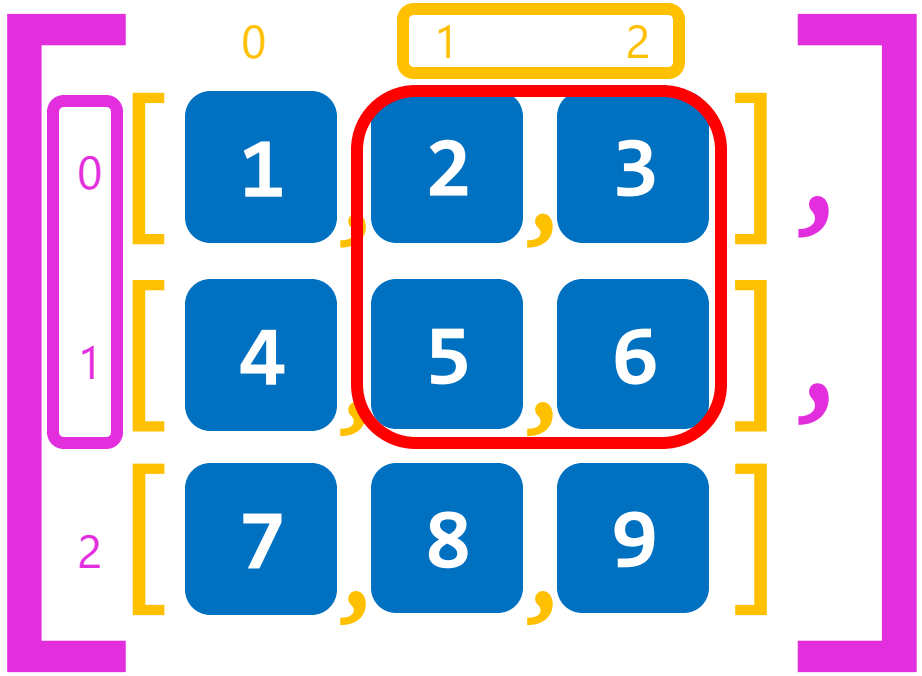
Numpy Array Operations




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

: bedeutet "alles"

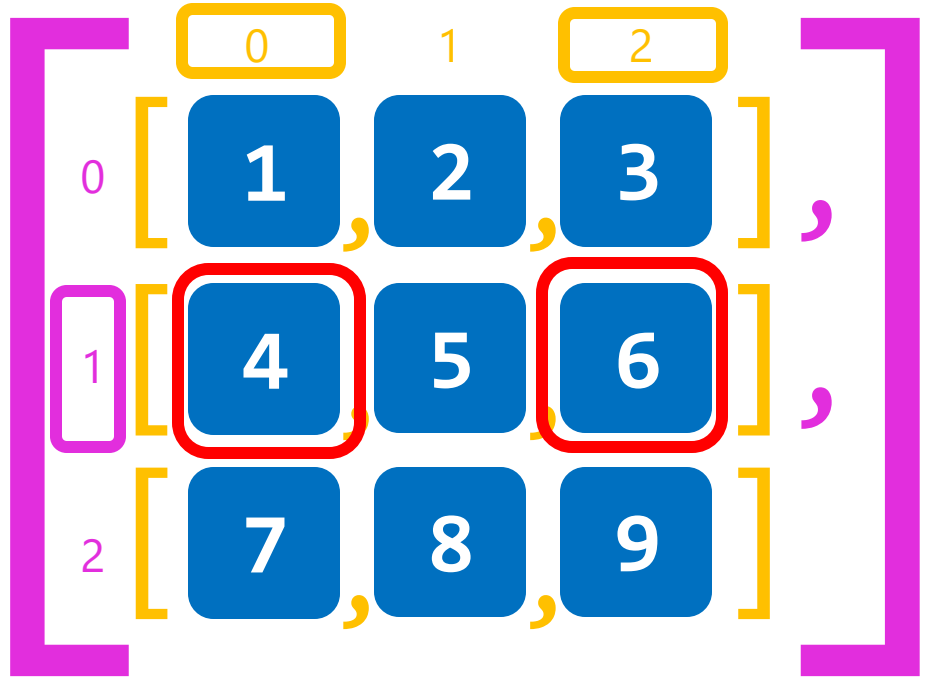
Numpy Array Operations




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

Slicing: Start:Stop ->step = 1

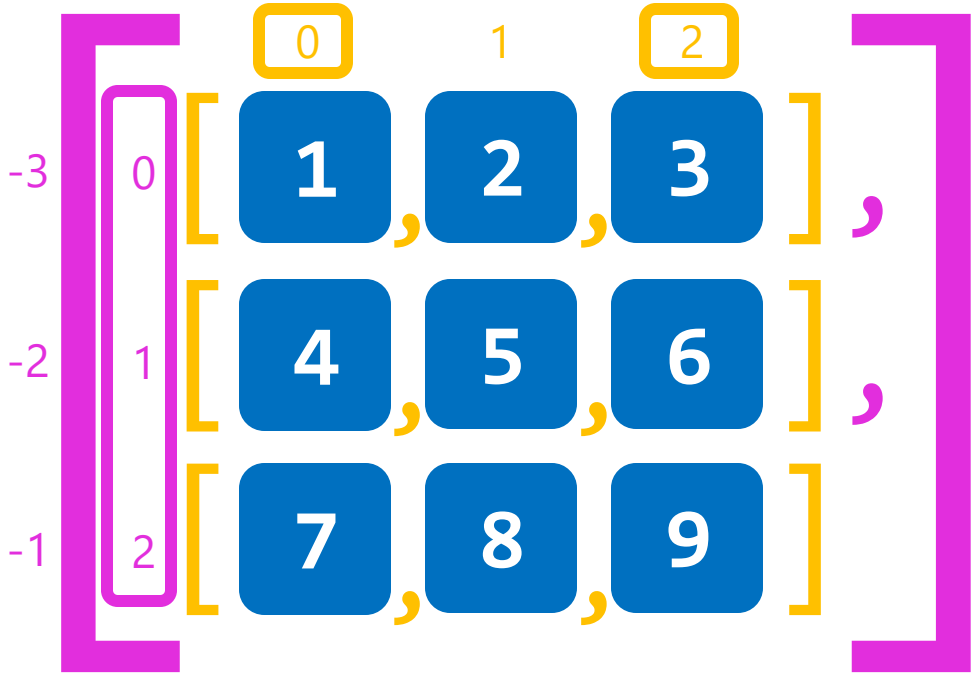
Numpy Array Operations




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

Slicing: Start:Stop ->step = 1
::2 -> Start = 0, Stop = len

Numpy Array Operations Quiz



- What is the value?

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

array([[1,3],
[4,6],
[7,9]])

Numpy Array Statistics

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

- Min und max value

```
a.min() #1
a.max() #8
```

- Durchschnitt (mean)

```
np.mean(a) #4.5
```

- Summe aller Elemente

```
a.sum() #36
```

- Standardabweichung

```
np.std(a) #2.291
```

Numpy Array Statistics - 2D

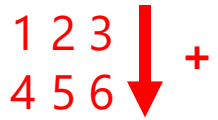
```
A = np.array([[1, 2, 3], [4, 5, 6]])
```

Alle Elemente von A aufsummieren

```
Result = A.sum() #21
```

Spalten von A aufsummieren

```
Result = A.sum(axis = 0) #[5, 7, 9]
```



Zeilen von A aufsummieren

```
Result = A.sum(axis = 1) #[6 15]
```

Numpy Array Operations

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

Element-wise operations

```
B = A + 1  #B = [[2, 3, 4], [5, 6, 7]]
C = A * 3  #C = [[3, 6, 9], [12, 15, 18]]
D = A ** 2  #D = [[1, 4, 9], [16, 25, 36]]
E = np.sin(A)  #E = [[sin(1),sin(2),sin(3)], [sin(4),sin(5),sin(6)]
F = A + B #F = [[3, 5, 7], [9, 11, 13]]
G = A * B #G = [[2, 6, 12], [20, 30, 42]]
```

Numpy Array Operations

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

Matrix Multiplikation

```
C = A @ B #C = [[19, 30], [43, 72]]
```

Numpy Array Filtering & Quiz

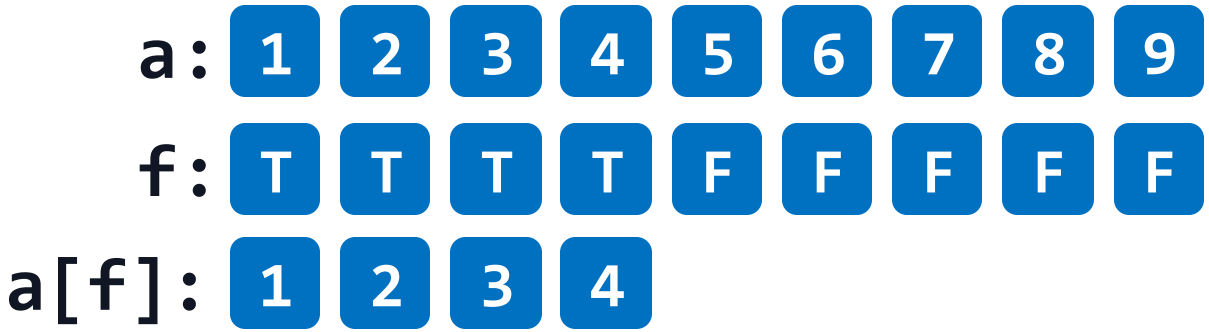
- Man kann Numpy Arrays auch filtern:

```
a = np.arange(1,10)
f = a % 3 == 0
a[f]
```

- Quiz: What is the Output?

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

1+2+3+4 = 10



5. 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:
  - **Base case**: Ein einfaches Problem, das direkt beantwortet werden kann.
  - **Recursive case**: 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

- Folgende Funktion gibt eine Zeile mit n *-Zeichen aus:

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

printStars(5)
```

Console Output:

*****

- Ziel: Eine rekursive Version dieser Funktion (ohne loops zu verwenden)

Rekursion Beispiel: Base Case

- Was ist der base case?

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

printStars(5)
```

Rekursion Beispiel: Weitere Fälle

- Umgang mit weiteren Fällen, ohne Schleifen zu verwenden (schlecht)

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

printStars(5)
```

Console Output:

*****

- 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: Zen of Recursion

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

```
def printStars(n):
    if n == 0:
        #base case: print new line
        print()
    else:
        #recursive case: just print one star
        print("*", end = ' ')
        printStars(n - 1)

printStars(5)
```

Console Output:

```
* * * * *
```

Rekursion Beispiel: Factorial

- Die Fakultät einer Zahl ist das Produkt aller ganzen Zahlen von 1 bis zu dieser Zahl. Beispiel: die Fakultät von 6 (Schreibweise: 6!) ist 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("Factorial of", num, "is", factorial(num))
```

Console Output:

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

Rekursion Beispiel: Factorial

```
def factorial(x):
    if x == 1:
        #base case: return 1
        return 1
    else:
        #recursive case: just print one star
        return(x * factorial(x-1))
num = 3
print("Factorial of", num, "is", factorial(num))
```

FUNCTION CALL	RETURN VALUE
factorial(3)	3*factorial(2)
factorial(2)	2*factorial(1)
factorial(1)	1

Rekursion: Vor- & Nachteile

Vorteile

- Rekursive Funktionen lassen den Code **clean 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.

Recursion: 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() #Calls itself infinitely

recursor()
```

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

RecursionError: maximum recursion depth exceeded
```

Rekursion: Quiz 1

- In welcher Datenstruktur werden Rekursionsaufrufe im Speicher abgelegt?

A. Heap

B. Stack

B. Tree

Rekursion: Quiz 2

- 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`

Rekursion: Quiz 3

Was ist die Ausgabe des unten angegebenen Codes?

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

print(mystery(5))
```

A. 5

B. 15

C. None

D. RecursionError

6. In-class Exercises

Python ternary operator

Ein ternary operator ist eine kompakte Weise, conditional statements in Python zu schreiben: in C++: val = condition ? true_val : false_val;

```
val = true_val if condition else false_val
```

Beispiel:

```
num = int(input("Enter a number : "))
msg = "Even" if num%2 == 0 else "Odd"
print(msg)
```

Dieser Code gibt "Even" for gerade Zahleninputs aus und "Odd" for ungerade

In-class exercise: 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:

- Als Input eine Grösse n nimmt und die Matrix als verschachtelte Liste ausgibt.
- Die Matrix muß nicht weiter formatiert sein. Jedoch muß man auf jedes Element der Matrix I über $I[r][c]$ zugreifen können, wobei r dem Index der Reihe und c dem Index der Spalte entspricht.

Hints: Listen können mit einer Konstanten multipliziert werden. Ausserdem können List Comprehensions und Python's Ternary Operator zu einer kompakteren Lösung führen.

In-class Exercise: Numpy Array Slicing & Masking

```
2 7 9 9 6
4 7 3 8 0
3 3 2 7 1
4 5 7 0 1
1 1 9 5 2
```

1. Build a 2D Array as shown:
2. Print the **sum of** elements in the second column.
3. Print the **arithmetic mean of** elements in the first, third and fifth columns.
4. Print the **number of** elements that are greater than 4.
5. Print the **arithmetic mean of** elements in the rows whose third elements are greater than 5.
6. Print the sum of the **number of** elements that are equal to 0 or 1.

7. Hausaufgaben

Exercise 2: Python II

Auf: https://expert.ethz.ch/enrolled/SS25/mavt2/exercises

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

Abgeben bis: Monday 10.03.2023, 20:00 MEZ

NO HARDCODING

Feedback?

Zu schnell? Zu langsam? Weniger Theorie, mehr Aufgaben?
Dankbar für Feedback am besten mir direkt sagen oder Mail schreiben

Credits

Die Slide(-templates) stammen ursprünglich von Julian Lotzer und Daniel Steinhauser, vielen Dank!

→Checkt ihre Websites ab für zusätzliches Material in Informatik I, Informatik II und Stochastik & Machine Learning.

- https://n.ethz.ch/~jlotzer/
- https://n.ethz.ch/~dsteinhauser/