Programmierung mit Python

Grundlagen Grundbegriffe Grundübungen

Alfatraining Kurs 2026-09 Dozent: Karsten Flügge

Themen Heute

```
Iteratoren in for Schleifen
Match case
Generatoren: yield
Unicode
```

Wunschthemen:

- **numpy (numerische Berechnungen, Matrizen und Vectoren)**
- **django server**
- **GUI**
- **… ?**

immutable types II

**In Python sind einige grundlegende Datentypen unveränderlich (immutable).
Diese Unveränderlichkeit bedeutet, dass, einmal ein Objekt eines dieser Datentypen erstellt wurde, sein Inhalt oder sein Zustand nicht mehr geändert werden kann.**

Integers: Ganze Zahlen können nicht verändert werden. Jede Operation, die eine Zahl zu verändern scheint, erstellt tatsächlich ein neues Integer-Objekt.
Floats: Wie ganze Zahlen sind auch Gleitkommazahlen unveränderlich.
Strings: Zeichenketten sind in Python unveränderlich. Änderungen an einem String führen zur Erstellung eines **kopierten** neuen String-Objekts.
Tuples: Ein Tuple ist eine Sammlung von Python-Objekten, die, einmal erstellt, nicht mehr geändert werden kann. Versuche, ein Element eines Tuples zu ändern, führen zu einem *Typfehler*.

NEU
@dataclass(frozen=True)
Frozensets: Ein frozenset ist eine unveränderliche Version eines Sets. Es kann nicht bearbeitet werden, nachdem es erstellt wurde, was bedeutet, dass man keine Elemente hinzufügen oder entfernen kann.

```
#Erstellung und Verwendung eines frozensets
fset = frozenset([1, 2, 3, 4])
fset.add(5) # AttributeError!
```

immutable types

Immutable types werden immer (effektiv) nicht als Referenz sondern **als Wert übergeben.**
Das bedeutet, mit einer Funktion ist der **Wert einer primitiven / immutable Variable nicht modifizierbar:**

```
def try_to_change(x):
  x = x+1  # block-lokale Variable hat keinen Effekt ausserhalb der Funktion!

x = 7
try_to_change(x)
print(x) # 7

y = x
y = y+1
print(x) # 7

man kann natürlich die Variable selbst neu zuweisen:
x = x+1 # x ist nun 8, aber 7 bleibt 7 ;)
```

mutable types

Wiederholung: Attribute von Klassen Objekten können per Funktion modifiziert werden:

```
class Person:
  def __init__(self, name):
    self.name = name

def changePersonAttribute(person):
  person.name="Changed"

p = Person("Original")
changePersonAttribute(p)
print(p.name) # Changed
```

*Die **Referenz** der **Variable p**, also das Objekt auf welches p zeigt, lässt sich aber nicht verbiegen:*

```
def cantChangePerson(person):
  person=Person("NoEffect") # local variable changed in block!

cantChangePerson(p)
print(p.name) # wie vorher
```

Iteratoren

Um in for Schleifen durch alle Elemente durch zu gehen ("zu iterieren"), gibt es in Python und anderswo unter der Haube das Konzept von Iteratoren. Das verallgemeinert das Konzept von Listen…

```
for zahl in range(1, 10000000): # zum Speicher sparen wird hier KEINE Liste angelegt!
  print(zahl)
```

Im einfachsten Falle ist das ein Objekt, welches mehrmals ein nächstes Element mit "next()" zurück liefert

- `__iter__()` bereitet das Objekt darauf vor, als **Iterator** zu funktionieren.
- `__next__()` wird verwendet, um nacheinander auf die Elemente zuzugreifen.

Was oft implizit geschieht:

```
zahlen = [1, 2, 3, 4]
for zahl in zahlen:
    print(zahl)

lässt sich zum Verständnis auch explizit schreiben:
```

Iteratoren unter der Haube

Was oft implizit geschieht:

```
zahlen = [1, 2, 3, 4]
for zahl in zahlen:
   print(zahl)
```

lässt sich zum Verständnis auch explizit schreiben:

```
zahlen = [1, 2, 3, 4]
iterator = iter(zahlen)  # Erzeugung des Iterators: calls .__iter__()

def for(item,iterator,block):
try:
   while True:
      item = next(iterator)  # Nächstes Element abrufen: calls .__next__()
      print(item)
      block(item)
except StopIteration:
   print("End vom Iterator erreicht")
```

next() signalisiert das Ende der nicht mit falsy None, sondern via StopIteration exception!

So ähnlich könnte das **for** Schleifen **keyword** implementiert sein

Eigene Iteratoren

Mit dem Wissen könnten wir unsere eigene Iteratoren Schreiben:

```
# Mit dem Wissen könnten wir unsere eigene Iteratoren Schreiben:
class MyDemoIterator: # keine Vererbung notwendig, solange wir __iter__ und dann __next__ Methoden haben
  def __init__(self):
    self.data = [1, 2, 3]
    self.index = 0

  def __iter__(self):
    return self

  def __next__(self):
    if self.index >= len(self.data):
      raise StopIteration
    value = self.data[self.index]
    self.index += 1
    return value


for i in MyDemoIterator():
  print(i) # 1 2 3
```

Unendliche Iteratoren

Anders als Listen kann man mit Iteratoren auch **Werte unendlicher Mengen** zurück liefern

```
# Anders als Listen kann man mit Iteratoren auch Werte unendlicher Mengen zurück liefern
class NatuerlicheZahlen:
  def __init__(self):
    self.current = 1

  def __iter__(self):
    return self

  def __next__(self):
    self.current += 1  # inefficient ;)
    return self.current  # es hindert uns keiner daran, ewig ein next zurück zu liefern.



for n in NatuerlicheZahlen():
  print(n)
  if n > 100:
    break
```

Unendliche Primzahlen

Anders als Listen kann man mit Iteratoren auch **Werte unendlicher Mengen** zurück liefern

```
# Anders als Listen kann man mit Iteratoren auch Werte unendlicher Mengen zurück liefern
class Primes:
   def __init__(self):
      self.current = 1

   def __iter__(self):
      return self

   def __next__(self):
      self.current += 1  # inefficient ;)
      while not self.is_prime(self.current):
         self.current += 1
      return self.current

   def is_prime(self, num):
      if num < 2:
         return False
      for i in range(2, num):
         if num % i == 0:
            return False
      return True


for p in Primes():
   if p > 100:
      break
   print(p)
```

Aufgabe

Schreibe Iterator welcher alle geraden Zahlen (**unendliche Menge**) zurück liefert

```
# Anders als Listen kann man mit Iteratoren auch Werte unendlicher Mengen zurück liefern
class Even:
  def __init__(self):
    self.current =  0

  def __iter__(self):
    return self

  def __next__(self):
     erhoehe current und liefere gerade Zahl

for p in Even():
  if p > 100:
    break
  print(p)
```

Generatoren: yield

Ähnlich zu Iteratoren sind **Generatoren**.

Statt aus Klassen mit __next__() die Elemente zu holen, werden sie aber mit einem magischen **Spezialkeyword yield** generiert:

```
def simple_generator():
  yield 1 # wie ein halbes return, was danach fortgesetzt werden kann
  yield 2
  yield 3


# Usage
for value in simple_generator():
  print(value)
```

Die Logik dahinter ist etwas gewöhnungsbedürftig, da die Funktion keine normale Funktion mehr ist, sondern eine welche '**nacheinander mehrere returns**' liefert und als inneren Zustand das letzte return merkt!

Die Implementierung von `yield` in Python ist **komplex** und erfordert das Zusammenspiel mehrerer interner Mechanismen der Python-Interpreter, insbesondere des Bytecode-Interpreters und des Frame-Stacks.

Generatoren: yield

Das yield kann auch potentiell unendlich oft Werte zurück liefern:

```
def infinite_sequence():
  num = 0
  while True:
    yield num
    num += 1


# Usage
for i in infinite_sequence():
  print(i)
  if i > 5:
    break
```

Aufgabe

Liefere alle **geraden Zahlen** mit **yield**

```
def infinite_sequence():
  num = 0
  while True:
    yield … # << Todo
    num += 1


# Usage
for i in infinite_sequence():
  print(i)
  if i > 10:
    break
```

yield all primes

Das yield kann auch potentiell unendlich oft Werte zurück liefern:

```
# yield alle Primzahlen
def primes():
   non_primes = {}
   n = 2
   while True:
      if n not in non_primes:
         yield n
         non_primes[n * n] = [n]
      else:
         for p in non_primes[n]:
            non_primes.setdefault(p + n, []).append(p)
         del non_primes[n]
      n += 1
```

Aufgaben

Schreibe eigenen **Iterator** oder **Generator** für:

- alle geraden Zahlen
- alle Fakultäten (**1*2*3*4*…*n**) n!
- alle Fibonacci Zahlen
- alle **Worte** in einem Dokument
- alle **Dateien** auf Festplatte(!)
- alle Brüche (schwer)
- alle Primzahlen (schwer)
- … eigene Idee?

async await nonlocal

Keywords

Keywords of Python 3.10:

False	*await*	else	import	pass
None	break	except	in	raise
True	class	finally	is	return
and	continue	for	***lambda***	try
as	def	from	*nonlocal*	while
assert	del	global	not	**with**
async	elif	if	or	*yield*

```
async await  # parallele programmierung
finally: after except:
nonlocal
match case
with:  eigener Block (für Dateien etc)
```

Fazit: Python ist immer noch einfach, aber wird mit der Zeit zunehmend etwas komplexer (für etwas mehr Komfort)

async await

async await parallele Programmierung

Mit **async** werden Funktionen markiert, welche asynchron laufen sollen.
Riesiges Problem bei diesem Ansatz:

1) man muss wissen, ob eine Funktion async ist oder nicht.
2) async **erzwingt** andere Funktionen auch async zu sein '**function coloring**'
3) await task() # ich kann NICHT einfach auf die Funktion warten WTH

Alternativ: der **Aufrufer** sagt, dass etwas parallel laufen soll (wie in **Go**, **Thread{}** …)

match case

```
old_way = {1: 'Sehr Gut', 2: 'Gut', 3: 'Befriedigend'}
note_int = 1
print(old_way[note_int])
print(old_way.get(4, 'default case'))

# Seit python 3.10 neue keywords (switch) match und case

match note_int: # minimal hübscher als if elif else
  case 1:
    print('Sehr Gut')

  case 2:
    print('Gut')

  case 3:
    print('Befriedigend')

  case _:
    print('Keine Befriedigende Schulnote eingegeben')
```

💡 **break** nicht notwendig in python, kein 'fallthrough'

match case

```
# Seit python 3.10 neue keywords (switch) match und case

match note_int: # minimal hübscher und komfortabler als if elif else
  case 1:
    print('Sehr Gut')

  case 2:
    print('Gut')

  case 3:
    print('Befriedigend')

  case _: # default case
    print('Keine Befriedigende Schulnote eingegeben')

Vergleich zu if else:

if note_int == 1:
  print('Sehr Gut')
elif note_int == 2:
  print('Gut')
elif note_int == 3:
  print('Befriedigend')
else:
  print('Keine Befriedigende Schulnote eingegeben')
```

nonlocal

nonlocal keyword ≈ wie global aber nur für aktuelle Funktion

```
def outer():
    x = 10

    def inner():
        nonlocal x
        x = 20  # This modifies the 'x' in the outer function

    inner()
    print(x)  # Output will be 20

outer()
```

@dataclass

ORM object relational mapping

Verschiedenste module, entweder per Dekorator, oder per Vererbung

```
from dataclasses import dataclass

@dataclass
class Person:
    name: str  # eigentliche Klassenattribute, per Magie in __init__ übernommen
    age: int

p = Person("Max", 25)

print(p)            # Person(name='Max', age=25)
print(p.name)       # Max
print(p.age)        # 25
```

@dataclass

ORM object relational mapping

Verschiedenste module, entweder per Dekorator, oder per Vererbung

z.B.

```
from pydantic import BaseModel

class User(BaseModel):
    name: str
    age: int

p = Person("Max", 25)

print(p)            # Person(name='Max', age=25)
print(p.name)       # Max
print(p.age)        # 25
```

Vertiefung try except else finally

```
try:
  x = 1 / 0  # Versuch, durch Null zu teilen, was einen Fehler verursacht
  print("Dieser Code wird nur ausgeführt wenn darüber kein Fehler kommt.")
except ZeroDivisionError:
  print("Es wurde versucht, durch Null zu teilen.")
else:
  print("Dieser Block wird nur ausgeführt, wenn kein Fehler auftritt.")
finally:
  print("Dieser Block wird immer ausgeführt.")
   # nützlich zum Aufräumen
```

Floats

Floats

In Python wird eine Gleitkommazahl (**float**) standardmäßig im **IEEE 754-Format** für doppelte Genauigkeit dargestellt, das 64 Bits verwendet. Diese **64 Bits** sind aufgeteilt in:

- **1 Bit für das Vorzeichen**
- **11 Bits für den Exponenten**
- **52 Bits für die Mantisse**

Die Darstellung erfolgt nach folgendem Schema:

```
(-1) * Mantisse * 2^(Exponent - 1023)
```

Hierbei ist der Exponent im Bereich von 0 bis 2047, wobei die tatsächliche Exponentenberechnung durch Subtraktion von 1023 erfolgt, um sowohl positive als auch negative Exponenten zu ermöglichen. Die Mantisse wird als Fraktion in der Form 1.Mantisse interpretiert, wobei das führende 1 implizit ist und die restlichen 52 Bits die Mantisse direkt repräsentieren.

Unicode

ASCII

Historischer Text im Computer: ASCII

Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex
0	00 NUL	16	10 DLE	32	20	48	30 0	64	40 @	80	50 P	96	60 `	112	70 p
1	01 SOH	17	11 DC1	33	21 !	49	31 1	65	41 A	81	51 Q	97	61 a	113	71 q
2	02 STX	18	12 DC2	34	22 "	50	32 2	66	42 B	82	52 R	98	62 b	114	72 r
3	03 ETX	19	13 DC3	35	23 #	51	33 3	67	43 C	83	53 S	99	63 c	115	73 s
4	04 EOT	20	14 DC4	36	24 $	52	34 4	68	44 D	84	54 T	100	64 d	116	74 t
5	05 ENQ	21	15 NAK	37	25 %	53	35 5	69	45 E	85	55 U	101	65 e	117	75 u
6	06 ACK	22	16 SYN	38	26 &	54	36 6	70	46 F	86	56 V	102	66 f	118	76 v
7	07 BEL	23	17 ETB	39	27 '	55	37 7	71	47 G	87	57 W	103	67 g	119	77 w
8	08 BS	24	18 CAN	40	28 (	56	38 8	72	48 H	88	58 X	104	68 h	120	78 x
9	09 HT	25	19 EM	41	29)	57	39 9	73	49 I	89	59 Y	105	69 i	121	79 y
10	0A LF	26	1A SUB	42	2A *	58	3A :	74	4A J	90	5A Z	106	6A j	122	7A z
11	0B VT	27	1B ESC	43	2B +	59	3B ;	75	4B K	91	5B [	107	6B k	123	7B {
12	0C FF	28	1C FS	44	2C ,	60	3C <	76	4C L	92	5C \	108	6C l	124	7C |
13	0D CR	29	1D GS	45	2D -	61	3D =	77	4D M	93	5D]	109	6D m	125	7D }
14	0E SO	30	1E RS	46	2E .	62	3E >	78	4E N	94	5E ^	110	6E n	126	7E ~
15	0F SI	31	1F US	47	2F /	63	3F ?	79	4F O	95	5F _	111	6F o	127	7F DEL

Dann Lateinische Erweiterungen ä à

Unicode

Unicode ist der Standard zur Darstellung von **Zeichen**

In Unicode wird eine
numerierte Liste von Zeichen/Modifizierern definiert.

```
O   48      O 0x0030   DIGIT ZERO
A   65      A 0x0041   LATIN CAPITAL LETTER A
ᾰ   8112   ᾰ  0x1FB0    GREEK SMALL LETTER ALPHA WITH VRACHY
ﷺ 65018 ﷺ  0xFDFA   ARABIC LIGATURE SALLALLAHOU ALAYHE WASALLAM
😸  128568 😸  0x1F638   GRINNING CAT FACE WITH SMILING EYES
     ^^
```

Diese Nummer nennt sich **Codepoint.**

Diese Codepoints können verschieden im Computer dargestellt werden.

Unicode

Reservierte Blöcke

Basic Latin (Lower half of ISO/IEC 8859-1: ISO/IEC 646:1991-IRV aka ASCII) (0000–007F)
Latin-1 Supplement (Upper half of ISO/IEC 8859-1) (0080–00FF)
Latin Extended-A (0100–017F)
Latin Extended-B (0180–026F)
IPA Extensions (0250–02AF)
Spacing Modifier Letters (02B0–02FF)
Combining Diacritical Marks (0300–036F)
Greek and Coptic (0370–03FF)
Cyrillic (0400–04FF)
Cyrillic Supplement (0500–052F)
Armenian (0530–058F)

…

Unicode

"Planes" Bereiche Grobe Anordnung

- Plane Name Range Description
- Plane 0 0000—FFFF Basic Multilingual Plane
- Plane 1 10000—1FFFF Supplementary Multilingual Plane "SMP"
- Plane 2 20000—2FFFF Supplementary Ideographic Plane
- Plane 3 30000—3FFFF Tertiary Ideographic Plane
- Planes 4—13 40000—DFFFF unassigned
- Plane 14 E0000—EFFFF Supplementary Special-purpose Plane
- Planes 15—16 F0000—10FFFF Supplementary Private Use Area Planes

Unicode in python

- chr(unicode) -> Zeichen "**character**"
- ord(Zeichen) -> unicode codepoint nummer "**ordinal**"

in strings:

```
# Unicode-String
# Codepoints werden mit \u dargestellt
my_unicode_string = 'đ😺\u1F60' # Unicode string mit codepoint 1F60 bis 0xFFFF
# for bigger unicode codepoints use \U
my_unicode_string2 = '\U0001F60D' # Unicode string mit codepoint 1F60D bis 0x10FFFF
# Um Fehler zu vermeiden, sollte man am besten gleich \U verwenden

# True Type Font (TTF) wird benötigt, um Unicode-Zeichen darzustellen
# TTF sind intern vectorbasiert so wie SVG, d.h. sie können beliebig skaliert werden
```

? falls das System ein Zeichen nicht kennt oder der **Font** nicht geladen ist.

UTF-8 everywhere

Diese Codepoints können verschieden im Computer dargestellt werden.

Als Erweiterung des ASCII Standards hat sich UTF-8 etabliert.

Unicode **T**ransformation **F**ormat (**8** bit)
UTF-8 sollte der einzige Standard sein zum Austausch von Unicode:
Benutzt es immer und **fordert es** von Eurer Gegenpartei!
automatisch in python, selten auf Windows: , encoding = "**utf-8**"

https://utf8everywhere.org/

Interne Behandlung als UTF-8/UTF-32 je nach Anwendung.

UTF-8 everywhere

Wie erkennt man UTF-8?
"Wenn man **Ascii** Zeichen lesen kann*"
Anhand von "Heuristiken" zB in Notepad++, heisst zB:
Komplexere Zeichen fangen mit **0xC3** o.ä. an
Man kann eigentlich heute von UTF-8 ausgehen
(*Latin-9 nur noch sehr selten)

bei UTF-16 UTF-32 sind immer 0'en zwischen den Zeichen

Microsoft pseudo standard: UTF-8 BOM
(EF BB BF) Bytefolge am Anfang der Datei

UTF-8 everywhere

Beispiel:

ASCII von 00 bis 7F

"Hallo Hallö 😸" sieht im Datei System so aus:

```
00000000  48 61 6c 6c 6f 20 48 61  6c 6c c3 b6 20 f0 9f 98  |Hallo Hall.. ...|
00000010  b8                                                |.|
```

c3 b6 für "ö"
f0 9f 98 b8 "😸"

Welche Zeichen vom Betriebssystem zur Verfügung gestellt werden ist etwas Glückssache.
Heutzutage aber **"alle bis auf exotische"**

Wenn der Font ein Zeichen nicht kennt, wird es oft als Kasten oder ? dargestellt:

□

☒

Dann muss man explizit den Font dazu laden

UTF-8 everywhere

Aufpassen mit Emojis:

Jedes Jahr kommen neue hinzu, werden ggf noch nicht unterstützt.

Wenn der Font ein Zeichen nicht kennt, wird es oft als Kasten oder ? dargestellt:
□
☒

Dann muss man explizit den Font dazu laden

Zeichenketten, Zeichen, character, grapheme, glyph

Surrogates "Ersatzmittel"

Ein **Zeichen** in der realen Welt ist weniger mehrdeutig als ein 'character' in der UTF-Welt!
Andererseits kann ein neuer Begriff wie **Ikone/Graphem** nicht schaden

In den allermeisten Fällen entspricht ein **Zeichen(Buchstabe / Icon ...)** einem so genannten **codepoint:**

Hier egal: ☃ besteht aus 3 **UTF-8 Code-Einheiten** (e2 98 83 **byte** char's)
Hier egal: ☃ besteht einer UTF-16 **Code-Einheit** (short 2-byte)
Hier egal: ☃ besteht einer **UTF-32 Code-Einheit** (≈ **int**)

In der Unicode Welt ist das was wir als '**Zeichen**' verstehen ein sogenanntes
grapheme: Visuelle Einheit aus einem oder mehreren codepoints

ä=a¨ (A+Umlaut) könnte zwei Codepunkte a¨ oder ein ä sein

☀️=☀ + 'Icon**Modifier**' e2 98 80 + ef b8 8f

Viele Darstellungen können als verschiedene codepoint / byte Folgen realisiert werden!

Unicode Kontroll-Anweisungen

- Von ASCII übernommen:
- **\n** new line
- **\r** go back to beginning of line
- …

Neu
- Text Richtung (Arabisch RTL right-to-left)
- Kombination
- Darstellung (**Farbe**, als Icon, als Text)
- 🎅🎅🏻🎅🏼🎅🏽🎅🏾 👂🏻👂🏼👂🏽👂🏾👂🏿

```
🎅 9f 8e 85 f0            Weihnachtsmann
🤶 9f 8e 85 f0 9f 8f bf   Weihnachtsmann + "braun"
```

Aufgabe: gib das Ohr 👂 in python aus!

a) copy paste

b) als **\u \U** escaped sequence 4/8 **hex** nibble

c) ohne Google:

```
>>> ord("👂") # codepoint fürs Zeichen
128066
>>> hex(_)
'0x1f442' => '\U0001f442'    👂 OK
```

Endianness

Endianness:
Die Zahl 0x01020304 kann im Speicher unterschiedlich abgelegt werden:

01 02 03 04 Big Endian
04 03 02 01 Little Endian

0x00000001
01 00 00 00 Little Endian

Slicing

💡 wie str kann man in einer Liste Elemente-Bereiche (**Teil-Listen**) auslesen

```
>>> x=["a","b","c","d"]

>>> x[:2] # first 2 elements
['a', 'b']

>>> x[2:] # after first 2 elements
['c', 'd']

>>> x[2:4] # from index 2 to index 4 exklusive ⚠️
['c', 'd']

x[2:-1] # von 2 bis zum letzten Element exklusive

allgemein
liste[start:ende]   start=0 end-index exklusive

Aufgabe:
gib mir ["b","c"] via slicing von x

default ende = len(x)
```

Fortgeschrittenes Slicing

Slicing ist eine Methode in Python, um Elemente aus Sequenzen wie **Listen, Tupeln** und **Strings** herauszugreifen. Fortgeschrittenes Slicing erweitert die Grundkonzepte und bietet mehr Flexibilität beim Zugriff auf Teile einer Sequenz.

```
liste[start:end:step]
```

```
liste = [1, 2, 3, 4, 5, 6, 7, 8, 9]
jedes_zweite = liste[::2] # "Schrittlänge 2"
```

```
umgekehrt = liste[::-1] # "Schrittlänge -1 : gehe rückwärts" => als reverse
```

Kombination von Slicing-Parametern

- Kombination von Start-, End- und Schrittindizes zur präzisen Kontrolle über die Auswahl.
- Beispiel: Auswahl jedes dritten Elements in einem Teilstück der Liste:

```
liste = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
teilstueck = liste[1:8:3]
[1, 4, 7]
```

Aufgabe jetzt: mit Listen [::] slicing experimentieren
gib mir [2,4,6,8] via slicing
gib mir [2,5,8] via slicing
gib mir [8,6,4,2] via slicing # "Schrittlänge -2"
gib mir per slicing "Hallo wie gehts" => "wie"

Aufgaben Slicing

```
liste[start:end:step]
```

```
liste = [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
Aufgabe jetzt: mit Listen [::] slicing experimentieren
a) gib mir [2,4,6,8] via slicing
b) gib mir [2,5,8] via slicing
c) gib mir [8,6,4,2] via slicing  # "Schrittlänge -2"
d) gib mir per slicing "Hallo wie gehts" => "wie"
e) gib mir per slicing "Hallo wie gehts" => "Hloweghs"
f) gib mir per slicing "Hallo" => "ollaH" !
```