Python Kurs 2019/2020

3: Listen, Schleifen

Bernhard Mallinger

`b.mallinger [at] gmx.at`

https://totycro.github.io/python-kurs

Liste

- Essentielle und allgegenwärtige *Datenstruktur*
- Enthält 0 oder mehr Elemente
  → nacheinander im Speicher

```
>>> woods = ["fichte", "erle", "tanne"]
>>> len(woods)
3
>>> woods[0]
'fichte'
>>> selection = 2
>>> woods[selection]
'tanne'
>>> woods[4]
Traceback (most recent call last):
  File "<stdin>", line 1,    <module>
IndexError: list index out of range
```

Liste

Index muss nicht fix sein, kann Programmausdruck sein

```
>>> shipment_sizes
[44, 20, 90, 50, 44, 200]
>>> len(shipment_sizes)
6
>>> shipment_sizes[len(shipment_sizes) - 1]
200
```

Negative Indizes → von hinten beginnen

```
>>> shipment_sizes[-1]
200
>>> shipment_sizes[-2]
44
```

Liste

AUFGABE

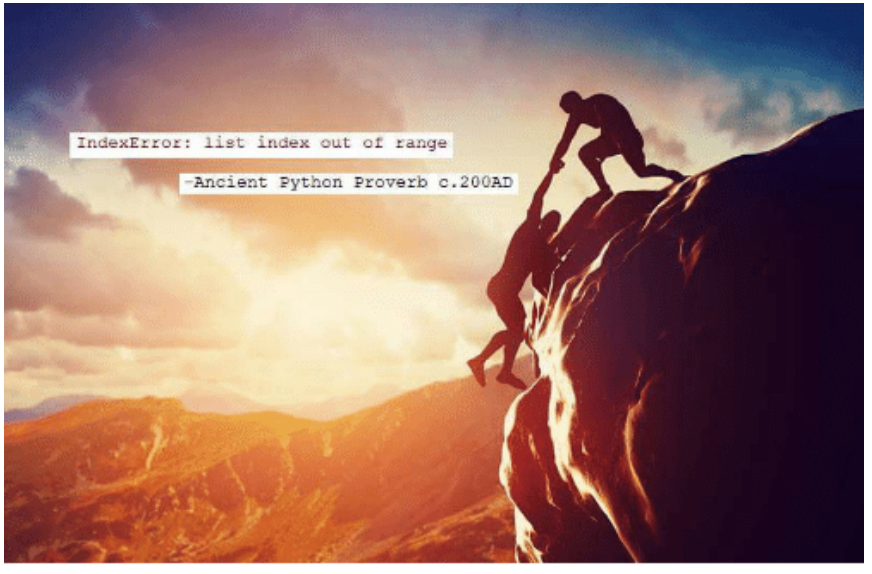
Für eine gegebene Liste `l`, schreibe Code, der das "mittlere" Element zurückgibt.

Beispiele:

`[1, 2, 3] → 2`
`[1] → 1`
`[1, 2, 3, 4] → 2 oder 3`

Achtung bei leeren Listen (`[]`)!

Liste


IndexError: list index out of range
-Ancient Python Proverb c.200AD

Liste


hiding from serial killer
Python: "omg I'm gonna die if he finds me"
Killer: "IndexError: L-"
Python: "List index out of range"

Liste

Essentiell um alles von nicht-fixer Anzahl auszudrücken
→ Dynamische Konstrukte später notwendig um damit zu arbeiten

```
>>> shipment_sizes = [44, 20, 90, 50]
>>> sum(shipment_sizes)
204
>>> new_shipments = [44, 200]
>>> shipment_sizes = shipment_sizes + new_shipments
>>> shipment_sizes
[44, 20, 90, 50, 44, 200]
>>> sum(shipment_sizes)
448
>>> max(shipment_sizes)
200
>>> min(shipment_sizes)
20
```

Liste

- Datentypen können gemischt werden
- Verarbeitung von gemischten Listen womöglich schwierig

```
>>> l = [1, 5.53, "foo"]
>>> sum(l)
Traceback (most recent call last):
  File "<stdin>", line 1,    <module>
TypeError: unsupported operand type(s)    +: 'float'     'str'
```

Liste: Slices

`l[begin:end]` gibt die Subliste zwischen den Indizes `begin` und `end` zurück

```
>>> shipment_sizes
[44, 20, 90, 50, 44, 200]
>>> shipment_sizes[2:4]
[90, 50]
>>> shipment_sizes[:3]
[44, 20, 90]
>>> shipment_sizes[3:]
[50, 44, 200]
>>> shipment_sizes[:]
[44, 20, 90, 50, 44, 200]
```

Liste: Elemente ändern

Auch Schreibzugriff mit `l[1]`

```
>>> l = [1, 2, 3]
>>> l[1]
2
>>> l[1] = 5
>>> l[1]
5
>>> l
[1, 5, 3]
```

Listen und Strings

So weit es Sinn macht haben Strings und Listen gleiche Interfaces:

```
>>> "holz"[2]
'l'
>>> len("holz")
4
>>> "holz"[2:]
'lz'
>>> "ho" + "lz"
'holz'
```

Schreibzugriff geht bei Strings (aus technischen Gründen) nicht.

Liste: Operationen

`append` zum Anfügen einzelner Elemente

```
>>> shipment_sizes
[44, 20, 90, 50, 44, 200]
>>> shipment_sizes.append(30)
>>> shipment_sizes.append(74)
>>> shipment_sizes
[44, 20, 90, 50, 44, 200, 30, 74]
```

`extend` zum Erweitern um mehrerer Elemente

```
>>> shipment_sizes = [44, 20, 90, 50, 44, 200]
>>> shipment_sizes.extend([30, 74])
>>> shipment_sizes
[44, 20, 90, 50, 44, 200, 30, 74]
```

Vgl. `[1, 2] + [3]`

Schleifen

```
shipment_sizes = [44, 20, 90]
    size    shipment_sizes:
     print(size)
```

Output:

```
44
20
90
```

Vgl.

```
size = shipment_sizes[0]
print(size)
size = shipment_sizes[1]
print(size)
size = shipment_sizes[2]
print(size)
```

Schleifen

AUFGABE

Berechne die Summe der Zahlen einer Liste (ohne `sum` zu verwenden).

Für diese und zukünftige Aufgaben: Das Ergebnis soll mit `print()` ausgegeben werden.

Schleifen

AUFGABE

Berechne, wie viele Zahlen in einer Liste größer oder gleich 3 und kleiner als 7 sind.

`[1, 3, 9, 7, 4]` → `2`
`[1]` → `0`

Erweiterung:
Berechne die Summe dieser Zahlen.

`[1, 3, 9, 7, 4]` → `7`
`[1]` → `0`

Schleifen

Die `for`-Schleife funktioniert auf "iterables", auch `str`:

```
    character    "Hello Boku!":
     print(character)
```

Output:

```
H
e
l
l
o

B
[...]
```

Schleifen

AUFGABE

Schreibe ein Programm, das mithilfe von `for` zählt, wie oft der Buchstabe "a" in einem String vorkommt.

Erweiterung:
Das Programm soll auch ausgeben, ob der Buchstabe "b" öfter als der Buchstabe "a" vorkommt.

Liste: Operationen

`in` überprüft, ob ein Element in einer Liste vorkommt:

```
>>> 2   [1, 2, 3]

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

`not in` für das Gegenteil:

```
>>> 2      [1, 2, 3]
```

Ineffiziente und nicht empfohlene Variante:

```
>>>    (2   [1, 2, 3])
```

Liste: Operationen

`l.sort()` sortiert die Liste.
`sorted(l)` gibt eine sortierte Liste zurück.

```
>>> shipment_sizes
[44, 20, 90, 50, 44, 200]
>>> sorted(shipment_sizes)
[20, 44, 44, 50, 90, 200]
>>> shipment_sizes
[44, 20, 90, 50, 44, 200]
>>> shipment_sizes.sort()
>>> shipment_sizes
[20, 44, 44, 50, 90, 200]
```

Liste: Operationen

AUFGABE

Schreibe Code, der für eine Liste von Werten den Median berechnet.

`[3, 4, 2]` → `3`
`[6.7, 5.2, 4.3, 2.2, 8.7]` → `'5.2'`

Ist die Anzahl der Elemente gerade, kann als Vereinfachung ein beliebiges dieser Elemente verwendet werden:

`[0, 1, 2, 3]` → `1` oder `2`

In der vollständigen Version soll der Durchschnitt der beiden Kandidaten ausgegeben werden:

`[0, 1, 2, 3]` → `1.5`

Liste: Operationen

`count` zählt Vorkommen:

```
>>> shipment_sizes = [44, 20, 90, 50, 44, 200]
>>> shipment_sizes.count(44)
2
>>> shipment_sizes.count(20)
1
>>> shipment_sizes.count(123)
0
```

`index` findet Position von Element (Vgl `"".find()`)

```
>>> shipment_sizes.index(50)
2
>>> shipment_sizes.index(123)
Traceback (most recent call last):
  File "<stdin>", line 1,    <module>
ValueError: 123          list
```

Liste: Operationen

Entfernen von Items auf Basis von Index (`del`):

```
>>> l = [33, 25, 48]
>>>     l[0]
>>> l
[25, 48]
```

Entfernen von Items auf Basis von Wert (`remove`):

```
>>> l = [33, 25, 48]
>>> l.remove(33)
>>> l
[25, 48]
```

Achtung: Nie aus einer Liste löschen, während man über sie iteriert!

Änderungen von Listen

Erweiterungen	
`l.append(3)`	Elemente an eine Liste anhängen
`l.extend([2, 3])`	Eine Liste um eine Liste erweitern
`l1 + l2`	2 Listen zusammenfügen

Austauschen	
`l[2] = "a"`	Einzelne Elemente überschreiben

Entfernen	
`del l[2]`	Elemente an Position löschen
`l.remove(3)`	Element entfernen

Liste: Operationen

AUFGABE

Gegeben eine Liste von Zahlen `l1`: Erzeuge eine Liste `l2`, die alle Zahlen aus `l1` enthält, die größer als 5 sind.
`l1 = [4, 7, 2, 5, 9]`
→ `l2 == [7, 9]`

AUFGABE

Gegeben 2 Listen `l` und `forbidden`: Erzeuge eine Liste `safe` mit allen Einträgen aus `l`, die nicht in `forbidden` vorkommen.
`l = ["a", "j", "i", "d", "e"]`
`forbidden = ["i", "j"]`
→ `safe == ["a", "d", "e"]`
Könnte man das auch mit Strings statt Listen implementieren?

`range` (1/2)

`range(n)` gibt ein Iterable über alle Zahlen kleiner `n` zurück.

```
    i    range(3):
     print(i)
```

Output:

```
0
1
2
```

`range` (2/2)

Man kann auch untere und obere Grenzen angeben sowie die Schrittweite.

```
# aus help(range):
range(stop) -> range object
range(start, stop[, step]) -> range object
```

`range()` gibt keine Liste zurück, kann aber iteriert werden:

```
>>> range(10)
range(0, 10)
```

Das berüchtigte FizzBuzz

Einstieg in algorithmisches Denken:

AUFGABE

Schreibe ein Programm, das alle Zahlen von 1 bis 100 ausgibt.
Anstelle von jedem Vielfachen von 3, schreibe "Fizz".
Anstelle von jedem Vielfachen von 5, schreibe "Buzz".
Bei Zahlen, die sowohl Vielfache von 3 und 5 sind, schreibe "FizzBuzz".

Gewünschte Ausgabe (gerne auch in Zeilen):

```
1 2 Fizz 4 Buzz Fizz 7 8 Fizz Buzz 11 Fizz 13 14 FizzBuzz 16 17 ...
```

Hinweis: Verwende % (modulo) um auf Vielfache zu prüfen.

Optimierung: Verwende % nur 2 Mal.

Liste: Operationen

`enumerate(l)` fügt zu den Listenelementen deren Index hinzu.

```
l = [40, 24, 31]
    index, item    enumerate(l):
    print(index, item)
```

Output:

```
0 40
1 24
2 31
```

Vgl.

```
    index    range(len(l)):
    print(index, l[index])
```

Beispiel: Anstiege in Liste

Alle "Anstiege" in einer Liste finden:

```
l = [5, 4, 7, 7, 3, 9, 4]

    index, item    enumerate(l):
    next_index = index + 1
      next_index < len(l):
        next_item = l[next_index]
          next_item > item:
            print("increase:", item, "to", next_item)
```

Output

```
increase: 4 to 7
increase: 3 to 9
```

→ Vorsicht bei Indexberechnungen

Beispiel: Anstiege in Liste (Variante)

Andere Formulierung mit dem `start`-Parameter von `range`:

```
l = [5, 4, 7, 7, 3, 9, 4]

    i    range(1, len(l)):
    previous_item = l[i - 1]
    current_item = l[i]
      previous_item < current_item:
        print("increase:", previous_item, "to", current_item)
```

Output

```
increase: 4 to 7
increase: 3 to 9
```

Was passiert bei `l = []` oder `l = [1]` oder `l = [1, 2]`?

Beispiel: Ansteige in Liste ohne Indexberechnung

Lösung mittels itertools recipe:
https://docs.python.org/3/library/itertools.html

```
     itertools        *
          (iterable):
    "s -> (s0,s1), (s1,s2), (s2, s3), ..."
    a, b = tee(iterable)
    next(b,      )
          zip(a, b)


l = [5, 4, 7, 7, 3, 9, 4]

    previous_item, current_item    pairwise(l):
      previous_item < current_item:
        print("increase:", previous_item, "to", current_item)
```

Programmiertipps

- Gib Ausdrücken, die öfter vorkommen, einen Namen:
  `next_item` ist lesbarer und ungefährlicher als `l[i + 1]`
- Achte auf Randfälle, z.B. bei Beginn und Ende von Datenstrukturen und bei leeren Listen und Strings
- Verwende möglichst sprechende Namen statt z.B. einzelne Buchstaben
  - `l` ist ein schlechter Name, `liste` auch.
  - In echtem Code am besten echte Namen verwenden, z.B. `students`, `shipments` (statt `student_list`).
- Funktionen aus Libraries sind meistens besser als eigene Implementierungen
  → möglichst guter Überblick bzw. Recherche zahlt sich aus

Übernächste Nachbarn

AUFGABE

Für eine gegebene Liste `l`, finde alle Elemente, die gleich ihrem übernächsten sind.

`l = ["apple", "banana", "apple", "pear", "banana", "pear"]`
→ "apple" ist bei Index `0` und `2`, "pear" bei `3` und `5`
→ "banana" gilt nicht, weil der Abstand zu groß ist

List comprehension

List comprehension

Wunderbar einfache Schreibweise für Datentransformationen

Statt:

```
numbers = [3, 5, 8]
squares = []

    n    numbers:
     squares.append(n ** 2)
```

Einfach:

```
numbers = [3, 5, 8]
squares = [(n ** 2)     n    numbers]
```

```
numbers = []
for i in range(100):
    numbers.append(i)
```

```
numbers = [i for i in range(100)]
```

List comprehension is always required

List comprehension

Ähnlichkeit zu Mengenschreibweise:

$$\{x^2 | x \in \mathbb{N}, 10 \leq x < 20\}$$

```
[x ** 2    x   range(10, 20)]
```

List comprehension

Auch Auswahl möglich:

```
numbers = [3, 5, 8]
squares_large = [
    n ** 2
       n    numbers
      n > 4
]
```

Vergleiche:

```
numbers = [3, 5, 8]
squares_large = []

   n    number:
      n > 4:
        squares.append(n ** 2)
```

List comprehension

Auf gute Lesbarkeit achten:

```
squares_large = [(n ** 2)    n    numbers    n > 4]
```

VS

```
squares_large = [
    n ** 2
       n    numbers
      n > 4
]
```

→ Zeilenumbrüche sind innerhalb von Klammern erlaubt, sonst mit `\` am Ende der Zeile.

List comprehension

AUFGABE

Gegeben eine Liste von Zahlen `l1`: Schreibe eine Comprehension, die alle Zahlen aus `l1` enthält, die größer als 5 sind.
`l1 = [4, 7, 2, 5, 9]`
→ `[7, 9]`

AUFGABE

Gegeben 2 Listen `l` und `forbidden`: Schreibe eine comprehension mit allen Einträgen aus `l`, die nicht in `forbidden` vorkommen.
`l = ["a", "j", "i", "d", "e"]`
`forbidden = ["i", "j"]`
→ `["a", "d", "e"]`
Könnte man das auch mit Strings statt Listen implementieren?