Builtin Datenstrukturen

Eric Niklas Wolf, Moritz Pflügner

Python-Kurs

2. November 2021

Gliederung

Exception Handling

- Alle Exceptions erben von `Exception`
- Catching mit try/except
- `else` Abschnitt wird vor `finally` ausgeführt falls keine Exception auftritt
- `finally` um Code auszuführen, der *unbedingt* laufen muss, egal ob eine Exception vorliegt oder nicht

Exception Handling - Beispiel

```
1  class MyException(Exception):
2      def __init__(self, message):
3          self.message = message
4
5      def __repr__(self):
6          return '{} mit nachricht {}'.format(self.__class__,
       self.message)
7
8  try:
9      # code
10     raise KeyError('message')
11 # mit nur einer exception
12 # except MyException as error:
13 except (KeyError, MyException) as error:
14     print(error)
15     pass
16 else:
17     # wird nur ausgefuehrt falls keine Exceptions auftreten
18     print("not gonna happen")
19 finally:
20     # was unbedingt zu tun ist
```

Boolsche Werte

- *type* ist `bool`
- Mögliche Werte: `True` oder `False`
- Operationen sind *und*, *oder*, *nicht* (`and`, `or`, `not`)

list

- enthält variable Anzahl von Objekten
- eine Liste kann beliebig viele verschiedene Datentypen enthalten (z.B. `bool` und `list`)
- Auch Listen können in Listen gespeichert werden!
- Listenobjekte haben eine feste Reihenfolge (*first in, last out*)
- optimiert für einseitige Benutzung wie z.B. Queue (`append` und `pop`)

list - Beispiel

```
1  l = [1, 9, 'string', object]
2
3  isinstance(l[0], int)  # ==> True
4  l[1] == 9  # ==> True
5  len(l)  # ==> 4
6
7  9 in l  # ==> True
8
9  l.pop()  # ==> object
10 len(l)  # ==> 3
11 l.append([])  # ==> None
12
13 l  # ==> [1, 9, 'string', []]
14 len(l)  # ==> 4
```

tuple

- Gruppiert Daten
- kann nicht mehr verändert werden, sobald es erstellt wurde
- Funktionen mit mehreren Rückgabewerten geben ein Tupel zurück

tuple - Beispiel

```
1  # Tupel mit 3 Elementen
2  t = (1, 3, 'ein string')
3  isinstance(t, tuple) # ==> True
4
5  t[0] == 1   # ==> True
6  t[1] == 3   # ==> True
7  t[2] == 'ein string'   # ==> True
8  t[4] == 'ein string'   # ==> IndexError: tuple index out of
       range
9  t[2] = 'ein anderer string'   # ==> TypeError: 'tuple' object
        does not support item assignment
10
11 # oder auch ohne klammern
12 t = 1, 3, 'ein string'
13 # macht es (manchmal) besser lesbar, z.b. bei
14 return 1, 2, 5
```

dict

- einfache Hashmap
- ungeordnet
- jeder hashbare Typ kann ein Key sein
- jedem Key ist dann ein Value zugeordnet

dict - Beispiel

```
1 d = { 'm': 4, 3: 'val2', object: 'auch typen koennen keys
      sein' }
2
3 d[3]   # ==> "val2"
4 d['q']   # ==> KeyError: 'q'
5 d.get('q')   # ==> None
6 d.get('q', 5)   # ==> 5
7
8 d[0] = 7
9 d   # ==> {3: 'val2', 'm': 4, 'q': 5, <class 'object'>: 'auch
        typen koennen keys sein'}
```

dict - Beispiel

```
1  d.setdefault('m')  # ==> 4
2  d.setdefault('q', 5)  # ==> 5
3  d  # ==> { 'm': 4, 3: 'val2', object: 'auch typen koennen
       keys sein', 0:7, 'q': 5 }
4  len(d)  # ==> 5
5  d.keys()  # ==> dict_keys([3, 0, 'm', 'q', <class 'object
       '>])
6  d.values()  # ==> dict_values(['val2', 7, 4, 5, 'auch typen
       koennen keys sein'])
7  d.items()  # ==> dict_items([(3, 'val2'), (0, 7), ('m', 4),
       ('q', 5), (<class 'object'>, 'auch typen koennen keys
       sein')])
8
9  'm' in d  # ==> True
10 object in d  # ==> True
11 tuple in d  # ==> False
```

set/frozenset

- kann nur hashbare Einträge enthalten
- `set` selbst ist nicht hashbar
- `frozensets` sind hashbar, jedoch nicht mehr veränderbar
- enthält jedes Element nur einmal
- schnellere Überprüfung mit `in` (prüft, ob Element enthalten ist)
- Mögliche Operationen: `superset()`, `subset()`, `isdisjoint()`, `difference()`, `<`, `>`, `disjoint()`, `-`
- ungeordnet
- (frozen)sets können frozensets enthalten (da sie einen festen Hashwert haben)

set/frozenset - Beispiel

```
1  s1 = {1, 2, 'string', object, ('ein', 'tuple')}
2
3  2 in s1   # ==> True
4  'ein' in s1   # ==> False
5  ('ein', 'tuple') in s1   # ==> True
6  set(('ein', 'tuple'))   # ==> {'ein', 'tuple'}
7
8  s2 = {'anderes', 'set'}
9  s1 > s2   # ==> False
10 s1.isdisjoint(s2)   # ==> True
11
12 s1.add('anderes')
13 s1 | s2   # ==> {1, 2, 'string', object, ('ein', 'tuple'), '
       set'}
14 s1 & s2   # ==> {'anderes'}
15 s2 - s1   # ==> {'set'}
16
17 s2 = frozenset(s2)
18 s1.add(s2)
19 s2.add(5)   # ==> AttributeError: 'frozenset' object has no
       attribute 'add'
```

Unpacking

- einfaches Auflösen von Listen und Tupeln in einzelne Variablen
- nützlich in `for`-Schleifen

Unpacking - Beispiel

```
# unpacking (geht auch mit listen)
t = 1, 3, 'ein string'   # tuple ohne klammern gebaut

x, y, z = t
x is t[0]   # ==> True
y is t[1]   # ==> True

# oder
x, *y = t
x   # ==> 1
y   # ==> [3, 'ein string']

a, b, c = 1, 2, 4

d, e, f, *g = [3, 0, 8, 7, 46, 42]
f   # ==> 8
g   # ==> [7, 46, 42]
```

Iteratoren

- alles mit einer `__next__` Methode ist ein *Iterator*
- Iteratoren stellen eine folge von Elementen dar, aus welcher man mit `next(iterator)` das nächste Element holen kann
- wenn der Iterator erschöpft ist wird eine `StopIteration` Exception ausgelöst
- gehören zu den *Iterables*

Wichtig: Iteratoren besitzen einen internen Zustand und sollten deswegen nicht von mehreren Benutzern gleichzeitig benutzt werden! Dies kann mit der Verwendung mehrerer unabhängiger Iteratoren umgangen werden.

Iterables

- alles mit einer `__iter__` Methode ist eine *Iterable*
- *Iterables* liefern mit `iter(obj)` einen *Iterator* über sich selbst
- `for` Schleifen ermöglichen ein einfaches Durchlaufen
- eine *Iterable* über Integer ist
  `range([start], stop, step=1)`
- um Iterables zu kombinieren kann man
  `zip(iterable_1, iterable_2, ..., iterable_n)`
  verwenden
- `any(iterable)` prüft ob mindestens ein Element einer *Iterable* wahr ist
- `all(iterable)` prüft ob alle Elemente einer *Iterable* wahr sind

Wichtig: Während des Iterierens können einige Iterables nicht verändert werden

Iteration - Beispiele

```
for i in [1,2,3]:
    if i > 9:
        break
        # verlaesst die Schleife
else:
    # wenn kein break vorkommt
    print("keine Zahl groesser als 9")

# Iterator ueber die Zahlen von 0 bis 3 (3 nicht enthalten)
iterator = iter(range(0, 3))

# gibt 0, 1 und 2 aus
for i in iterator:
    print(i)

# kein Output, Iterator ist erschoepft
for i in iterator:
    print(i)
```

Iteration - Beispiele

```
# gibt 1, 'value1' und 2, 'value2' aus
for i in {1:'value1', 2:'value2'}.items():
    # i ist ein Tuple von (key, value)
    print(i[0], i[1])

# iteration mit tuple unpacking
# gleiche Ausgabe wie letztes Beispiel
for key, value in {1:'value1', 2:'value2'}.items():
    print(key, value)

# zip verbindet zwei Iterables und stoppt wenn einer die
    Elemente ausgehen
# -4 wird deswegen nicht ausgegeben
for value1, value2 in zip((1,2,3), (-1,-2,-3,-4)):
    print(value1, value2)

# True != False
any((False, True, False)) != all((False, True, False))
```

Context Manager

- Aufruf mit `with`
- kann jedes Objekt sein, welches eine `__enter__` und `__exit__` Methode hat
- praktisch beim Arbeiten mit Dateien

Context Manager

```
class MyManager:
    def __enter__(self):
        # tue dinge
        return self

    def __exit__(self, type, value, traceback):
        # schliesse handler etc ...
        pass

    def do_things(self):
        # ...
        pass

with MyManager() as m:
    m.do_things()
```

Files

- Dateien werden mit `open(filename, mode="r")` geöffnet
- `r` steht für Lesezugriff, `w` für Schreibzugriff
- `+` erlaubt zusätzlich zum Modus Lesen/Schreiben, `x` schlägt fehl wenn die Datei bereits existiert
- die zurückgegebenen Objekte sind *Iteratoren* über ihre Zeilen und *Context Manager*
- können über `.read(size=-1)` und `.write(text)` gelesen und geschrieben werden

Wichtig: Dateien müssen wieder mit `.close()` geschlossen werden
Beachte: Wird eine Datei mit `w` geöffnet, wird sie geleert! Nutze stattdessen `r+`.

Files - Beispiel

```
# gibt alle Zeilen einer Datei aus
with open("test1.txt", mode='r') as f:
    for line in f:
        print(line)

# schreibe einen String und lese ihn wieder
with open("test2.txt", mode='w+') as f:
    # oder print("Test", file=f)
    f.write("Test\n")
    # seek aendert die Position innerhalb der Datei
    # und gibt ihren neuen Wert zurueck
    f.seek(0) == 0
    # tell gibt diese zurueck ohne sie zu aendern
    f.tell() == 0
    print(f.read())
```

Files - Beispiel

```
# oeffnet eine Datei falls sie nicht existiert
try:
    f = open("test.txt", "x")
except FileExistsError:
    print("Datei existiert bereits")
else:
    with f:
        f.write("Erster!")

# Nutzung ohne Context Manager
f = open("test.txt")
try:
    # nutze die Datei
    pass
finally:
    f.close()
```