

Builtin Datenstrukturen

Dominik Pataky

2017-10-30

Python-Kurs 2017

1. Exceptions
2. Booleans
3. Lists
4. Tuples
5. Dicts

6. Set/Frozenset
7. Iterations
8. Unpacking
9. File Handling
10. Context Manager

Exceptions

- Alle Exceptions erben von **Exception**
- Catching mit try/except
- **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
7         .message)
8
9 try:
10     # code
11     raise KeyError('message')
12 # mit nur einer exception
13 # except MyException as error:
14 except (KeyError, MyException) as error:
15     print(error)
16     pass
17 finally:
18     # was unbedingt zu tun ist
```

Booleans

Boolsche Werte

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

Lists

- 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
```

Tuples

- 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
```

Dicts

- 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'
```

Iterations

- nur `foreach`
- für Iterationen über Integer gibt es
`range([start], stop, step=1)`
- um Iteratoren zu kombinieren kann man
`zip(iterator_1, iterator_2, ..., iterator_n)`
verwenden
- alles mit einer `__iter__` Methode ist iterierbar
- `iter(iterable)` konstruiert einen *stateful iterator*

Iteration - Beispiel

```
1 for i in [1,2,3]:
2     if i > 9:
3         break
4     # code
5     else:
6         pass
7         # wenn kein break vorkommt
8
9 for i in (1,2,3):
10    # code
11    pass
12
13 for i in {1:'value1', 2:'value2'}:
14    # iteration ueber die keys
15    pass
```

Iteration - Beispiel

```
1 for i in {1:'value1', 2:'value2'}.items():
2     # i ist tuple von (key, value)
3     pass
4
5 for value1, value2 in [
6     (1, 'werner'),
7     (3, 'geh mal in den keller'),
8     (42, 'ich glaub die russen komm\')]
9     ]:
10    # iteration mit tuple unpacking
11    # code
12
13 # oder auch
14
15 for value1, value2 in zip([1,3,42], ['werner', 'geh mal in den
16     keller', 'ich glaub die russen komm\'], 'dieser string wird
17     in der iteration nicht auftauchen', 'dieser auch nicht'])
18
19 for key, value in {1:'value1', 2:'value2'}.items():
20     # iteration ueber keys und values mit tuple unpacking
21     pass
```


Unpacking

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

Unpacking - Beispiel

```
1 # unpacking (geht auch mit listen)
2 t = 1, 3, 'ein string' # tuple ohne klammern gebaut
3
4 x, y, z = t
5 x is t[0] # ==> True
6 y is t[1] # ==> True
7
8 # oder
9 x, *y = t
10 x # ==> 1
11 y # ==> [3, 'ein string']
12
13 a, b, c = 1, 2, 4
14
15 d, e, f, *g = [3, 0, 8, 7, 46, 42]
16 f # ==> 8
17 g # ==> [7, 46, 42]
```

File Handling

- Dateien können mit `open(filename, mode="r")` geöffnet werden
- *File Handler* sind Iteratoren über die Zeilen einer Datei
- **Wichtig:** File Handler müssen auch wieder geschlossen werden
- **r** steht für Lesezugriff, **w** für Schreibzugriff

Beachte: Wird eine Datei mit Schreibzugriff geöffnet, wird sie geleert!
Also wichtige Inhalte vorher auslesen.

File Handling - Beispiel

```
1 with open(myfile, mode='r') as f:
2     for line in f:
3         # code
4
5
6 with open(myfile, mode='w+') as f:
7     for line in document:
8         f.write(line)
9         # oder
10        print(line, file=f)
11
12 f = open(myfile)
13
14 # code
15
16 f.close()
```

Context Manager

- Aufruf mit `with`
- kann jedes Objekt sein, welches eine `__enter__` und `__exit__` Methode hat
- praktisch beim *File Handling*

Context Manager

```
1 class MyManager:
2     def __enter__(self):
3         # tue dinge
4         pass
5
6     def __exit__(self):
7         # schliesse handler etc ...
8         pass
9
10    def do_things(self):
11        # ...
12        pass
13
14 with MyManager() as m:
15     m.do_things()
```