Informatik I: Einführung in die Programmierung

14. Ausnahmen, Generatoren und Iteratoren, Backtracking

Albert-Ludwigs-Universität Freiburg

Peter Thiemann

22.01.2019

1 Prolog: Ausnahmen (Exceptions)

- Ausnahmen
- `try-except`
- `try-except-else`-Blöcke
- `finally`-Blöcke
- `raise`-Anweisung

Prolog: Ausnahmen (Exceptions)
Ausnahmen
try-except
try-except-else-Blöcke
finally-Blöcke
raise-Anweisung
Generatoren
Iteratoren
Dateien
Zugabe: Sudoku
Zusammen-fassung

Ausnahmen (1)

- In vielen Beispielen sind uns *Tracebacks* wie der folgende begegnet:

Python-Interpreter

```
>>> print({"spam": "egg"}["parrot"])
Traceback (most recent call last): ...
KeyError: 'parrot'
```

- Solche Fehler heissen Ausnahmen (*exceptions*).
- Jetzt wollen wir solche Fehler abfangen und selbst melden.

Ausnahmen (2)

- Ausnahmen haben zwei Anwendungen
  1. Signalisieren einer Situation, die im Programm nicht vorgesehen ist. Meist im Zusammenhang mit externen Ereignissen.
     Beispiel: physikalischer Fehler beim Lesen einer Datei, mangelnder Speicherplatz, etc
  2. Vereinfachte Behandlung des “Normalfalls” einer Funktion. Die Ausnahme wird dabei als alternativer Rückgabewert verwendet.
- Das Auslösen einer Ausnahme bricht den normalen Programmablauf ab. Die Anweisungen, die normalerweise den Kontrollfluss steuern (**`if`**, **`for`**, **`while`**, **`return`**, `...`), werden ignoriert. Stattdessen wird die Ausnahme solange hochgereicht, bis sich ein Block findet, der die Ausnahme bearbeitet.
- Zur Ausnahmebehandlung dienen in Python die Anweisungen `raise` und `try` mit Optionen `except`, `finally` und `else`.

Prolog: Ausnahmen (Exceptions)
Ausnahmen
try-except
try-except-else-Blöcke
finally-Blöcke
raise-Anweisung
Generatoren
Iteratoren
Dateien
Zugabe: Sudoku
Zusammenfassung

try-except

- Eine try-except-Anweisung kann Ausnahmen behandeln, die während der Ausführung des try-Blocks auftreten.

```
try:
    call_critical_code()
except NameError as e:
    print("Sieh mal einer an:", e)
except KeyError:
    print("Oops! Ein KeyError!")
except (IOError, OSError):
    print("Na sowas!")
except:
    print("Ich verschwinde lieber!")
    raise
```

except-Blöcke (1)

`except XYError as e`

Ein solcher Block wird ausgeführt, wenn innerhalb des `try`-Blocks eine Ausnahme `XYError` auftritt. Die Variable `e` enthält die Ausnahme.

`except XYError`

Ohne Variable, wenn die Ausnahme nicht im Detail interessiert.

`except (XYError, YZError) as e`

Ein Tupel fängt mehrere Ausnahmetypen gemeinsam ab.

`except` — unspezifisch

Ohne weitere Angaben werden alle Ausnahmen abgefangen. Vorsicht: Auch CTRL-C-Ausnahmen! Besser den Ausnahmetyp `Exception` verwenden.

Prolog: Ausnahmen (Exceptions)
Ausnahmen
try-except
try-except-else-Blöcke
finally-Blöcke
raise-Anweisung
Generatoren
Iteratoren
Dateien
Zugabe: Sudoku
Zusammenfassung

except-Blöcke (2)

Prolog: Ausnahmen (Exceptions)
Ausnahmen
try-except
try-except-else-Blöcke
finally-Blöcke
raise-Anweisung
Generatoren
Iteratoren
Dateien
Zugabe: Sudoku
Zusammenfassung

- `except`-Blöcke werden der Reihe nach abgearbeitet, bis der erste passende Block gefunden wird (falls überhaupt einer passt).
- Unspezifische `except`-Blöcke sind nur als letzter Test sinnvoll.
- Die Ausnahme kann mit einer `raise`-Anweisung ohne Argument weitergereicht werden.

try – except – else

- Ein try-except-Block kann mit einem else-Block abgeschlossen werden, der ausgeführt wird, falls im try-Block keine Ausnahme ausgelöst wurde:

```
try:
    call_critical_code()
except IOError:
    print("IOError!")
else:
    print("Keine Ausnahme")
```

Prolog: Ausnahmen (Exceptions)
Ausnahmen
try-except
try-except-else-Blöcke
finally-Blöcke
raise-Anweisung
Generatoren
Iteratoren
Dateien
Zugabe: Sudoku
Zusammenfassung

finally-Blöcke

- Wenn eine Ausnahme nicht behandelt werden kann, müssen trotzdem oft Ressourcen freigegeben werden — etwa um Netzwerkverbindungen zu schließen.
- Dazu dient der finally-Block:

```
try:
    call_critical_code()
finally:
    print("Das letzte Wort habe ich!")
```

- Der `finally`-Block wird *immer* beim Verlassen des `try`-Blocks ausgeführt, egal ob Ausnahmen auftreten oder nicht. Auch bei einem `return` im `try`-Block wird der `finally`-Block vor Rückgabe des Resultats ausgeführt.
- Wurde eine Ausnahme signalisiert, wird sie nach Behandlung des `finally`-Blocks weitergegeben.

Beispiel

kaboom.py

```
def kaboom(x, y):
    print(x + y)

def tryout():
    kaboom("abc", [1, 2])

try:
    tryout()
except TypeError as e:
    print("Hello world", e)
else:
    print("All OK")
finally:
    print("Cleaning up")
print("Resuming ...")
```


Prolog: Ausnahmen (Exceptions)
Ausnahmen
try-except
try-except-else-Blöcke
finally-Blöcke
raise-Anweisung
Generatoren
Iteratoren
Dateien
Zugabe: Sudoku
Zusammenfassung

Die raise-Anweisung

Prolog: Ausnahmen (Exceptions)
Ausnahmen
try-except
try-except-else-Blöcke
finally-Blöcke
raise-Anweisung
Generatoren
Iteratoren
Dateien
Zugabe: Sudoku
Zusammenfassung

- Die `raise`-Anweisung signalisiert eine Ausnahme.
- **`raise`** hat als optionales Argument ein `Exception` Objekt.
- Beispiele

```
raise KeyError("Fehlerbeschreibung")
raise KeyError()
raise KeyError
```

.

- `raise` ohne Argument dient zum Weiterreichen einer Ausnahme in einem **`except`**-Block.

2 Generatoren

Prolog: Ausnahmen (Exceptions)

Generatoren

Iteratoren

Dateien

Zugabe: Sudoku

Zusammen-fassung

Das Geheimnis von `range` & Co

Python-Interpreter

```
>>> for i in range(3): print(i)
...
0
1
2
>>> rng = range(3)
>>> rng
range(0, 3)
>>> for i in rng: print(i)
...
0
1
2
```

Prolog: Ausnahmen (Exceptions)
Generatoren
Iteratoren
Dateien
Zugabe: Sudoku
Zusammenfassung

Beobachtungen

- `range(3)` liefert keine Liste, sondern ein spezielles Objekt
- Dieses Objekt kann durch `for` zum “Durchlaufen” einer Sequenz gebracht werden.
- Dieses Verhalten ist in Python eingebaut, aber es kann auch selbst programmiert werden.
- Dafür gibt es mehrere Möglichkeiten u.a.
  - Generatoren
  - Iteratoren

Ein Generator für `range`

Prolog: Ausnahmen (Exceptions)
Generatoren
Iteratoren
Dateien
Zugabe: Sudoku
Zusammenfassung

```
def myRange(n):
    """ generator to count from 0 to n-1 """
    i = 0
    while i<n:
        yield i
        i = i+1
```

- Neue Anweisung: `yield i`
- Das Vorkommen von `yield` bewirkt, dass der Funktionsaufruf `myRange(3)` als Ergebnis einen **Generator** liefert.
- Ein Generator kann durch Methoden oder durch `for` gesteuert werden.

Verwendung von Generatoren

Erster Aufruf: `next(gen)`

1. Starte den Rumpf des Generators (Bsp: Funktionsrumpf von `myRange`)
2. Führe aus bis zum ersten `yield`
3. Speichere den aktuellen Stand der Ausführung (Belegung der lokalen Variablen und Parameter, sowie die nächste Anweisung) im Generator
4. Liefere das Argument von `yield` als Ergebnis.

Nachfolgende Aufrufe: `next(gen)`

1. Restauriere den gespeicherten Stand der Ausführung beim zuletzt ausgeführten `yield`
2. Führe aus bis zum nächsten `yield`, dann weiter wie Nr. 3 oben.
3. Falls Ende des Rumpfs ohne `yield` erreicht: Ausnahme `StopIteration` wird ausgelöst.

Prolog: Ausnahmen (Exceptions)
Generatoren
Iteratoren
Dateien
Zugabe: Sudoku
Zusammen-fassung

Python-Interpreter

```
>>> mr = myRange(2)
>>> next(mr)
0
>>> next(mr)
1
>>> next(mr)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Python-Interpreter

```
>>> mr = myRange(2)
>>> list(mr)
[0, 1]
>>> list(mr)
[]
```

Prolog: Ausnahmen (Exceptions)
Generatoren
Iteratoren
Dateien
Zugabe: Sudoku
Zusammenfassung

Ein Generator muss nicht endlich sein

```
def upFrom(n):
    while True:
        yield n
        n = n + 1
```

Python-Interpreter

```
>>> uf = upFrom(10)
>>> next(uf)
10
>>> next(uf)
11
>>> list(uf)
^CTraceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<stdin>", line 3, in upFrom
```


Prolog: Ausnahmen (Exceptions)

Generatoren

Iteratoren

Dateien

Zugabe: Sudoku

Zusammenfassung

Abfragen eines (potentiell) unendlichen Generators

Zu Fuß mit Ausnahmen

```
def printGen(gen):
    try:
        while True:
            v = next(gen)
            print(v)
    except StopIteration:
        pass
```

Besser: mit `for`-Schleife

```
def printGenBetter(gen):
    for v in gen:
        print(v)
```

Prolog: Ausnahmen (Exceptions)
Generatoren
Iteratoren
Dateien
Zugabe: Sudoku
Zusammen-fassung

Zwei weitere Beispiele: map und filter

Prolog: Ausnahmen (Exceptions)
Generatoren
Iteratoren
Dateien
Zugabe: Sudoku
Zusammenfassung

```
def myMap (f, seq):
    for x in seq:
        yield f (x)


def twox1 (x):
    return 2*x+1

printGenBetter(
    myMap(twox1, upFrom(10)))
```

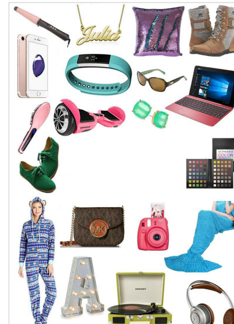
Was wird gedruckt?

```
def myFilter (p, seq):
    for x in seq:
        if p(x):
            yield x

def div3 (x):
    return x % 3 == 0

printGenBetter(
    myFilter(div3, upFrom(0)))
```

Was wird gedruckt?

Ein Problem

Nanga Elmkombo wird seine Schwester in Kamerun besuchen. Sein Koffer darf 23kg wiegen, die er mit Geschenken komplett ausnutzen will.



Prolog: Ausnahmen (Exceptions)
Generatoren
Iteratoren
Dateien
Zugabe: Sudoku
Zusammenfassung

Sublisten

Definition: Subliste

Sei $L = [x_1, \ldots, x_n]$ eine Liste. Eine Subliste von L hat die Form $[x_{i_1}, \ldots, x_{i_k}]$ und ist gegeben durch eine Folge von Indizes $i_1 < i_2 < \cdots < i_k$ mit $i_j \in \{1, \ldots, n\}$.

Beispiel: Sublisten von $L = [1, 5, 5, 2, 1, 7]$

$L_1 = [1, 5, 5, 2, 1, 7]$ $L_2 = [1, 5, 1, 7]$ $L_3 = [5, 5]$

$L_4 = [1, 2]$ $L_5 = [2, 1]$ $L_6 = []$

keine Sublisten von L:

$[1, 2, 8]$ $[2, 5, 1]$

Fakt

Es gibt 2^n Sublisten von $L = [x_1, \ldots, x_n]$, wenn alle x_i unterschiedlich.

Prolog: Ausnahmen (Exceptions)

Generatoren

Iteratoren

Dateien

Zugabe: Sudoku

Zusammenfassung

UNI FREIBURG

Ein spezielles 0/1 Rucksackproblem

Gegeben ist eine Liste L von n ganzen Zahlen (Gewichten). Gibt es eine Subliste von L, deren Summe exakt S (Zielgewicht) ergibt?

Ein schweres Problem

- Es ist nicht bekannt, ob es dafür einen effizienten Algoritmus gibt.
- Der naive Algorithmus probiert alle maximal möglichen 2^n Sublisten durch.

Unser Algorithmus verwendet ein Dictionary

```
gifts = {'phone': 200, 'boots': 1200, 'laptop': 2200, 'glasses': 50,
         'camera': 150, 'jumpsuit': 2340, 'headphones': 80, 'fitbit': 40,
         'hanger': 10, 'pillow': 400, 'hoverboard': 870, 'handbag': 430}
```


Prolog: Ausnahmen (Exceptions)

Generatoren

Iteratoren

Dateien

Zugabe: Sudoku

Zusammen-fassung

Ein rekursiver Algorithmus mit Generatoren

Prolog: Ausnahmen (Exceptions)
Generatoren
Iteratoren
Dateien
Zugabe: Sudoku
Zusammenfassung

```
def knapsack (goal : int, items : dict):
    if goal == 0:
        yield []                            # solution found
    elif not items:
        return                              # out of items, no solution
    else:
        item, weight = items.popitem()
        yield from knapsack (goal, items) # solutions without item
        if weight <= goal:
            for solution in knapsack (goal - weight, items):
                yield [item] + solution
        items[item] = weight                # put back item
```

Neu und wichtig

- Wird der Rumpf eines Generators mit `return` beendet, löst der Generator eine `StopIteration` Ausnahme aus.
- `dict.popitem()` entfernt einen beliebigen Schlüssel aus `dict` und liefert das Paar aus Schlüssel und zugehörigem Wert.
- `yield from gen` entspricht

```
for x in gen:
    yield x
```

- Der Algorithmus verwendet Backtracking:
  - Ein Lösungsansatz wird Schritt für Schritt zusammengesetzt.
  - Erweist sich ein Ansatz als falsch, so werden Schritte zurückgenommen (Backtracking) bis ein alternativer Schritt möglich ist.
- Mit rekursiven Generatoren ist Rücknahme von Schritten besonders einfach.

Prolog: Ausnahmen (Exceptions)
Generatoren
Iteratoren
Dateien
Zugabe: Sudoku
Zusammenfassung

Muster zur Verarbeitung von Dictionaries

Iterativ

```
def work_it (d : dict):
    # initialization
    while dict:
        key, val = d.popitem()
        # process association (key, val)
    return
```

Rekursiv

```
def work_rec (d : dict):
    if not dict:
        return # base case for empty dict
    else:
        key, val = d.popitem()
        # process association
        res = work_rec (d)
        # postprocess result using key, val, res
```


Prolog: Ausnahmen (Exceptions)
Generatoren
Iteratoren
Dateien
Zugabe: Sudoku
Zusammen-fassung

3 Iteratoren

Prolog: Ausnahmen (Exceptions)

Generatoren

Iteratoren

Dateien

Zugabe: Sudoku

Zusammenfassung

Iterierbare Objekte

- Generalisierung von Generatoren
- Die `for`-Schleife kann für viele Container-Objekte die Elemente durchlaufen.
- Dazu gehören Sequenzen, Tupel, Listen, Strings, `dicts`, Mengen usw:

Python-Interpreter

```
>>> for el in set((1, 5, 3, 0)): print(el, end=' ')
...
0 1 3 5
```

Prolog: Ausnahmen (Exceptions)
Generatoren
Iteratoren
Dateien
Zugabe: Sudoku
Zusammenfassung

UNI FREIBURG

Das Iterator-Protokoll

UNI FREIBURG

- Wir können ein Objekt iterierbar machen, indem wir das Iterator-Protokoll implementieren.
- Dafür muss die magische Methode `__iter__` definiert werden, die einen Iterator zurückliefert.
- Jeder Iterator implementiert die magische Methode `__next__`, die das nächste Element liefert. Gibt es kein weiteres Element, so muss die Ausnahme StopIteration ausgelöst werden.
- Die Funktion `iter(`*object*`)` ruft die `__iter__`-Methode auf.
- Die Funktion `next(`*object*`)` ruft die `__next__`-Methode auf.

Prolog: Ausnahmen (Exceptions)

Generatoren

Iteratoren

Dateien

Zugabe: Sudoku

Zusammenfassung

Implementierung der `for`-Schleife

Prolog: Ausnahmen (Exceptions)
Generatoren
Iteratoren
Dateien
Zugabe: Sudoku
Zusammenfassung

for

```
for el in seq:
    do_something(el)
```

wird intern wie die folgende `while`-Schleife ausgeführt

iterator

```
iterator = iter(seq)
try:
    while True:
        el = next(iterator)
        do_something(el)
except StopIteration:
    pass
```

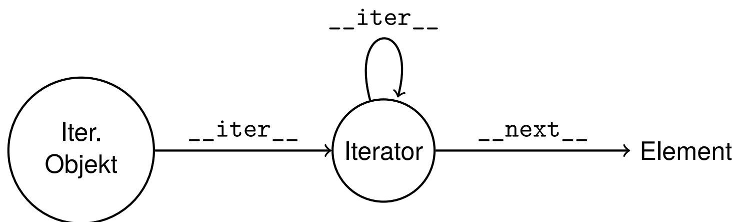
Das Iterator-Protokoll bei der Arbeit

Python-Interpreter

```
>>> seq = ['Crackpot', 'Religion']
>>> iter_seq = iter(seq)
>>> iter_seq
<list_iterator object at 0x1094d8610>
>>> print(next(iter_seq))
Crackpot
>>> print(next(iter_seq))
Religion
>>> print(next(iter_seq))
Traceback (most recent call last): ...
StopIteration
```

Iterierbare Objekte vs. Iteratoren (1)

Prolog: Ausnahmen (Exceptions)
Generatoren
Iteratoren
Dateien
Zugabe: Sudoku
Zusammen-fassung



- Ein iterierbares Objekt erzeugt bei jedem Aufruf von `iter()` einen neuen Iterator, besitzt aber selbst keine `__next__`-Methode.
- Ein Iterator dagegen liefert sich selbst beim Aufruf von `iter()`, aber jeder Aufruf von `next()` ergibt ein neues Objekt aus dem Container.
- Da Iteratoren auch die `__iter__`-Methode besitzen, können Iteratoren an allen Stellen stehen, an denen ein iterierbares Objekt stehen kann (z.B. `for`-Schleife).

Iterierbare Objekte vs. Iteratoren (2)

- EIn Iterator (z.B. der Generator `myMap`) kann dort stehen, wo ein iterierbares Objekt (z.B. eine Liste) stehen kann, aber es passiert etwas anderes!
- Iteratoren sind nach einem Durchlauf, der mit `StopIteration` abgeschlossen wurde, erschöpft, wie im nächsten Beispiel:

Python-Interpreter

```
>>> iterator = myMap(twox1, range(2))
>>> for x in iterator:
...     for y in iterator:
...         print(x,y)
...
1 3
>>>
```

Prolog: Ausnahmen (Exceptions)
Generatoren
Iteratoren
Dateien
Zugabe: Sudoku
Zusammenfassung

Iterierbare Objekte vs. Iteratoren (3)

- Wird bei jedem Start eines Schleifendurchlaufs ein neuer Iterator erzeugt, läuft alles wie erwartet:

Python-Interpreter

```
>>> for x in myMap(twox1, range(2)):
...     for y in myMap(twox1, range(2)):
...         print(x,y)
...
1 1
1 3
3 1
3 3
>>>
```

Prolog: Ausnahmen (Exceptions)
Generatoren
Iteratoren
Dateien
Zugabe: Sudoku
Zusammenfassung

Weitere iterierbare Objekte

- Die range-Funktion liefert ein range-Objekt, das iterierbar ist.
- D.h. das Objekt liefert bei jedem iter()-Aufruf einen neuen Iterator.

Python-Interpreter

```
>>> range_obj = range(10)
>>> range_obj
range(0, 10)
>>> range_iter = iter(range_obj)
>>> range_iter
<range_iterator object at 0x108b10e70>
```

Direkte Verwendung des Iterator-Protokolls: myZip

Erinnerung:

Python-Interpreter

```
>>> zz = zip(range(20), range(0,20,3)); zz
<zip object at 0x10340e908>
>>> list(zz)
[(0, 0), (1, 3), (2, 6), (3, 9), (4, 12), (5, 15), (6, 18)]
```

- Muss explizit das Iterator Interface verwenden, da zwei Eingaben unabhängig voneinander iteriert werden müssen.

Implementierung: myZip

Prolog: Ausnahmen (Exceptions)

Generatoren

Iteratoren

Dateien

Zugabe: Sudoku

Zusammenfassung

```
def myZip (s1, s2):
    i1 = iter(s1)
    i2 = iter(s2)
    try:
        while True:
            e1 = next(i1)
            e2 = next(i2)
            yield (e1, e2)
    except StopIteration:
        pass
```

Iteratoren – selbst gestrickt

UNI FREIBURG

Prolog: Ausnahmen (Exceptions)
Generatoren
Iteratoren
Dateien
Zugabe: Sudoku
Zusammen-fassung

- Iteratoren bieten:
  1. eine einheitliche Schnittstelle zum Aufzählen von Elementen;
  2. ohne dabei eine List o.ä. aufbauen zu müssen (Speicher-schonend!);
  3. weniger Beschränkungen als Generatoren;
  4. die Möglichkeit, unendliche Mengen zu durchlaufen (natürlich nur endliche Anfangsstücke!).

Fibonacci-Iterator

fibiter.py

```
class FibIterator():
    def __init__(self, max_n=0):
        self.max_n = max_n
        self.n, self.a, self.b = 0, 0, 1

    def __iter__(self):
        return self                 # an iterator object!

    def __next__(self):
        self.n += 1
        self.a, self.b = self.b, self.a + self.b
        if not self.max_n or self.n <= self.max_n:
            return self.a
        else:
            raise StopIteration
```

Prolog: Ausnahmen (Exceptions)
Generatoren
Iteratoren
Dateien
Zugabe: Sudoku
Zusammenfassung

FibIterator bei der Arbeit

Python-Interpreter

```
>>> f = FibIterator(10)
>>> list(f)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
>>> list(f)
[]
>>> for i in FibIterator(): print(i)
...
1
1
2
3
5
...
```

4 Dateien

Prolog: Ausnahmen (Exceptions)

Generatoren

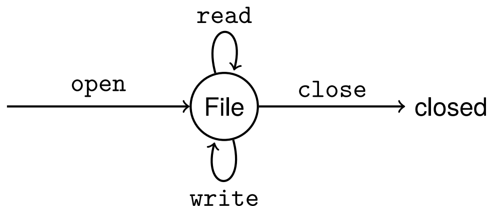
Iteratoren

Dateien

Zugabe: Sudoku

Zusammenfassung

Dateien bearbeiten



- `open(filename : str, mode = 'r': str)`:
  Öffnet die Datei mit dem Namen `filename` und liefert ein `file`-Objekt zurück.
- `mode` bestimmt, ob die Datei gelesen oder geschrieben werden soll (oder beides):
  - `"r"`: Lesen von Textdateien mit `file.read()`
  - `"w"`: Schreiben von Textdateien mit `file.write()`
  - `"r+"`: Schreiben und Lesen von Textdateien

Prolog: Ausnahmen (Exceptions)
Generatoren
Iteratoren
Dateien
Zugabe: Sudoku
Zusammenfassung

Muster: Dateien mit Iterator und Kontextmanager lesen

```
with open (filename) as f:
    # initialize
    for line in f:
        # process this line
```

- `with resource as name` startet einen Kontextmanager
- Falls Ausnahmen im zugehörigen Block auftreten, wird die `resource` korrekt finalisiert. D.h. es ist kein extra `try`-Block erforderlich.
- Für Dateien heisst das, dass sie geschlossen werden, egal wie der `with`-Block verlassen wird.

Prolog: Ausnahmen (Exceptions)
Generatoren
Iteratoren
Dateien
Zugabe: Sudoku
Zusammenfassung

Beispiel: fgrep

Das Unix-Kommando `fgrep` durchsucht Dateien nach einem festen String.

```
def fgrep (subject:str, filename:str):
    with open (filename) as f:
        for line in f:
            if subject in line:
                print(line)

fgrep ("joke", "killing_joke_sketch.txt")
```

Prolog: Ausnahmen (Exceptions)
Generatoren
Iteratoren
Dateien
Zugabe: Sudoku
Zusammenfassung

Beispiel: fgrep mit Ausgabe

Prolog: Ausnahmen (Exceptions)
Generatoren
Iteratoren
Dateien
Zugabe: Sudoku
Zusammen-fassung

```
def fgrep2 (subject:str, infile:str, outfile:str):
    with open (infile) as fin, open (outfile, 'w') as fout:
        for line in fin:
            if subject in line:
                print(line, file=fout)
```

- Hier schützt `with` zwei Ressourcen, die Eingabedatei und die Ausgabedatei.
- Zum Schreiben wird `print` mit dem Keyword-Argument `file` verwendet.

5 Zugabe: Sudoku

Prolog: Ausnahmen (Exceptions)

Generatoren

Iteratoren

Dateien

Zugabe: Sudoku

Zusammen-fassung

Sudoku

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

Sudoku-Regeln

1. Eine Gruppe von Zellen ist entweder
   - eine Zeile,
   - eine Spalte oder
   - ein fett umrahmter 3x3 Block.
2. Jede Gruppe muss die Ziffern 1-9 genau einmal enthalten.
3. Fülle die leeren Zellen, sodass (2) erfüllt ist!

Ein ähnlich schweres Problem wie das Rucksackproblem

UNI FREIBURG

Prolog: Ausnahmen (Exceptions)
Generatoren
Iteratoren
Dateien
Zugabe: Sudoku
Zusammenfassung

Suchraum

- Der Suchraum hat in den meisten Fällen (17 Vorgaben) eine Größe von ca. 10^{61} möglichen Kombinationen.
- Würden wir eine Milliarde (10^9) Kombinationen pro Sekunde testen können, wäre die benötigte Rechenzeit $10^{61}/(10^9 \cdot 3 \cdot 10^7) \approx 3 \cdot 10^{44}$ Jahre.
- Die Lebensdauer des Weltalls wird mit 10^{11} Jahren angenommen (falls das Weltall geschlossen ist).
- Selbst bei einer Beschleunigung um den Faktor 10^{30} würde die Rechnung nicht innerhalb der Lebensdauer des Weltalls abgeschlossen werden können.
- Trotzdem scheint das Lösen von Sudokus ja nicht so schwierig zu sein ...

Sudoku mit Backtracking lösen

- Wir repräsentieren das Spielfeld durch ein Dictionary.
- Das Dictionary bildet das Paar `(row, col)` auf ein `int` zwischen 1 und 9 ab.
- Wir möchten das initiale Spielfeld von einer Datei einlesen.
- Beispiel (leere Felder durch `-`, vgl. Wikipedia):

```
53--7----
6--195---
-98----6-
8---6---3
4--8-3--1
7---2---6
-6----28-
---419--5
----8--79
```

Prolog: Ausnahmen (Exceptions)
Generatoren
Iteratoren
Dateien
Zugabe: Sudoku
Zusammenfassung

Einlesen/Ausdrucken des Spielfelds

Prolog: Ausnahmen (Exceptions)

Generatoren

Iteratoren

Dateien

Zugabe: Sudoku

Zusammenfassung

```
def read_board_from_file(file):
    with open (file) as bfile:
        board = dict()
        row = 0
        for line in bfile:
            for col, x in zip(range(9), line):
                if x in "123456789":
                    board[ (row, col) ] = int(x)
            row += 1
        return board
```

```
def print_board(board):
    for row in range(9):
        line = ""
        for col in range(9):
            line += str(board.get ((row, col), '-'))
        print (line)
```

Prüfen eines Zuges

Prolog: Ausnahmen (Exceptions)
Generatoren
Iteratoren
Dateien
Zugabe: Sudoku
Zusammenfassung

Gesucht wird

```
can_set(board, row:int, col:int, x:int) -> bool
```

- Kann `x` in Zeile `row` und Spalte `col` auf dem Spielfeld `board` eingetragen werden, ohne dass die Regeln verletzt werden?
- Annahme: `board[(row, col)]` ist noch nicht gesetzt!

Prüfen eines Zuges

```
def can_set(board, row, col, x):
    # which block
    brow = row // 3
    bcol = col // 3
    # check whether x already occurs in row or col
    for (r,c) , v in board.items():
        if v == x:
            if r == row: return False
            if c == col: return False
            if r // 3 == brow and c // 3 == bcol: return False
    return True
```


Prolog: Ausnahmen (Exceptions)

Generatoren

Iteratoren

Dateien

Zugabe: Sudoku

Zusammenfassung

Naive Suche mit Backtracking

- Durchlaufe systematisch die Zeilen/Spalten-Paare von `(0,0)` bis `(8,8)`
- Für alle `x` in 1, ..., 9 für die die Zelle gemäß `can_set` gesetzt werden kann, setze `x` auf dem Spielfeld und versuche diese Belegung zu einer Lösung zu vervollständigen.
- Falls alle `x`e erfolglos durchprobiert wurden, dann Backtracking zur vorherigen Zelle.

Prolog: Ausnahmen (Exceptions)
Generatoren
Iteratoren
Dateien
Zugabe: Sudoku
Zusammenfassung

Naive Suche mit Backtracking

Prolog: Ausnahmen (Exceptions)
Generatoren
Iteratoren
Dateien
Zugabe: Sudoku
Zusammenfassung

```
def find_solution2(board, rc = (0,0)):
    while rc is not None and rc in board:
        rc = advance (*rc)
    if rc is None:
        yield board.copy()
    else:
        row, col = rc
        rc = advance (row, col)
        for x in range(1,10):
            if can_set (board, row, col, x):
                board[(row, col)] = x
                yield from find_solution (board, rc)
                del board[(row,col)]
```

6 Zusammenfassung

Prolog: Ausnahmen (Exceptions)

Generatoren

Iteratoren

Dateien

Zugabe: Sudoku

Zusammenfassung

Zusammenfassung

- Ausnahmen sind in Python allgegenwärtig.
- Sie können mit `try`, `except`, `else` und `finally` abgefangen und behandelt werden.
- Mit `raise` können Ausnahmen ausgelöst werden.
- Generatoren sehen aus wie Funktionen, geben ihre Werte aber mit `yield` zurück.
- Ein Generatoraufruf liefert einen Iterator, der beim Aufruf von `next()` bis zum nächsten `yield` läuft.
- Generatoren sind besonders nützlich zur Lösung von Suchproblemen mit Backtracking.
- Iteratoren besitzen die Methoden `__iter__` und `__next__`.
- Mit Aufrufen der `next()`-Funktion werden alle Elemente aufgezählt.
- Iterierbare Objekte besitzen eine Methode `__iter__`, die mit Hilfe der Funktion `iter()` oder in einer `for`-Schleife einen Iterator erzeugen.
- Dateien erlauben es, externe Inhalte zu lesen und zu schreiben.
- Am einfachsten mit dem Kontextmanager `with`/`as`.