

# Informatik I: Einführung in die Programmierung

## 16. Ausnahmen, Generatoren und Iteratoren, Backtracking

Albert-Ludwigs-Universität Freiburg



Prof. Dr. Peter Thiemann

13.01.2026

# 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

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

```
>>> print({"spam": "egg"}["parrot"])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'parrot'
```

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

Prolog:  
Ausnahmen  
(Exceptions)

Ausnahmen  
try-except

try-except-else  
Blöcke

finally-Blöcke  
raise-Anweisung

Generatoren

Iteratoren

Dateien

Zugabe:  
Sudoku

Zusammen-  
fassung

## Anwendungen von Ausnahmen

- 1 Signalisieren einer Situation, die nicht spezifiziert 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.

Prolog:  
Ausnahmen  
(Exceptions)

Ausnahmen

try-except

try-except-else-  
Blöcke

finally-Blöcke

raise-Anweisung

Generatoren

Iteratoren

Dateien

Zugabe:  
Sudoku

Zusammen-  
fassung

## Beispiele

`exception OSError` (1) This exception is raised when a system function returns a system-related error, including I/O failures such as “file not found” or “disk full” (not for illegal argument types or other incidental errors).

`exception RecursionError` (1) This exception is raised when the interpreter detects that the maximum recursion depth is exceeded.

`exception IndexError` (2) Raised when a sequence subscript is out of range.

`exception KeyError` (2) Raised when a mapping (dictionary) key is not found in the set of existing keys.

Prolog:  
Ausnahmen  
(Exceptions)

Ausnahmen

try-except

try-except-else-  
Blöcke

finally-Blöcke

raise-Anweisung

Generatoren

Iteratoren

Dateien

Zugabe:  
Sudoku

Zusammen-  
fassung

- Das Auslösen einer Ausnahme bricht den normalen Programmablauf ab.
- Stattdessen beginnt ab der Stelle, wo die Ausnahme ausgelöst wurde, die Suche nach der **Ausnahmebehandlung** mit der Anweisung `try` mit Optionen `except`, `finally` und `else`.
- Wird die Ausnahme nicht innerhalb des aktuellen Funktionsaufrufs behandelt, so wird der Funktionsaufruf beendet, der zugehörige Kellerrahmen entfernt und die Ausnahme wird an den **Aufrufer der Funktion** hochgereicht. Dabei wird **kein Rückgabewert** bestimmt!
- Dort wird die Suche nach einer `try` Anweisung beginnend nach dem Aufruf der Funktion fortgesetzt.
- Das geschieht solange, bis sich ein Kellerrahmen findet, in dem die Ausnahme behandelt wird.

Prolog:  
Ausnahmen  
(Exceptions)

Ausnahmen

`try-except`

`try-except-else`-  
Blöcke

`finally`-Blöcke

`raise`-Anweisung

Generatoren

Iteratoren

Dateien

Zugabe:  
Sudoku

Zusammen-  
fassung

- Ausnahmen sind selbst Objekte.
- Sie sind Instanzen von Subklassen der Klasse `BaseException`.
- Die Subklasse `Exception` dient als Basisklasse für selbstdefinierte Ausnahmen.

Prolog:  
Ausnahmen  
(Exceptions)

Ausnahmen

`try-except`

`try-except-else`-  
Blöcke

`finally`-Blöcke

`raise`-Anweisung

Generatoren

Iteratoren

Dateien

Zugabe:  
Sudoku

Zusammen-  
fassung

# try-except



Eine `try-except`-Anweisung behandelt Ausnahmen, die während der Ausführung des `try`-Blocks auftreten. Wenn dort keine Ausnahme ausgelöst wurde oder die Ausnahme in einer der `except`-Klauseln bearbeitet wurde, geht es nach der `try`-Anweisung einfach weiter.

```
try:
    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
```

Prolog:  
Ausnahmen  
(Exceptions)

Ausnahmen

`try-except`

`try-except-else`-  
Blöcke

`finally`-Blöcke

`raise`-Anweisung

Generatoren

Iteratoren

Dateien

Zugabe:  
Sudoku

Zusammen-  
fassung



## except-Blöcke (1)

`except XYError:`

Ein solcher Block wird ausgeführt, wenn innerhalb des `try`-Blocks eine Ausnahme ausgelöst wird, die eine Instanz von `XYError` (oder Subklasse) ist.

`except XYError as e:`

Wie oben; zusätzlich wird das Ausnahmeobjekt an die Variable `e` zugewiesen.

`except (XYError, YZError):`

Ein Tupel fängt mehrere Ausnahmetypen gemeinsam ab: sowohl `XYError` als auch `YZError`.

`except:`

So werden unspezifisch **alle** Ausnahmen abgefangen.

Prolog:  
Ausnahmen  
(Exceptions)

Ausnahmen

`try-except`

`try-except-else`-  
Blöcke

`finally`-Blöcke

`raise`-Anweisung

Generatoren

Iteratoren

Dateien

Zugabe:  
Sudoku

Zusammen-  
fassung

## except-Blöcke (2)

- Die `except`-Blöcke werden der Reihe nach abgearbeitet, bis der erste passende Block gefunden wird (falls überhaupt einer passt).
- Unspezifische `except`-Blöcke sind daher nur an letzter Stelle sinnvoll.
- In einem `except`-Block kann die abgefangene Ausnahme mit einer `raise`-Anweisung **ohne Argument** weitergereicht werden.

Prolog:  
Ausnahmen  
(Exceptions)

Ausnahmen

`try-except`

`try-except-else`-  
Blöcke

`finally`-Blöcke

`raise`-Anweisung

Generatoren

Iteratoren

Dateien

Zugabe:  
Sudoku

Zusammen-  
fassung

# 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:
    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

Zusammen-  
fassung

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

```
try:  
    critical_code()  
finally:  
    print("Ich komme zurück...")
```

- Der finally-Block wird *immer* beim Verlassen des try-Blocks ausgeführt:
  - Bei einem return im try-Block wird der finally-Block vor Rückgabe des Ergebnisses ausgeführt.
  - Bleibt eine Ausnahme auch nach Bearbeitung der try-Anweisung bestehen, so wird sie nach Ausführung des finally-Blocks weitergegeben.

Prolog:  
Ausnahmen  
(Exceptions)

Ausnahmen  
try-except

try-except-else-  
Blöcke

**finally-Blöcke**  
raise-Anweisung

Generatoren

Iteratoren

Dateien

Zugabe:  
Sudoku

Zusammen-  
fassung

## 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

Zusammen-  
fassung

- 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.

Prolog:  
Ausnahmen  
(Exceptions)

Ausnahmen  
try-except

try-except-else-  
Blöcke

finally-Blöcke  
raise-Anweisung

Generatoren

Iteratoren

Dateien

Zugabe:  
Sudoku

Zusammen-  
fassung

# 2 Generatoren



## ■ Anwendung von Generatoren

Prolog:  
Ausnahmen  
(Exceptions)

**Generatoren**

Anwendung von  
Generatoren

Iteratoren

Dateien

Zugabe:  
Sudoku

Zusammen-  
fassung

# Das Geheimnis von range & Co

```
>>> 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**

Anwendung von  
Generatoren

Iteratoren

Dateien

Zugabe:  
Sudoku

Zusammen-  
fassung



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

Prolog:  
Ausnahmen  
(Exceptions)

Generatoren

Anwendung von  
Generatoren

Iteratoren

Dateien

Zugabe:  
Sudoku

Zusammen-  
fassung

# Ein Generator für range



UNI  
FREIBURG

```
>>> from typing import Iterator
>>> def myRange(n : int) -> Iterator[int]:
...     """ generator that counts from 0 to n-1 """
...     i = 0
...     while i<n:
...         yield i
...         i = i+1
... 
```

- Neue Anweisung: **yield**. Ihr Vorkommen bewirkt, dass der Funktionsaufruf `myRange(3)` als Ergebnis einen **Generator** liefert.
- Ein Generator ist ein Objekt, das eine Folge von Werten erzeugt, die mit der Funktion `next()` einmal durchlaufen werden kann.
- Typ eines Generators (vereinfacht): **Iterator[T]**, wobei T der Typ vom Argument von `yield` ist.

Prolog:  
Ausnahmen  
(Exceptions)

Generatoren

Anwendung von  
Generatoren

Iteratoren

Dateien

Zugabe:  
Sudoku

Zusammen-  
fassung

## Action: Die Funktion `next`



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

Prolog:  
Ausnahmen  
(Exceptions)

Generatoren

Anwendung von  
Generatoren

Iteratoren

Dateien

Zugabe:  
Sudoku

Zusammen-  
fassung

- Intuitiv “läuft” `myRange` beim ersten Aufruf von `next` bis zum `yield`.
- Beim nächsten `next` läuft es an dieser Stelle weiter bis zum nächsten `yield`.

- Führt Buch über den Stand der Ausführung des Generators.
- Stand der Ausführung = Kellerrahmen: Belegung der lokalen Variablen und Parameter, sowie die als nächstes auszuführende Anweisung.
- Bei Konstruktion (d.h. Aufruf der Generatorfunktion) wird ein Generatorobjekt erzeugt:
  - Kellerrahmen mit den übergebenen Parametern,
  - erste Anweisung des Funktionsrumpfes.
  - Die Generatorfunktion läuft **noch nicht** los!
- Beispiel: Beim Aufruf von `gen = myRange (3)` enthält das Generatorobjekt
  - Parameter `n = 3`
  - Nächste Anweisung ist `i = 0`

Prolog:  
Ausnahmen  
(Exceptions)

Generatoren  
Anwendung von  
Generatoren

Iteratoren

Dateien

Zugabe:  
Sudoku

Zusammen-  
fassung

## Aufruf von `next(gen)`

- 1 Restauriere den zuletzt gespeicherten Stand der Ausführung.
- 2 Fahre dort fort mit der Ausführung des Rumpfes des Generators (Bsp: Funktionsrumpf von `myRange`).
- 3 Führe aus bis zum nächsten `yield`:
  - Speichere den aktuellen Stand der Ausführung im Generator.
  - Liefere das Argument von `yield` als Ergebnis.
- 4 Falls das Ende des Rumpfs ohne `yield` erreicht wird:
  - Speichere den aktuellen Stand der Ausführung im Generator.
  - Löse die Ausnahme `StopIteration` aus.

Prolog:  
Ausnahmen  
(Exceptions)

Generatoren

Anwendung von  
Generatoren

Iteratoren

Dateien

Zugabe:  
Sudoku

Zusammen-  
fassung

- Viele Funktionen erlauben Iteratoren als Argumente, z.B. die Funktion `list`:

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

- Intern baut sie die Ergebnisliste durch wiederholtes Aufrufen von `next` auf, bis `StopIteration` ausgelöst wird.
- Auch eine `for`-Schleife kann durch einen Iterator gesteuert werden:

```
>>> for i in myRange(3): print(i, end=' ')
...
0 1 2
```

Prolog:  
Ausnahmen  
(Exceptions)

Generatoren

Anwendung von  
Generatoren

Iteratoren

Dateien

Zugabe:  
Sudoku

Zusammen-  
fassung

# Ein Generator muss nicht endlich sein



```
def upFrom(n:int) -> Iterator[int]:  
    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  
KeyboardInterrupt
```

Prolog:  
Ausnahmen  
(Exceptions)

Generatoren

Anwendung von  
Generatoren

Iteratoren

Dateien

Zugabe:  
Sudoku

Zusammen-  
fassung

# Abfragen eines (potentiell) unendlichen Generators

## Zu Fuß mit Ausnahmen

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

## Elegant mit for-Schleife

```
def printGenFor(gen: Iterator[Any]):  
    for v in gen:  
        print(v)
```

Prolog:  
Ausnahmen  
(Exceptions)

Generatoren

Anwendung von  
Generatoren

Iteratoren

Dateien

Zugabe:  
Sudoku

Zusammen-  
fassung



## Zwei weitere Beispiele: map und filter



```
def myMap[A,B](f : Callable[[A], B],
               seq : Iterator[A]
               ) -> Iterator[B]:
    for x in seq:
        yield f (x)

def twox1 (x : int) -> int:
    return 2*x+1

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

Was wird gedruckt?

```
def myFilter[A](p : Callable[[A], bool],
                seq : Iterator[A]
                ) -> Iterator[A]:
    for x in seq:
        if p(x):
            yield x

def div3 (x : int) -> bool:
    return x % 3 == 0

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

Was wird gedruckt?

Prolog:  
Ausnahmen  
(Exceptions)

Generatoren

Anwendung von  
Generatoren

Iteratoren

Dateien

Zugabe:  
Sudoku

Zusammen-  
fassung

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



## Zusammenfassung

## Definition: Subliste

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

## 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  $L = [x_1, \dots, x_n]$ , wenn alle  $x_i$  unterschiedlich sind.

Prolog:  
Ausnahmen  
(Exceptions)

Generatoren  
Anwendung von  
Generatoren

Iteratoren

Dateien

Zugabe:  
Sudoku

Zusammen-  
fassung

## Ein spezielles 0/1 Rucksackproblem

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

## Ein schweres Problem

- Der naive Algorithmus probiert alle maximal möglichen  $2^{\text{len}(L)}$  Sublisten durch.
- Es ist nicht bekannt, ob es für dieses Problem einen effizienteren Algorithmus gibt.

Prolog:  
Ausnahmen  
(Exceptions)

Generatoren  
Anwendung von  
Generatoren

Iteratoren

Dateien

Zugabe:  
Sudoku

Zusammen-  
fassung

# Ein rekursiver Algorithmus mit Generatoren



```
def knapsack[A](goal : int, items : list[tuple[A,int]]) -> Iterator[list[A]]:
  if goal == 0:
    yield []                                # solution found
  elif not items:
    return                                  # out of items, no solution
  else:
    item0, weight = items[0]
    remaining_items = items[1:]
    yield from knapsack(goal, remaining_items) # solutions without item0
    if weight <= goal:
      for solution in knapsack(goal - weight, remaining_items):
        yield [item0] + solution
```

Prolog:  
Ausnahmen  
(Exceptions)

Generatoren

Anwendung von  
Generatoren

Iteratoren

Dateien

Zugabe:  
Sudoku

Zusammen-  
fassung

## Beispielhafte Eingabe (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}
```

- Wird der Rumpf eines Generators mit `return` beendet, löst der Generator eine `StopIteration`-Ausnahme aus.
- Anstelle des Dictionaries wird `list(gifts.items())` übergeben, eine Liste von key-value-Paaren.
- `yield from gen` entspricht der Schleife

```
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 und dem Verzicht auf Änderungen in der Datenstruktur ist die Rücknahme von Schritten besonders einfach.

Prolog:  
Ausnahmen  
(Exceptions)

Generatoren  
Anwendung von  
Generatoren

Iteratoren

Dateien

Zugabe:  
Sudoku

Zusammen-  
fassung

# 3 Iteratoren



Prolog:  
Ausnahmen  
(Exceptions)

Generatoren

**Iteratoren**

Dateien

Zugabe:  
Sudoku

Zusammen-  
fassung



- Ein Objekt heißt **Container-Objekt**, falls es untergeordnete Objekte verwaltet.
- Die `for`-Schleife kann für viele **Container-Objekte** die Elemente durchlaufen.
- Dazu gehören Sequenzen, Tupel, Listen, Strings, Dictionaries, Mengen usw:

```
>>> for e1 in {1, 5, 3, 0}: print(e1, end=' ')  
...  
0 1 3 5
```

- Dies alles sind Beispiele für **iterierbare Objekte** (Generalisierung von Generatoren).
- Das Verhalten dieser Objekte wird durch das **Iterator-Protokoll** bestimmt.

Prolog:  
Ausnahmen  
(Exceptions)

Generatoren

Iteratoren

Dateien

Zugabe:  
Sudoku

Zusammen-  
fassung

- Das **Iterator-Protokoll** unterscheidet zwei Arten von Objekten: **iterierbare Objekte** (Typ `Iterable[X]`) und **Iteratoren** (Typ `Iterator[X]`).
- Ein iterierbares Objekt implementiert die dunder Methode `__iter__`, die dann ein **Iterator**-Objekt zurückliefert.
- Ein Iterator-Objekt implementiert die dunder Methoden
  - `__iter__`, die dann immer `self` liefert, und
  - `__next__`, die das nächste Element liefert. Gibt es kein weiteres Element, so löst die Methode die Ausnahme **StopIteration** aus.
- Die Funktion `iter(object)` ruft `object.__iter__()` auf.
- Die Funktion `next(object)` ruft `object.__next__()` auf.

Prolog:  
Ausnahmen  
(Exceptions)

Generatoren

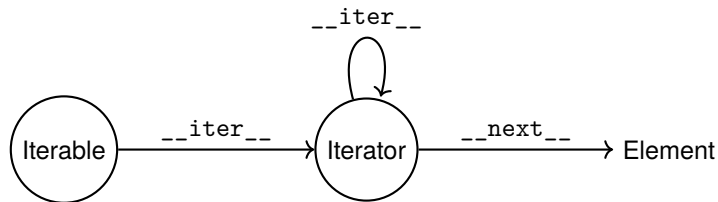
Iteratoren

Dateien

Zugabe:  
Sudoku

Zusammen-  
fassung

## Das Iterator-Protokoll (2)



- Ein **iterierbares Objekt** (Iterable) erzeugt bei jedem Aufruf von `__iter__` einen **neuen Iterator** für eine Menge von Objekten.
- Ein **Iterator** liefert **sich selbst** beim Aufruf von `__iter__`; jeder Aufruf von `__next__` liefert ein neues Objekt aus der Menge.
- Da jeder Iterator die `__iter__`-Methode besitzen, können Iteratoren auch dort verwendet werden, wo ein iterierbares Objekt erwartet wird (z.B. `for`-Schleife).

Prolog:  
Ausnahmen  
(Exceptions)

Generatoren

**Iteratoren**

Dateien

Zugabe:  
Sudoku

Zusammen-  
fassung

# Implementierung der for-Schleife

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

wird intern wie die folgende while-Schleife ausgeführt

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

Prolog:  
Ausnahmen  
(Exceptions)

Generatoren

Iteratoren

Dateien

Zugabe:  
Sudoku

Zusammen-  
fassung

```
>>> seq = ['Crackpot', 'Religion']
>>> seq_iter = iter(seq)
>>> seq_iter
<list_iterator object at 0x110d24ca0>
>>> print(next(seq_iter))
Crackpot
>>> print(next(seq_iter))
Religion
>>> print(next(seq_iter))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Prolog:  
Ausnahmen  
(Exceptions)

Generatoren

Iteratoren

Dateien

Zugabe:  
Sudoku

Zusammen-  
fassung

# Iterierbare Objekte vs. Iteratoren (1)

Ein Iterator ist nach einem Durchlauf, der mit `StopIteration` abgeschlossen wurde, **erschöpft**, wie in diesem 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

Zusammen-  
fassung

# Iterierbare Objekte vs. Iteratoren (2)

Alternativ: erzeuge bei jedem Start eines Schleifendurchlaufs einen neuen Iterator.

## 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

Zusammen-  
fassung

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

```
>>> range_obj = range(10)
>>> range_obj
range(0, 10)
>>> range_iter = iter(range_obj)
>>> range_iter
<range_iterator object at 0x110d51d40>
>>> list(range_iter)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range_iter)
[]
>>> list(range_obj)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Prolog:  
Ausnahmen  
(Exceptions)

Generatoren

Iteratoren

Dateien

Zugabe:  
Sudoku

Zusammen-  
fassung



Erinnerung:

```
>>> zz = zip(range(20), range(0,20,3)); zz
```

```
<zip object at 0x110d1e600>
```

```
>>> list(zz)
```

```
[(0, 0), (1, 3), (2, 6), (3, 9), (4, 12), (5, 15), (6, 18)]
```

- Für die Implementierung von `zip` muss explizit das Iterator-Protokoll verwendet werden, da **zwei** Eingaben unabhängig voneinander iteriert werden müssen.
- Eine Implementierung als Generator mit einer `for`-Schleife ist daher nicht möglich!

Prolog:  
Ausnahmen  
(Exceptions)

Generatoren

Iteratoren

Dateien

Zugabe:  
Sudoku

Zusammen-  
fassung

```
def myZip[A,B](s1: Iterable[A], s2: Iterable[B]) -> Iterator[tuple[A,B]]:  
  i1 = iter(s1)  
  i2 = iter(s2)  
  try:  
    while True:  
      e1 = next(i1)  
      e2 = next(i2)  
      yield (e1, e2)  
  except StopIteration:  
    pass
```

Prolog:  
Ausnahmen  
(Exceptions)

Generatoren

Iteratoren

Dateien

Zugabe:  
Sudoku

Zusammen-  
fassung

# Beispiel: Fibonacci-Iterator

fibiter.py

```
@dataclass
class FibIterator:
    maxn : int = 0

    def __post_init__(self):
        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.maxn or self.n <= self.maxn:
            return self.a
        else:
            raise StopIteration
```

Prolog:  
Ausnahmen  
(Exceptions)

Generatoren

Iteratoren

Dateien

Zugabe:  
Sudoku

Zusammen-  
fassung

## 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
...
```

Prolog:  
Ausnahmen  
(Exceptions)

Generatoren

**Iteratoren**

Dateien

Zugabe:  
Sudoku

Zusammen-  
fassung

Iteratoren bieten:

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

Prolog:  
Ausnahmen  
(Exceptions)

Generatoren

**Iteratoren**

Dateien

Zugabe:  
Sudoku

Zusammen-  
fassung

# 4 Dateien



Prolog:  
Ausnahmen  
(Exceptions)

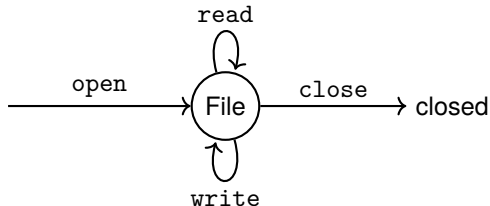
Generatoren

Iteratoren

**Dateien**

Zugabe:  
Sudoku

Zusammen-  
fassung



- `open(filename : str, mode = 'r': str) -> file:`  
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

Zusammen-  
fassung

# Muster: Dateien mit Iterator und Kontextmanager lesen



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

- Die Anweisung `with resource as name:` startet einen **Kontextmanager**
- Der Ausdruck `resource` initialisiert eine Ressource. Sie ist im zugehörigen Block als `name` verfügbar.
- 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

Zusammen-  
fassung



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", "text/killing_joke_sketch.txt")
```

Prolog:  
Ausnahmen  
(Exceptions)

Generatoren

Iteratoren

Dateien

Zugabe:  
Sudoku

Zusammen-  
fassung

## Beispiel: fgrep mit Ausgabe



```
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 der Kontextmanager zwei Ressourcen, die Eingabedatei und die Ausgabedatei.
- Zum Schreiben in eine Datei wird `print` mit dem Keyword-Argument `file` verwendet.

Prolog:  
Ausnahmen  
(Exceptions)

Generatoren

Iteratoren

Dateien

Zugabe:  
Sudoku

Zusammen-  
fassung

# 5 Zugabe: Sudoku



Prolog:  
Ausnahmen  
(Exceptions)

Generatoren

Iteratoren

Dateien

**Zugabe:**  
**Sudoku**

Zusammen-  
fassung

			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!

Prolog:  
Ausnahmen  
(Exceptions)

Generatoren

Iteratoren

Dateien

Zugabe:  
Sudoku

Zusammen-  
fassung

## 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.
- 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 ...

Prolog:  
Ausnahmen  
(Exceptions)

Generatoren

Iteratoren

Dateien

Zugabe:  
Sudoku

Zusammen-  
fassung

- Repräsentiere das Spielfeld durch ein Dictionary  
`type Board = dict[Pos, set[int]]` mit  
`type Pos = tuple[int, int]`.
- Das Dictionary `b : Board` bildet das Paar `(row, col)` auf die Menge der möglichen Werte an Zeile `row` und Spalte `col` ab.
  - Dabei ist  $row, col \in \{1, \dots, 9\}$ .
  - Wir verwenden die Invariante  $\emptyset \subset b[(row, col)] \subseteq \{1, \dots, 9\}$ .

Prolog:  
Ausnahmen  
(Exceptions)

Generatoren

Iteratoren

Dateien

Zugabe:  
Sudoku

Zusammen-  
fassung

- Wir möchten das initiale Spielfeld von einer Datei einlesen.

- Wenn ein Feld mit  $k$  vorbesetzt ist, dann gilt

$b[(row, col)] = \{k\}$ .

- Wenn ein Feld frei ist, dann gilt

$b[(row, col)] = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ .

- Beispiel (leere Felder durch -, entnommen 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

Zusammen-  
fassung

# Einlesen/Ausdrucken des Spielfelds



```
def read_board_from_file(filename : str) -> Board:
    with open (filename, 'r') as bfile:
        board = dict()
        empty = set(range(1,10))
        row = 1
        for line in bfile:
            for col, x in zip(range(1,10), line):
                board[ (row, col) ] = {int(x)} if x in "123456789" else empty.copy()
            row += 1
    return board
```

```
def print_board(board : Board):
    for row in range(1,10):
        line = ""
        for col in range(1,10):
            line += print_single(board[(row, col)])
        print (line)
```

Prolog:  
Ausnahmen  
(Exceptions)

Generatoren

Iteratoren

Dateien

Zugabe:  
Sudoku

Zusammen-  
fassung



- Durchlaufe systematisch die Zeilen/Spalten-Paare von (1,1) bis (9,9).
- Betrachte die Zelle `candidates = b[(row,col)]`. Wir können voraussetzen, dass diese Zelle nicht leer ist! (Warum?)
- Für jeden möglichen Kandidaten `c` in `candidates`:
  - Setze die Zelle auf `c`.
    - Entferne `c` aus den anderen Zellen in der gleichen Zeile.
    - Entferne `c` aus den anderen Zellen in der gleichen Spalte.
    - Entferne `c` aus den anderen Zellen im gleichen Block.
  - Wenn dabei eine Zelle leer wird, verwerfen wir den Kandidaten `c`.
  - Wenn dabei keine Zelle leer wird, dann betrachten wir rekursiv die nächste Zelle.
  - Danach stellen wir den Zustand vor Betrachtung von `c` wieder her (Backtracking) und betrachten den nächsten Kandidaten.
- Wenn die letzte Zelle erfolgreich bearbeitet wurde, haben wir eine Lösung!

Prolog:  
Ausnahmen  
(Exceptions)

Generatoren

Iteratoren

Dateien

Zugabe:  
Sudoku

Zusammen-  
fassung

Gesucht wird

`propagate_row(b : Board, p : Pos, c : int) -> bool`

- Annahme: c wurde schon in `b[p]` eingetragen.
- Entferne c aus allen weiteren Zellen der gleichen Zeile!
- Liefere `False`, falls dabei eine Zelle leer wird.
- Ansonsten liefere `True`.

Prolog:  
Ausnahmen  
(Exceptions)

Generatoren

Iteratoren

Dateien

Zugabe:  
Sudoku

Zusammen-  
fassung

Gesucht wird

`copy_board (b : Board) -> Board`

- Es muss eine vollständige Kopie angefertigt werden, weil `b` noch für das Backtracking benötigt wird!
  - Ein neues Dictionary
  - Eine frische Kopie von jeder Menge

Prolog:  
Ausnahmen  
(Exceptions)

Generatoren

Iteratoren

Dateien

Zugabe:  
Sudoku

Zusammen-  
fassung

# Naive Suche mit Backtracking

```
def try_from(b : Board, p : Optional[Pos] = None):  
    p = next_pos(p)  
    if p is None:  
        print_board(b)  
        return  
    candidates = b[ p ]  
    for c in candidates:  
        next_b = copy_board(b)  
        next_b[ p ] = { c }  
        if (propagate_row(next_b, p, c) and  
            propagate_col(next_b, p, c) and  
            propagate_blk(next_b, p, c)):  
            try_from(next_b, p)
```

Prolog:  
Ausnahmen  
(Exceptions)

Generatoren

Iteratoren

Dateien

Zugabe:  
Sudoku

Zusammen-  
fassung

# 6 Zusammenfassung



Prolog:  
Ausnahmen  
(Exceptions)

Generatoren

Iteratoren

Dateien

Zugabe:  
Sudoku

**Zusammen-  
fassung**

- **Ausnahmen** sind in Python allgegenwärtig.
  - Sie können mit `raise` ausgelöst werden.
  - Sie können mit `try`, `except`, `else` und `finally` **abgefangen** und **behandelt** werden.
- **Generatoren** sehen aus wie Funktionen, geben ihre Werte aber mit `yield` zurück.
- Ein **Generatorkaufruf** 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__`.
- Durch Aufrufen der `__next__`-Methode werden alle Elemente aufgezählt.
- **Iterierbare Objekte** besitzen eine Methode `__iter__`, die einen **Iterator** für die enthaltenen Objekte erzeugt.
- Dateien erlauben es, externe Inhalte zu lesen und zu schreiben.
- Am einfachsten mit dem **Kontextmanager** `with/as`.

Prolog:  
Ausnahmen  
(Exceptions)

Generatoren

Iteratoren

Dateien

Zugabe:  
Sudoku

Zusammen-  
fassung