

Programmieren in Python

7. Dictionaries, Mengen & Dateien

Malte Helmert

Albert-Ludwigs-Universität Freiburg

KI-Praktikum, Sommersemester 2009

In dieser Lektion befassen wir uns mit weiteren wichtigen eingebauten Datentypen von Python:

- ▶ Dictionaries: `dict`
- ▶ Mengen: `set` und `frozenset`
- ▶ Dateien: `open` und `file`

In dieser Lektion befassen wir uns mit weiteren wichtigen eingebauten Datentypen von Python:

- ▶ Dictionaries: `dict`
- ▶ Mengen: `set` und `frozenset`
- ▶ Dateien: `open` und `file`

- ▶ Dictionaries (Wörterbücher) oder kurz *Dicts* sind assoziative Arrays so wie `map` in C++ oder `HashMap` in Java.
- ▶ Dictionaries speichern Paare von *Schlüsseln* (*keys*) und zugehörigen *Werten* (*values*) und sind so implementiert, dass man sehr effizient den Wert zu einem gegebenen Schlüssel bestimmen kann.
- ▶ Im Gegensatz zu Sequenzen sind Dictionaries *ungeordnete* Container; es ist nicht sinnvoll, von einem ersten (zweiten, usw.) Element zu sprechen.

Python-Interpreter

```
>>> description = {"parrot": "dead", "spam": "tasty",
...                (1, 2, 3): "no witchcraft"}
>>> print description["parrot"]
dead
>>> print "spam" in description
True
>>> description["parrot"] = "pining for the fjords"
>>> description["slides"] = "unfinished"
>>> print description
{'slides': 'unfinished', (1, 2, 3): 'no witchcraft',
 'parrot': 'pining for the fjords', 'spam': 'tasty'}
```

Dictionaries können auf verschiedene Weisen erzeugt werden:

- ▶ `{key1: value1, key2: value2, ...}`:
Hier sind `key1`, `value1` usw. normale Python-Objekte, z.B. Strings, Zahlen oder Tupel.
- ▶ `dict(key1=value1, key2=value2, ...)`:
Hier sind die Schlüssel `key1` usw. **Variablennamen**, die vom `dict`-Konstruktor in Strings konvertiert werden.
Die Werte `value1` usw. sind normale Objekte.
- ▶ `dict(sequence_of_pairs)`:
`dict([(key1, value1), (key2, value2), ...])`
entspricht `{key1: value1, key2: value2, ...}`.
- ▶ `dict.fromkeys(seq[, value])`:
Ist `seq` eine Sequenz mit Elementen `key1`, `key2`, erhalten wir `{key1: value, key2: value, ...}`.
Wird `value` weggelassen, wird `None` verwendet.

Dictionaries erzeugen: Beispiele

Python-Interpreter

```
>>> print {"parrot": "dead", "spam": "tasty", 10:
"zehn"}
{10: 'zehn', 'parrot': 'dead', 'spam': 'tasty'}
>>> print dict(six=6, nine=9, six_times_nine=42)
{'six_times_nine': 42, 'six': 6, 'nine': 9}
>>> investigators = ["Justus", "Peter", "Bob"]
>>> girlfriends = ["Lys", "Kelly", "Liz"]
>>> print dict(zip(investigators, girlfriends))
{'Bob': 'Liz', 'Peter': 'Kelly', 'Justus': 'Lys'}
>>> print dict.fromkeys("abc")
{'a': None, 'c': None, 'b': None}
>>> print dict.fromkeys(range(3), "eine Zahl")
{0: 'eine Zahl', 1: 'eine Zahl', 2: 'eine Zahl'}
```

- ▶ `key in d`, `d.has_key(key)`:
True, falls das Dictionary `d` den Schlüssel `key` enthält.
(Die Variante mit `has_key` ist *deprecated*.)
- ▶ `bool(d)`:
True, falls das Dictionary nicht leer ist.
- ▶ `len(d)`:
Liefert die Zahl der Elemente (Paare) in `d`.
- ▶ `d.copy()`:
Liefert eine (flache) Kopie von `d`.

- ▶ `d[key]`:
Liefert den Wert zum Schlüssel `key`.
Fehler bei nicht vorhandenen Schlüsseln.
- ▶ `d.get(key[, default])`:
Wie `d[key]`, aber es ist kein Fehler, wenn `key` nicht vorhanden ist. Stattdessen wird in diesem Fall `default` zurückgeliefert (`None`, wenn kein Default angegeben wurde).

food_inventory.py

```
def get_food_amount(food):
    food_amounts = {"spam": 2, "egg": 1, "cheese": 4}
    return food_amounts.get(food, 0)

for food in ["egg", "vinegar", "cheese"]:
    amount = get_food_amount(food)
    print "We have enough %s for %d people." % (
        food, amount)

# Ausgabe:
# We have enough egg for 1 people.
# We have enough vinegar for 0 people.
# We have enough cheese for 4 people.
```

- ▶ `d[key] = value`:
Weist dem Schlüssel `key` einen Wert zu. Befindet sich bereits ein Paar mit Schlüssel `key` in `d`, wird es ersetzt.
- ▶ `d.setdefault(key, default)`:
Vom Rückgabewert äquivalent zu `d.get(key, default)`. Falls das Dictionary den Schlüssel noch nicht enthält, wird zusätzlich `d[key] = default` ausgeführt.
↪ In neueren Python-Versionen kann der durch `setdefault` abgedeckte Use-Case komfortabler mit der Klasse `collections.defaultdict` behandelt werden.

hobbies.py

```
hobby_dict = {}
def add_hobby(person, hobby):
    hobby_dict.setdefault(person, []).append(hobby)

add_hobby("Justus", "Reading")
add_hobby("Peter", "Cycling")
add_hobby("Bob", "Music")
add_hobby("Justus", "Riddles")
add_hobby("Bob", "Girls")
print hobby_dict
# Ausgabe: {'Bob': ['Music', 'Girls'],
#           'Peter': ['Cycling'],
#           'Justus': ['Reading', 'Riddles']}
```

Äquivalentes Beispiel mit defaultdict

hobbies.py

```
from collections import defaultdict
hobby_dict = defaultdict(list)
def add_hobby(person, hobby):
    hobby_dict[person].append(hobby)

add_hobby("Justus", "Reading")
add_hobby("Peter", "Cycling")
add_hobby("Bob", "Music")
add_hobby("Justus", "Riddles")
add_hobby("Bob", "Girls")
print hobby_dict
# Ausgabe: defaultdict(<type 'list'>,
#                 {'Bob': ['Music', 'Girls'],
#                 'Peter': ['Cycling'],
#                 'Justus': ['Reading', 'Riddles']})
```

- ▶ `d.update(another_dict)`:
Führt `d[key] = value` für alle `(key, value)`-Paare in `another_dict` aus.
Überträgt also alle Einträge aus `another_dict` nach `d` und überschreibt bestehende Einträge mit dem gleichen Schlüssel.
- ▶ `d.update(sequence_of_pairs)`:
Entspricht `d.update(dict(sequence_of_pairs))`.
- ▶ `d.update(key1=value1, key2=value2, ...)`:
Entspricht `d.update(dict(key1=value1, key2=value2, ...))`.

Operationen auf Dictionaries: Einträge entfernen

- ▶ `del d[key]:`
Entfernt das Paar mit dem Schlüssel `key` aus `d`. Fehler, falls kein solches Paar existiert.
- ▶ `d.pop(key[, default]):`
Entfernt das Paar mit dem Schlüssel `key` aus `d` und liefert den zugehörigen Wert. Existiert kein solches Paar, wird `default` zurückgeliefert, falls angegeben (sonst Fehler).
- ▶ `d.popitem():`
Entfernt ein (willkürliches) Paar (`key`, `value`) aus `d` und liefert es zurück. Fehler, falls `d` leer ist.
- ▶ `d.clear():`
Entfernt alle Elemente aus `d`.
 - ▶ Was ist der Unterschied zwischen `d.clear()` und `d = {}`?

- ▶ `d.keys()`, `d.iterkeys()`:
Liefert alle Schlüssel in `d` zurück.
`keys` liefert eine Liste, `iterkeys` einen Iterator
(vgl. `enumerate`).
- ▶ `d.values()`, `d.itervalues()`:
Liefert alle Werte in `d` zurück.
`values` liefert eine Liste, `itervalues` einen Iterator.
- ▶ `d.items()`, `d.iteritems()`:
Liefert alle Einträge, d.h. (`key`, `value`)-Paare in `d` zurück.
`items` liefert eine Liste (von Paaren), `iteritems` einen
Iterator.
- ▶ Dictionaries können auch in `for`-Schleifen verwendet werden.
Dabei wird die Methode `iterkeys` benutzt, `for`-Schleifen
über Dictionaries durchlaufen also die *Schlüssel*.

Dictionaries sind als Hashtabellen implementiert.

Das hat einige Konsequenzen:

- ▶ Alle grundlegenden Operationen (`key in d`, `d[key] = value`, `print d[key]`) haben eine erwartete Laufzeit von $O(1)$.
- ▶ Dictionaries haben keine spezielle Ordnung für die Elemente. Daher liefert `keys` die Schlüssel nicht unbedingt in der Einfügereihenfolge.
- ▶ Objekte, die als Schlüssel in einem Dictionary verwendet werden, dürfen nicht verändert werden. Ansonsten könnte es zu Problemen kommen.

Veränderliche Dictionary-Keys (1)

```
potential_trouble.py
```

```
mydict = {}  
mylist = [10, 20, 30]  
mydict[mylist] = "spam"  
mylist.remove(20)  
print mydict.get([10, 20, 30])  
print mydict.get([10, 30])  
  
# Was kann passieren?  
# Was sollte passieren?
```

Veränderliche Dictionary-Keys (2)

- ▶ Um solche Problem zu vermeiden, sind in Python nur *unveränderliche* Objekte wie Tupel, Strings und Zahlen als Dictionary-Schlüssel erlaubt.
 - ▶ Genauer: Selbst Tupel sind verboten, wenn sie direkt oder indirekt veränderliche Objekte beinhalten.
- ▶ Verboten sind also Listen und Dictionaries oder Objekte, die Listen oder Dictionaries beinhalten.
- ▶ Für die *Werte* sind beliebige Objekte zulässig; die Einschränkung gilt nur für Schlüssel!

Python-Interpreter

```
>>> mydict = {"spam", "egg": [1, 2, 3]}
```

```
>>> mydict[[10, 20]] = "spam"
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in ?
```

```
TypeError: list objects are unhashable
```

```
>>> mydict[("spam", [], "egg")] = 1
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in ?
```

```
TypeError: list objects are unhashable
```

In dieser Lektion befassen wir uns mit weiteren wichtigen eingebauten Datentypen von Python:

- ▶ Dictionaries: `dict`
- ▶ Mengen: `set` und `frozenset`
- ▶ Dateien: `open` und `file`

Seit Python 2.4 gibt es die builtin-Datentypen `set` und `frozenset` zur Repräsentation von Mengen.

- ▶ Sets (Mengen) sind ungeordnete Sammlungen von Elementen.
- ▶ Mengenelemente sind einzigartig; eine Menge kann also nicht dasselbe Element “mehrmals” beinhalten.
- ▶ Mengenelemente müssen *hashbar* sein (wie bei Dictionaries).
- ▶ `set` vs. `frozenset`:
 - ▶ `frozensets` sind unveränderlich \rightsquigarrow *hashbar*
 - ▶ `sets` sind veränderlich
 - ▶ Insbesondere können `frozensets` also auch als Elemente von `sets` und `frozensets` verwendet werden.

- ▶ Mengen sind in Python ähnlich wie Dictionaries über Hash-Tabellen implementiert, so dass die elementaren Mengenoperationen alle sehr effizient unterstützt werden.
- ▶ Im Folgenden wird die Zeitkomplexität der Operationen in O -Notation dargestellt. Dabei ist n immer die Anzahl der Elemente der ersten Mengen, m die Anzahl der Elemente der zweiten Menge (wenn zwei Mengen beteiligt sind).
- ▶ Die Laufzeitangaben gehen vom “durchschnittlichen Fall” für das Hashverhalten aus und sind bei einigen Operationen (z.B. Einfügen) *amortisiert*. In einigen degenerierten Fällen können manche Operationen deutlich mehr Zeit erfordern.

Wir teilen die Operationen auf Mengen in Gruppen ein:

- ▶ Konstruktion
- ▶ Grundlegende Operationen
- ▶ Einfügen und Entfernen von Elementen
- ▶ Mengenvergleiche
- ▶ Klassische Mengenoperationen

Konstruktion von Mengen

- ▶ `set()`: $O(1)$
Erzeugt eine veränderliche leere Menge.
- ▶ `set(iterable)`: $O(n)$
Erzeugt eine veränderliche Menge aus den Elementen des Objekts `iterable`.
- ▶ `frozenset()`: $O(1)$
Erzeugt eine unveränderliche leere Menge.
- ▶ `frozenset(iterable)`: $O(n)$
Erzeugt eine unveränderliche Menge aus den Elementen des Objekts `iterable`.

- ▶ `set` und `frozenset` können aus beliebigen iterierbaren Objekten `iterable` erstellt werden, also solchen, die `for` unterstützen (z.B. `str`, `list`, `dict`, `set`, `frozenset`.)
- ▶ Jedoch dürfen innerhalb von `iterable` nur *hashbare* Objekte (z.B. keine Listen!) enthalten sein (sonst `TypeError`).

Konstruktion von Mengen: Beispiele (1)

Python-Interpreter

```
>>> print set("spamspam")
set(['a', 'p', 's', 'm'])
>>> print frozenset("spamspam")
frozenset(['a', 'p', 's', 'm'])
>>> print set(["spam", 1, [2, 3]])
Traceback (most recent call last):
File "<stdin>", line 1, in ?
TypeError: list objects are unhashable
>>> print set(("spam", 1, (2, 3)))
set([1, (2, 3), 'spam'])
>>> print set({"spam": 20, "jam": 30})
set(['jam', 'spam'])
```

Python-Interpreter

```
>>> s = set(["jam", "spam"])
>>> print set([1, 2, 3, s])
Traceback (most recent call last):
File "<stdin>", line 1, in ?
TypeError: set objects are unhashable
>>> print set([1, 2, 3, frozenset(s)])
set([1, 2, 3, frozenset(['jam', 'spam'])])
```

- ▶ `element in s`, `element not in s`: $O(1)$
Test auf Mitgliedschaft bzw. Nicht-Mitgliedschaft (liefert `True` oder `False`).
- ▶ `bool(s)`: $O(1)$
`True`, falls die Menge `s` nicht leer ist.
- ▶ `len(s)`: $O(1)$
Liefert die Zahl der Elemente der Menge `s`.
- ▶ `for element in s`: $O(n)$
Über Mengen kann natürlich iteriert werden.
- ▶ `s.copy()`: $O(n)$
Liefert eine (flache) Kopie der Menge `s`.

Mengen: Einfügen und Entfernen von Elementen

- ▶ `s.add(element)`: nur für set, $O(1)$
Fügt das Objekt `element` zur Menge `s` hinzu, falls es noch nicht Element der Menge ist.
- ▶ `s.remove(element)`: nur für set, $O(1)$
Entfernt `element` aus der Menge `s`, falls es dort enthalten ist.
Sonst: `KeyError`.
- ▶ `s.discard(element)`: nur für set, $O(1)$
Wie `remove`, aber kein Fehler, wenn `element` nicht in der Menge enthalten ist.
- ▶ `s.pop()`: nur für set, $O(1)$
Entfernt ein willkürliches Element aus `s` und liefert es zurück.
- ▶ `s.clear()`: nur für set, $O(n)$
Entfernt alle Elemente aus der Menge `s`.

Viele Operationen auf Mengen sind sowohl als benannte Methoden als auch über Operatoren verfügbar. Beispiel:

- ▶ Benannte Methode: `s.intersection_update(t)`
- ▶ Operator: `s &= t`.

Diese Formen sind immer identisch abgesehen davon, dass

- ▶ die benannten Methoden ihr Argument automatisch in ein `set/frozenset` konvertieren, während
- ▶ die Operatoren ein `set/frozenset` auf der rechten Seite erfordern.

Mengenvergleiche

- ▶ `s.issubset(t)`, `s <= t`: $O(\min(n, m))$
Testet, ob alle Elemente von s in t enthalten sind ($s \subseteq t$)
- ▶ `s < t`: $O(\min(n, m))$
Wie `s <= t`, aber echter Teilmengentest ($s \subset t$).
- ▶ `s.issuperset(t)`, `s >= t`, `s > t`: $O(\min(n, m))$
Analog für Obermengentests bzw. echte Obermengentests.
- ▶ `s == t`: $O(\min(n, m))$
Gleichheitstest. Wie `s <= t` and `t <= s`, aber effizienter.
 - ▶ Anders als bei den anderen Operatoren ist es *kein* Typfehler, wenn nur eines der Argumente eine Menge ist.
 - ▶ In diesem Fall ist `s == t` immer `False`.
 - ▶ Ein `set` kann gleich einem `frozenset` sein.
- ▶ `s != t`: $O(\min(n, m))$
Äquivalent zu `not (s == t)`. Anmerkungen gelten analog.

Klassische Mengenoperationen

- ▶ `s.union(t)`, `s | t`: $O(n + m)$
- `s.intersection(t)`, `s & t`: $O(n + m)$
- `s.difference(t)`, `s - t`: $O(n + m)$
- `s.symmetric_difference(t)`, `s ^ t`: $O(n + m)$

Liefert Vereinigung ($s \cup t$), Schnitt ($s \cap t$), Mengendifferenz ($s \setminus t$) bzw. symmetrische Mengendifferenz ($s \Delta t$) von `s` und `t`.
Das Resultat hat denselben Typ wie `s`.

- ▶ `s.update(t)`, `s |= t`: nur für `set`, $O(m)$
- `s.intersection_update(t)`, `s &= t`: nur für `set`, $O(n + m)$
- `s.difference_update(t)`, `s -= t`: nur für `set`, $O(m)$
- `s.symmetric_difference_update(t)`, `s ^= t`: nur für `set`, $O(m)$

In-Situ-Varianten der Mengenoperationen.

(Ändern also `s`, statt eine neue Menge zu liefern.)

Python-Interpreter

```
>>> s1 = frozenset([1, 2, 3])
>>> s2 = set([3, 4, 5])
>>> s1 | s2
frozenset([1, 2, 3, 4, 5])
>>> s2 | s1
set([1, 2, 3, 4, 5])
>>> s1 | [3, 4, 5]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: unsupported operand type(s) for |:
'frozenset' and 'list'
>>> s1.union([3, 4, 5])
frozenset([1, 2, 3, 4, 5])
```

Klassische Mengenoperationen: Beispiele (2)

Python-Interpreter

```
>>> s1 = set([1, 2, 3, 4, 5])
>>> s2 = set([3, 4])
>>> s2.update([4, 5, 6, 7])
>>> print s2
set([3, 4, 5, 6, 7])
>>> print s1 - s2
set([1, 2])
>>> print s1.symmetric_difference(s2)
set([1, 2, 6, 7])
```

In dieser Lektion befassen wir uns mit weiteren wichtigen eingebauten Datentypen von Python:

- ▶ Dictionaries: `dict`
- ▶ Mengen: `set` und `frozenset`
- ▶ Dateien: `open` und `file`

- ▶ Unsere Programme kranken bisher daran, dass sie kaum mit der Außenwelt kommunizieren können. Um das zu ändern, beschäftigen wir uns jetzt mit Dateien.
- ▶ Dateien werden in Python entweder mit `open` oder mit `file` geöffnet. Konzeptuell besteht ein kleiner Unterschied:
 - ▶ `file` ist eine Klasse und die Benutzung von `file` als Funktion damit ein Konstrukturaufruf.
 - ▶ `open` ist dagegen eine „normale“ Funktion, die ein `file`-Objekt zurückliefert.
- ▶ Tatsächlich verhalten sich `file` und `open` aber exakt gleich. Genauer gesagt ist `open` in etwa so definiert:

```
open.py
```

```
open = file
```

- ▶ Offiziell wird empfohlen, `open` zu benutzen, aber man findet auch viel Code, der `file` verwendet. Im Folgenden verwenden wir `open`, aber alles Gesagte gilt genauso für `file`.

- ▶ `open(filename[, mode[, bufsize]])`:

Öffnet die Datei mit dem Namen `filename` und liefert ein entsprechendes `file`-Objekt zurück.

Die optionalen Parameter haben folgende Bedeutung:

- ▶ `mode` bestimmt, ob die Datei gelesen oder geschrieben werden soll (oder beides). Mögliche Werte werden auf der nächsten Folie beschrieben. Lässt man den Parameter weg, wird die Datei zum Lesen geöffnet.
- ▶ `bufsize` gibt an, ob und wie Zugriffe auf diese Datei gepuffert werden sollen:
 - ▶ negativ/nicht angegeben: Überlasse die Wahl dem Betriebssystem.
 - ▶ 0: Nicht puffern.
 - ▶ 1: Einzelne Zeilen puffern.
 - ▶ $x \geq 2$: Ca. x Bytes puffern.

open unterstützt folgende Modi:

- ▶ Lesen: "r" für Textdateien, "rb" für Binärdateien.
- ▶ Lesen im universellen Newline-Modus: "U" oder "rU".
↪ Erklärung folgt später.
- ▶ Schreiben: "w" bzw. "wb".
Achtung: Existiert die Datei bereits, wird sie überschrieben (gelöscht).
- ▶ Lesen und Schreiben: "r+" bzw. "r+b".
- ▶ Anhängen: "a" bzw. "ab".
Schreibt an das Ende einer (bestehenden) Datei.
Legt eine neue Datei an, falls erforderlich.

Außerdem gibt es noch die seltener genutzten Modi w+, w+b, a+ und a+b, die hier nicht beschrieben werden.

- ▶ Unter Unix/Linux kann man auf das nachgestellte b für Binärdateien verzichten; es schadet aber nicht, es anzugeben.

- ▶ Unix, Windows und Mac OS verwenden leider drei unterschiedliche Kodierungen für Zeilenenden.
- ▶ Daher sehen Zeilenenden häufig merkwürdig aus, wenn man Dateien zwischen diesen Betriebssystemen kopiert.
- ▶ Normalerweise folgt Python bei der Interpretation von Zeilenenden den Konventionen des Betriebssystems, auf dem das Programm ausgeführt wird.
- ▶ Im universellen Newline-Modus ist das anders: Hier werden alle drei Arten von Zeilenenden gleichermaßen akzeptiert.
- ▶ Dieser Modus ist also beispielsweise dann interessant, wenn man unter Unix Dateien von einem Windows-System einlesen will oder umgekehrt.

- ▶ `f.close()`:
Schließt eine Datei.
 - ▶ Geschlossene Dateien können nicht weiter für Lese- oder Schreibzugriffe verwendet werden.
 - ▶ Es ist erlaubt, Dateien mehrfach zu schließen.
 - ▶ Es ist normalerweise nicht nötig, Dateien zu schließen, weil dies automatisch geschieht, sobald das entsprechende Objekt nicht mehr benötigt wird.
Allerdings gibt es alternative Implementierungen von Python, bei denen dies nicht der Fall ist. Vollkommen portable Programme sollten also `close` verwenden.

`file`-Objekte haben folgende drei Attribute, die ausgelesen, aber nicht modifiziert werden können:

- ▶ `f.name`: Dateiname
- ▶ `f.mode`: Modus, unter dem die Datei geöffnet wurde.
- ▶ `f.closed`: `True`, falls die Datei geschlossen wurde, sonst `False`.

Die grundlegenden Methoden zum Lesen und Schreiben sind:

- ▶ `f.read([size])`:
Liest `size` Bytes aus der Datei und liefert sie als String zurück.
Falls vorher das Ende der Datei erreicht wird, werden alle verbleibenden Daten gelesen; ebenso wenn `size` weggelassen wird.
- ▶ `f.write(string)`:
Schreibt `string` in die Datei; es wird kein Newline o.ä. angehängt. Wenn die Datei gepuffert wird, sind Änderungen meistens nicht sofort für andere Programme sichtbar.
- ▶ `f.flush()`:
Leert die internen Puffer; ausstehende Daten werden in die Datei geschrieben.

Schreib- und Leseposition setzen und abfragen

- ▶ `f.tell()`:

Liefert die aktuelle Schreib-/Leseposition, gemessen in Bytes nach dem Dateianfang.

- ▶ `f.seek(offset[, whence])`:

Setzt die Schreib-/Leseposition auf die Position `offset`...

- ▶ vom Dateianfang gemessen, falls `whence == 0` oder weggelassen,
- ▶ von der aktuellen Position aus gemessen, falls `whence == 1`,
- ▶ vom Dateiende gemessen, falls `whence == 2` ist.

Im zweiten und dritten Fall ist `offset` oft negativ.

- ▶ Bei Textdateien besteht je nach Betriebssystem und Zeilenendekonvention kein direkter Zusammenhang zwischen dem Offset und der Zahl an gelesenen/geschriebenen Zeichen. Hier sollte man `seek` nur Werte übergeben, die man vorher mit `tell` ermittelt hat.
- ▶ Bei Dateien, die zum Anhängen geöffnet wurden, hat `seek` auf vielen Plattformen keinen oder einen unerwarteten Effekt.

Tatsächlich werden die vorgestellten Methoden wie `read` und `write` nur selten direkt verwendet, da es für die üblichen Aufgaben einfachere Operationen gibt.

Zum Einlesen von Dateien verwendet man üblicherweise die Iteration (`for line in f`):

- ▶ Über Dateien kann ebenso wie über Sequenzen oder Dictionaries iteriert werden.
- ▶ Dabei wird in jedem Schleifendurchlauf eine Zeile aus der Datei gelesen und der Schleifenvariable (hier `line`) zugewiesen, inklusive Newline-Zeichen am Ende.
- ▶ Da die Eingabe speziell gepuffert wird, sollte man die Iteration über Dateien nicht mit `read` und `tell` mischen, um verwirrende Ergebnisse zu vermeiden.
 - ▶ Will man innerhalb der `for`-Schleife „Extra“-Zeilen lesen, verwendet man dafür die Methode `f.next()`.

```
grep_spam.py
```

```
def grep_spam(filename):  
    for line in open(filename):  
        if "spam" in line.lower():  
            print line.rstrip("\n")  
  
grep_spam("spam_sketch.txt")
```

An dieser Stelle lohnt es sich anzumerken, dass viele Funktionen, die wir im Zusammenhang mit Sequenzen besprochen haben, mit *beliebigen* Objekte funktionieren, über die man iterieren kann, also beispielsweise auch mit Dictionaries und Dateien.

- ▶ Beispielsweise kann man mit `list(f)` eine Liste mit allen Zeilen einer Datei erzeugen oder mit `max(f)` die lexikographisch größte Zeile bestimmen.
- ▶ Es gibt allerdings auch Ausnahmen: `len(f)` funktioniert beispielsweise nicht. (Warum nicht?) Im Zweifelsfall hilft Ausprobieren oder die Dokumentation.

Auch Ausgaben werden selten mit `write` direkt ausgeführt. Stattdessen verwendet man oft eine erweiterte Form der `print`-Anweisung:

- ▶ In der Form

```
print >> f, ausdruck1, ausdruck2, ...
```

kann `print` benutzt werden, um in eine Datei `f` statt in die Standardausgabe zu schreiben.

- ▶ Die Form

```
print >> f
```

schreibt eine Leerzeile (genauer: ein Zeilenende) in die Datei `f`.

- ▶ Tatsächlich funktioniert `print >> f ...` für beliebige Objekte `f`, die über eine `write`-Methode verfügen sowie für `f is None`; in letzterem Fall wird in die Standardausgabe geschrieben.

Ein weiteres Feature von `print` blieb bisher unerwähnt und komplettiert die Beschreibung dieser Anweisung:

- ▶ Beendet man die `print`-Anweisung mit einem Komma (etwa `print "spam", "egg",`), dann wird kein Zeilenende erzeugt.
- ▶ Stattdessen wird die Ausgabe von nachfolgenden Ausgaben durch ein Leerzeichen getrennt.
- ▶ Dieses Leerzeichen wird jedoch nicht sofort geschrieben, sondern erst bei der nächsten Ausgabe, und auch nur dann, wenn nicht als nächstes ein Zeilenende geschrieben wird. Auf diese Weise werden Leerzeichen am Zeilenende vermieden.

```
grep_and_save_spam.py
```

```
def grep_and_save_spam(in_filename, out_filename):  
    outfile = open(out_filename, "w")  
    for line in open(in_filename):  
        if "spam" in line.lower():  
            print >> outfile, line.rstrip("\n")  
  
grep_and_save_spam("spam_sketch.txt", "spam.txt")
```

- ▶ `f.readline([size])`:
Liest eine Zeile (oder max. `size` Bytes, falls angegeben) aus der Datei. \rightsquigarrow Leerer String, wenn das Dateiende erreicht ist.
- ▶ `f.readlines([size])`:
Ruft wiederholt `f.readline` auf, bis das Dateiende erreicht ist oder etwa `size` Bytes eingelesen wurden (falls angegeben). Liefert eine Liste der eingelesenen Strings.
- ▶ `f.writelines(seq)`:
Schreibt eine Folge von Strings in die Datei. Entspricht `f.write("".join(seq))`.
Trotz des Namens müssen die einzelnen Strings keine einzelne Zeilen sein und es werden auch keine Zeilenenden eingefügt.
- ▶ `readline` und `readlines` werden in modernen Versionen von Python kaum noch verwendet; `for line in f` ist meist vorzuziehen und in der Regel auch effizienter.

Die folgenden Methoden und Attribute wurden aus verschiedenen Gründen nicht besprochen:

- ▶ `newlines`: nur für etwas exotische Anwendungen im Universal-Newlines-Modus interessant.
- ▶ `fileno`, `isatty`, `truncate`: relativ esoterisch und mehr oder weniger Betriebssystem-spezifisch.
- ▶ `encoding`: Unicode-spezifisch.
- ▶ `xreadlines`, `readinto`: deprecated bzw. inoffiziell.
- ▶ `softspace`: nur für den internen Gebrauch gedacht.