Informatik I: Einführung in die Programmierung

13. Dictionaries & Mengen

Albert-Ludwigs-Universität Freiburg

Bernhard Nebel

20. November 2015

1 Dictionaries

Dictionaries

- Dictionaries (Wörterbücher) oder kurz *Dicts* sind assoziative Arrays/Listen.
- 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.

Dictionaries: Ein Beispiel

Python-Interpreter

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

Dictionaries erzeugen

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` (und das Komma) weggelassen, wird `None` verwendet.

Exkurs: Die Punkt-Notation

- Manche Funktionen auf Dicts (oder auch anderen Python-Typen und -Objekten) werden mit Hilfe der Punkt-Notation angegeben: `dict.fromkeys(seq, value)`.
- Bei einem Typen (wie `dict`) wird dann ein entsprechendes Objekt erzeugt.
- Handelt es sich um ein Objekt, wird eine Operation auf dem Objekt durchgeführt
- Man nennt diese mit einem Punkt angehängten Funktionen Methoden.

→ Objekt-basierte Notation (führt zu OOP)

Dictionaries erzeugen: Beispiele

Python-Interpreter

```
>>> {"parrot": "dead", "spam": "tasty", 10: "zehn"}
{10: 'zehn', 'parrot': 'dead', 'spam': 'tasty'}
>>> dict(six=6, nine=9, six_times_nine=42)
{'six_times_nine': 42, 'nine': 9, 'six': 6}
>>> english = ["red", "blue", "green"]
>>> german = ["rot", "blau", "grün"]
>>> dict(zip(english, german))
{'red': 'rot', 'green': 'grün', 'blue': 'blau'}
>>> dict.fromkeys("abc")
{'a': None, 'c': None, 'b': None}
>>> dict.fromkeys(range(3), "eine Zahl")
{0: 'eine Zahl', 1: 'eine Zahl', 2: 'eine Zahl'}
```


Operationen auf Dictionaries: Grundlegendes

Sei `d` ein Dict:

- `key in d`:
  True, falls das Dictionary `d` den Schlüssel `key` enthält.
- `bool(d)` (bzw. einfach `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` (tiefe Kopie kommt gleich)

Operationen auf Dictionaries: Werte auslesen

- `d[key]`:
  Liefert den Wert zum Schlüssel `key`.
  Fehler bei nicht vorhandenen Schlüsseln.
- `d.get(key, default)` (oder `d.get(key)`):
  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).

get: Beispiel

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", food, "for", amount ,"people.")

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


Operationen auf Dictionaries: Werte eintragen

- `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)` (oder `d.setdefault(key)`):
  Vom Rückgabewert äquivalent zu `d.get(key, default)`.
  Falls das Dictionary den Schlüssel noch nicht enthält, wird zusätzlich `d[key] = default` ausgeführt.
- Hier kann man oft besser `defaultdict` aus dem Modul `collections` als Spezialisierung von `dict` einsetzen!
- `collections.defaultdict(`*Defaultgenerator*`)` liefert ein `dict`, bei dem immer die Funktion *Defaultgenerator* aufgerufen wird, wenn kein Wert für den *key* vorhanden ist.

Beispiel zu `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(<class 'list'>,
#            {'Bob': ['Music', 'Girls'],
#             'Peter': ['Cycling'],
#             'Justus': ['Reading', 'Riddles']})
```

Exkurs: Flaches und tiefes Kopieren (1)

- Wie schon bei Listen, erzeugt eine Zuweisung keine Kopie!

Python-Interpreter

```
>>> en_de={'red': 'rot', 'green': 'grün', 'blue':
'blau'}
>>> en_sw = en_de
>>> en_sw['green'] = 'gräa'
>>> en_de['green']
'gräa'
>>> en_de={'red': 'rot', 'green': 'grün', 'blue':
'blau'}
>>> en_sw = en_de.copy()
>>> en_sw['green'] = 'gräa'
>>> en_de['green']
'grün'
```

Visualisierung

Exkurs: Flaches und tiefes Kopieren (2)

- Rekursiv enthaltene Strukturen werden beim flachen Kopieren nicht kopiert!

Python-Interpreter

```
>>> snums={'even': [2, 4, 6], 'odd': [1, 3, 5]}
>>> sprimes = snums.copy()
>>> del(sprimes['even'][1:]); del(sprimes['odd'][0])
>>> snums
{'even': [2], 'odd': [3, 5]}
>>> import copy
>>> snums={'even': [2, 4, 6], 'odd': [1, 3, 5]}
>>> sprimes = copy.deepcopy(snums)
>>> del(sprimes['even'][1:]); del(sprimes['odd'][0]
>>> snums
{'even': [2, 4, 6], 'odd': [1, 3, 5]}
```

- Funktioniert bei Listen ebenso!

Geschachtelte Dicts

- Ebenso wie Listen kann man auch Dicts rekursiv einbetten.

Python-Interpreter

```
>>> en_de={'red': 'rot', 'green': 'grün', 'blue':
'blau'}
>>> de_fr ={'rot': 'rouge', 'grün': 'vert', 'blau':
'bleu'}
>>> dicts = {'en->de': en_de, 'de->fr': de_fr}
>>> dicts['de->fr']['blau']
'bleu'
>>> dicts['de->fr'][dicts['en->de']['blue']]
'bleu'
```


Operationen auf Dictionaries: Werte eintragen mit `update`

- `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)` (oder `d.pop(key)`):
  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()`:
  Enfernt alle Elemente aus `d`.
  - Was ist der Unterschied zwischen `d.clear()` und `d = {}`?

Operationen auf Dictionaries: Iteration

Die folgenden Methoden liefern iterierbare `views` zurück, die Änderungen an dem zugrundeliegenden `dict` reflektieren!

- `d.keys()`:
  Liefert alle Schlüssel in `d` zurück.
- `d.values()`:
  Liefert alle Werte in `d` zurück.
- `d.items()`:
  Liefert alle Einträge, d.h. `(key, value)`-Paare in `d` zurück.

- Dictionaries können auch in `for`-Schleifen verwendet werden. Dabei wird die Methode `keys` benutzt.
  `for`-Schleifen über Dictionaries durchlaufen also die *Schlüssel*.

Wie funktionieren Dictionaries?

Dictionaries sind als Hashtabellen implementiert:

- Es wird initial eine große Liste/Tabelle (die Hashtabelle) eingerichtet.
- Jeder Schlüssel wird mit Hilfe einer Hashfunktion in einen Index (dem Hashwert) übersetzt.
- Bei gleichen Hashwerten für verschiedene Schlüssel gibt es eine Spezialbehandlung.
- Der Zugriff erfolgt damit in (erwarteter) konstanter Zeit.
- 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"
del mylist[1]
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!

Veränderliche Dictionary-Keys (3)

Python-Interpreter

```
>>> mydict = {("parrot", "dead"): [1, 2, 3]}
>>> mydict[[10, 20]] = "spam"
Traceback (most recent call last): ...
TypeError: unhashable type: 'list'
>>> mydict[("parrot", [], "dead")] = 1
Traceback (most recent call last): ...
TypeError: unhashable type: 'list'
```


2 Mengen

Mengen

- Mengen sind Zusammenfassungen von Elementen (in unserem Fall immer endlich),
- Mengenelemente sind einzigartig; eine Menge kann also nicht dasselbe Element ‚mehrmals' beinhalten.
- Man könnte Mengen duch Listen implementieren (müsste dann immer die Liste durchsuchen)
- Man könnte Mengen durch Dicts implementieren, wobei die Elemente durch Schlüssel realisiert würden und der Wert immer `None` ist (konstante Zugriffszeit).
- Es gibt allerdings eigene Datentypen für Mengen in Python (auch mit Hilfe von Hashtabellen realisiert), die alle Mengenoperation unterstützen.

Mengen: `set` und `frozenset`

- Mengenelemente müssen *hashbar* sein (wie bei Dictionaries).
- `set` vs. `frozenset`:
  - `frozenset`s sind unveränderlich $\rightsquigarrow$ hashbar,
  - `set`s sind veränderlich
  - Insbesondere können `frozenset`s also auch als Elemente von `set`s und `frozenset`s verwendet werden.

Operationen auf Mengen

Wir teilen die Operationen auf Mengen in Gruppen ein:

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

Konstruktion von Mengen

- `{elem1, ..., elemN}`: Erzeugt die veränderliche Menge {`elem1`,...,`elemN`}.
- `set()`: Erzeugt eine veränderliche leere Menge.
- `set(iterable)`: Erzeugt eine veränderliche Menge aus Elementen von `iterable`.
- `frozenset()`: Erzeugt eine unveränderliche leere Menge.
- `frozenset(iterable)`: Erzeugt eine unveränderliche Menge aus Elementen von `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

```
>>> set("spamspam")
{'a', 'p', 's', 'm'}
>>> frozenset("spamspam")
frozenset({'a', 'p', 's', 'm'})
>>> set(["spam", 1, [2, 3]])
Traceback (most recent call last): ...
TypeError: unhashable type: 'list'
>>> set(("spam", 1, (2, 3)))
{1, (2, 3), 'spam'}
>>> set({"spam": 20, "jam": 30})
{'jam', 'spam'}
```

Konstruktion von Mengen: Beispiele (2)

Python-Interpreter

```
>>> s = set(["jam", "spam"])
>>> set([1, 2, 3, s])
Traceback (most recent call last): ...
TypeError: unhashable type: 'set'
>>> set([1, 2, 3, frozenset(s)])
{1, 2, 3, frozenset({'jam', 'spam'})}
```


Grundlegende Operationen auf Mengen

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

Mengen: Einfügen und Entfernen von Elementen

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

Benannte Methoden vs. Operatoren

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

- Operator: `s & t`.
- Benannte Methode: `s.intersection(t)`
- Zuweisungsoperator: `s &= t`.
- Benannte Modifikationsmethode: `s.intersection_update(t)`
- Im Falle der Methoden wird das Argument in eine Menge konvertiert, wenn das Argument *iterierbar* ist.

Mengenvergleiche

- `s.issubset(t)`, `s <= t`:
  Testet, ob alle Elemente von s in t enthalten sind ($s \subseteq t$)
- `s < t`:
  Wie `s <= t`, aber echter Teilmengentest ($s \subset t$).
- `s.issuperset(t)`, `s >= t`, `s > t`:
  Analog für Obermengentests bzw. echte Obermengentests.
- `s == t`:
  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 ein `frozenset` sein.
- `s != t`:
  Äquvalent zu `not (s == t)`.

Klassische Mengenoperationen

- `s.union(t)`, `s | t`
  `s.intersection(t)`, `s & t`
  `s.difference(t)`, `s - t`
  `s.symmetric_difference(t)`, `s ^ t`
  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`
  `s.intersection_update(t)`, `s &= t`
  `s.difference_update(t)`, `s -= t`
  `s.symmetric_difference_update(t)`, `s ^= t`
  In-Situ-Varianten der Mengenoperationen.
  (Ändern also `s`, statt eine neue Menge zu liefern.)

Klassische Mengenoperationen: Beispiele (1)

Python-Interpreter

```
>>> s1 = frozenset([1, 2, 3])
>>> s2 = set([3, 4, 5])
>>> s1 | s2
frozenset(1, 2, 3, 4, 5)
>>> s2 | s1
{1, 2, 3, 4, 5}
>>> s1 | [3, 4, 5]
Traceback (most recent call last): ...
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("dead")
>>> s2 = set("parrot")
>>> s2.update({'a','b','c','d'})
>>> s2
{'t', 'd', 'p', 'r', 'a', 'b', 'c', 'o'}
>>> s1 - s2
{'e'}
>>> s1.symmetric_difference(s2)
{'t', 'p', 'e', 'r', 'b', 'c', 'o'}
>>> (s1 - s2) | (s2 - s1)
{'o', 't', 'b', 'p', 'c', 'e', 'r' }
>>> (s1-s2)|(s2-s1) == s1.symmetric_difference(s2)
True
```

Zusammenfassung

- `dict`s können wir als Verallgemeinerung von Listen verstehen, bei denen der Index ein beliebiger (nicht änderbarer) Wert ist.
- Der Zugriff auf Elemente von `dict`s ist fast so effizient wie der Zugriff auf indizierte Listenelemente.
- `dict`s sind änderbare Elemente. Die Kopie eines `dict`s ist erst einmal nur eine Kopie der oberen Struktur!
- Um eine Kopie aller Substrukturen zu erreichen, muss das Modul `deepcopy` benutzt werden (funktioniert genauso bei Listen).
- Mengen in Python (`set`) kann man als `dict`s verstehen, bei denen alle Werte `None` sind.
- Es existieren alle Mengenoperationen.
- Mengen sind veränderliche Strukturen, eingefrorene Mengen (`frozenset`) dagegen nicht.