Informatik I: Einführung in die Programmierung

13. Dictionaries und Mengen

Albert-Ludwigs-Universität Freiburg

UNI FREIBURG

Prof. Dr. Peter Thiemann

14.12.2021

UNI FREIBURG

Dictionaries

Dictionaries

UNI FREIBURG

- Ein Dictionary (Wörterbuch), kurz *Dict*, ist eine Abbildung von *Schlüsseln* (*keys*) auf zugehörige *Werte* (*values*).

Dictionaries

UNI FREIBURG

- Ein Dictionary (Wörterbuch), kurz *Dict*, ist eine Abbildung von *Schlüsseln* (*keys*) auf zugehörige *Werte* (*values*).
- Alternative Bezeichnung: *assoziatives Array*

Dictionaries

UNI FREIBURG

- Ein Dictionary (Wörterbuch), kurz *Dict*, ist eine Abbildung von *Schlüsseln* (*keys*) auf zugehörige *Werte* (*values*).
- Alternative Bezeichnung: *assoziatives Array*
- Grundoperationen auf Dictionaries (mutable):

Dictionaries

UNI FREIBURG

- Ein Dictionary (Wörterbuch), kurz *Dict*, ist eine Abbildung von *Schlüsseln* (*keys*) auf zugehörige *Werte* (*values*).
- Alternative Bezeichnung: *assoziatives Array*
- Grundoperationen auf Dictionaries (mutable):
  - Einfügen einer Assoziation (Schlüssel $\mapsto$ Wert), evtl. vorhandene Assoziation mit Schlüssel wird überschrieben

Dictionaries

UNI FREIBURG

- Ein Dictionary (Wörterbuch), kurz *Dict*, ist eine Abbildung von *Schlüsseln* (*keys*) auf zugehörige *Werte* (*values*).
- Alternative Bezeichnung: *assoziatives Array*
- Grundoperationen auf Dictionaries (mutable):
  - Einfügen einer Assoziation (Schlüssel $\mapsto$ Wert), evtl. vorhandene Assoziation mit Schlüssel wird überschrieben
  - Entfernen einer Assoziation (Schlüssel),

Dictionaries

UNI FREIBURG

- Ein Dictionary (Wörterbuch), kurz *Dict*, ist eine Abbildung von *Schlüsseln* (*keys*) auf zugehörige *Werte* (*values*).
- Alternative Bezeichnung: *assoziatives Array*
- Grundoperationen auf Dictionaries (mutable):
  - Einfügen einer Assoziation (Schlüssel $\mapsto$ Wert), evtl. vorhandene Assoziation mit Schlüssel wird überschrieben
  - Entfernen einer Assoziation (Schlüssel),
  - Nachschlagen des Werts zu einem Schlüssel,

Dictionaries

UNI FREIBURG

- Ein Dictionary (Wörterbuch), kurz *Dict*, ist eine Abbildung von *Schlüsseln* (*keys*) auf zugehörige *Werte* (*values*).
- Alternative Bezeichnung: *assoziatives Array*
- Grundoperationen auf Dictionaries (mutable):
  - Einfügen einer Assoziation (Schlüssel $\mapsto$ Wert), evtl. vorhandene Assoziation mit Schlüssel wird überschrieben
  - Entfernen einer Assoziation (Schlüssel),
  - Nachschlagen des Werts zu einem Schlüssel,
  - Anwesenheit eines Schlüssels

Dictionaries

UNI FREIBURG

- Ein Dictionary (Wörterbuch), kurz *Dict*, ist eine Abbildung von *Schlüsseln* (*keys*) auf zugehörige *Werte* (*values*).
- Alternative Bezeichnung: *assoziatives Array*
- Grundoperationen auf Dictionaries (mutable):
  - Einfügen einer Assoziation (Schlüssel $\mapsto$ Wert), evtl. vorhandene Assoziation mit Schlüssel wird überschrieben
  - Entfernen einer Assoziation (Schlüssel),
  - Nachschlagen des Werts zu einem Schlüssel,
  - Anwesenheit eines Schlüssels
- Voraussetzungen

Dictionaries

UNI FREIBURG

- Ein Dictionary (Wörterbuch), kurz *Dict*, ist eine Abbildung von *Schlüsseln* (*keys*) auf zugehörige *Werte* (*values*).
- Alternative Bezeichnung: *assoziatives Array*
- Grundoperationen auf Dictionaries (mutable):
  - Einfügen einer Assoziation (Schlüssel $\mapsto$ Wert), evtl. vorhandene Assoziation mit Schlüssel wird überschrieben
  - Entfernen einer Assoziation (Schlüssel),
  - Nachschlagen des Werts zu einem Schlüssel,
  - Anwesenheit eines Schlüssels
- Voraussetzungen
  - Schlüssel müssen auf Gleichheit getestet werden können!

Dictionaries

UNI FREIBURG

- Ein Dictionary (Wörterbuch), kurz *Dict*, ist eine Abbildung von *Schlüsseln* (*keys*) auf zugehörige *Werte* (*values*).
- Alternative Bezeichnung: *assoziatives Array*
- Grundoperationen auf Dictionaries (mutable):
  - Einfügen einer Assoziation (Schlüssel $\mapsto$ Wert), evtl. vorhandene Assoziation mit Schlüssel wird überschrieben
  - Entfernen einer Assoziation (Schlüssel),
  - Nachschlagen des Werts zu einem Schlüssel,
  - Anwesenheit eines Schlüssels
- Voraussetzungen
  - Schlüssel müssen auf Gleichheit getestet werden können!
  - Schlüssel müssen unveränderlich (immutable) sein!

Dictionaries (Fortsetzung)

UNI FREIBURG

- Dictionaries sind so implementiert, dass der Wert zu einem gegebenen Schlüssel sehr effizient unabhängig von der Anzahl der bestehenden Einträge bestimmt werden kann.

Dictionaries (Fortsetzung)

- Dictionaries sind so implementiert, dass der Wert zu einem gegebenen Schlüssel sehr effizient unabhängig von der Anzahl der bestehenden Einträge bestimmt werden kann.
- Dictionaries sind *ungeordnet*; d.h., es ist nicht sinnvoll, von einem ersten (zweiten, usw.) Element zu sprechen.

Dictionaries (Fortsetzung)

UNI FREIBURG

- Dictionaries sind so implementiert, dass der Wert zu einem gegebenen Schlüssel sehr effizient unabhängig von der Anzahl der bestehenden Einträge bestimmt werden kann.
- Dictionaries sind *ungeordnet*; d.h., es ist nicht sinnvoll, von einem ersten (zweiten, usw.) Element zu sprechen.
- (Ein aktuelles Thema: key-value stores; das sind netzweit verteilte Dictionaries.)

Dictionaries: Ein Beispiel

UNI FREIBURG

Python-Interpreter

```
>>> description = {"walk": "silly", "parrot": "dead",
...                (1, 2, 3): "no witchcraft"}
>>> description["parrot"]
'dead'
>>> "walk" 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', 'walk': 'silly'}
```

Dictionaries erzeugen (Auswahl)

UNI FREIBURG

- `{key1: value1, key2: value2, ...}`
  Hier sind `key1`, `key2`, ... unveränderliche Python-Objekte, d.h. Zahlen, Strings, Tupel, etc.

Die Werte `value1`, `value2` usw. sind beliebige Objekte.

Dictionaries erzeugen (Auswahl)

UNI FREIBURG

- `{key1: value1, key2: value2, ...}`
  Hier sind `key1`, `key2`, ... unveränderliche Python-Objekte, d.h. Zahlen, Strings, Tupel, etc.
- `dict(key1=value1, key2=value2, ...)`:
  Hier sind die Schlüssel `key1`, `key2`, ... Variablennamen, die vom `dict`-Konstruktor in Strings konvertiert werden.

Die Werte `value1`, `value2` usw. sind beliebige Objekte.

Dictionaries erzeugen (Auswahl)

UNI FREIBURG

- `{key1: value1, key2: value2, ...}`
  Hier sind `key1`, `key2`, ... unveränderliche Python-Objekte, d.h. Zahlen, Strings, Tupel, etc.
- `dict(key1=value1, key2=value2, ...)`:
  Hier sind die Schlüssel `key1`, `key2`, ... Variablennamen, die vom `dict`-Konstruktor in Strings konvertiert werden.
- `dict(sop)` wobei `sop: Sequence[tuple[Any,Any]]`:
  `dict([(key1, value1), (key2, value2), ...])`
  entspricht `{key1: value1, key2: value2, ...}`.

Die Werte `value1`, `value2` usw. sind beliebige Objekte.

Dictionaries erzeugen: Beispiele

UNI FREIBURG

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'}
```

Operationen auf Dictionaries: Grundlegendes

UNI FREIBURG

Sei `d: dict`

- `key in d`:
  True, falls das Dictionary `d` den Schlüssel `key` enthält.

Operationen auf Dictionaries: Grundlegendes

Sei `d: dict`

- `key in d`:
  True, falls das Dictionary `d` den Schlüssel `key` enthält.
- `bool(d)`:
  True, falls das Dictionary nicht leer ist.

Operationen auf Dictionaries: Grundlegendes

UNI FREIBURG

Sei `d: dict`

- `key in d`:
  True, falls das Dictionary `d` den Schlüssel `key` enthält.
- `bool(d)`:
  True, falls das Dictionary nicht leer ist.
- `len(d)`:
  Liefert die Zahl der Elemente (Assoziationen) in `d`.

Operationen auf Dictionaries: Werte auslesen

UNI FREIBURG

- `d[`*`key`*`]`:
  Liefert den Wert zum Schlüssel *`key`*.
  Fehler bei nicht vorhandenen Schlüsseln.

Operationen auf Dictionaries: Werte auslesen

- `d[`*`key`*`]`:
  Liefert den Wert zum Schlüssel *`key`*.
  Fehler bei nicht vorhandenen Schlüsseln.
- `d.get(`*`key`*`, `*`value`*`)`:
  Wie `d[`*`key`*`]`, aber es ist kein Fehler, wenn *`key`* nicht vorhanden ist.
  Stattdessen wird in diesem Fall das optionale zweite Argument zurückgegeben (`None`, wenn es weggelassen wurde).

get: Beispiel

UNI FREIBURG

food_inventory.py

```
def get_food_amount(food : str):
  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 eine Assoziation mit Schlüssel *key* in `d`, wird sie ersetzt.

Operationen auf Dictionaries: Werte eintragen

- `d[`*`key`*`] = `*`value`*
  Weist dem Schlüssel *key* einen Wert zu. Befindet sich bereits eine Assoziation mit Schlüssel *key* in `d`, wird sie ersetzt.
- `d.setdefault(key, default= None)`
  Vom Rückgabewert äquivalent zu `d.get(key, default)`.
  Falls `d` den Schlüssel noch nicht enthält, wird `d[key] = default` ausgeführt.

Geschachtelte Dicts

UNI FREIBURG

- Auch Dicts können selbst Dicts enthalten.

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: Iteration

UNI FREIBURG

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

- `d.keys()`
  Liefert alle Schlüssel in `d` zurück.

Operationen auf Dictionaries: Iteration

UNI FREIBURG

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.

Operationen auf Dictionaries: Iteration

UNI FREIBURG

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)`-Assoziationen in `d` zurück.

Operationen auf Dictionaries: Iteration

UNI FREIBURG

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)`-Assoziationen 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?

UNI FREIBURG

Dictionaries sind als Hashtabellen implementiert:

- Bei der Erzeugung eines Dictionaries wird eine große Tabelle (die Hashtabelle) eingerichtet.

Wie funktionieren Dictionaries?

UNI FREIBURG

Dictionaries sind als Hashtabellen implementiert:

- Bei der Erzeugung eines Dictionaries wird eine große Tabelle (die Hashtabelle) eingerichtet.
- Jedem Schlüssel wird mit Hilfe einer Hashfunktion ein Hashwert zugeordnet, der als Tabellenindex dient.

Wie funktionieren Dictionaries?

UNI FREIBURG

Dictionaries sind als Hashtabellen implementiert:

- Bei der Erzeugung eines Dictionaries wird eine große Tabelle (die Hashtabelle) eingerichtet.
- Jedem Schlüssel wird mit Hilfe einer Hashfunktion ein Hashwert zugeordnet, der als Tabellenindex dient.
- Der zum Schlüssel gehörige Wert wird an dieser Stelle in der Tabelle abgelegt, es sei denn…

Wie funktionieren Dictionaries?

UNI FREIBURG

Dictionaries sind als Hashtabellen implementiert:

- Bei der Erzeugung eines Dictionaries wird eine große Tabelle (die Hashtabelle) eingerichtet.
- Jedem Schlüssel wird mit Hilfe einer Hashfunktion ein Hashwert zugeordnet, der als Tabellenindex dient.
- Der zum Schlüssel gehörige Wert wird an dieser Stelle in der Tabelle abgelegt, es sei denn…
- an diesem Index ist bereits ein Eintrag für einen anderen Schlüssel vorhanden: eine Hashfunktion kann unterschiedlichen Schlüsseln den gleichen Hashwert zuordnen (Kollision).

Wie funktionieren Dictionaries?

UNI FREIBURG

Dictionaries sind als Hashtabellen implementiert:

- Bei der Erzeugung eines Dictionaries wird eine große Tabelle (die Hashtabelle) eingerichtet.
- Jedem Schlüssel wird mit Hilfe einer Hashfunktion ein Hashwert zugeordnet, der als Tabellenindex dient.
- Der zum Schlüssel gehörige Wert wird an dieser Stelle in der Tabelle abgelegt, es sei denn…
- an diesem Index ist bereits ein Eintrag für einen anderen Schlüssel vorhanden: eine Hashfunktion kann unterschiedlichen Schlüsseln den gleichen Hashwert zuordnen (Kollision).
- Bei gleichen Hashwerten für verschiedene Schlüssel gibt es eine Spezialbehandlung (z.B. Ablegen des Werts in der nächsten freien Zelle).

Wie funktionieren Dictionaries?

UNI FREIBURG

Dictionaries sind als Hashtabellen implementiert:

- Bei der Erzeugung eines Dictionaries wird eine große Tabelle (die Hashtabelle) eingerichtet.
- Jedem Schlüssel wird mit Hilfe einer Hashfunktion ein Hashwert zugeordnet, der als Tabellenindex dient.
- Der zum Schlüssel gehörige Wert wird an dieser Stelle in der Tabelle abgelegt, es sei denn…
- an diesem Index ist bereits ein Eintrag für einen anderen Schlüssel vorhanden: eine Hashfunktion kann unterschiedlichen Schlüsseln den gleichen Hashwert zuordnen (Kollision).
- Bei gleichen Hashwerten für verschiedene Schlüssel gibt es eine Spezialbehandlung (z.B. Ablegen des Werts in der nächsten freien Zelle).
- Der Zugriff erfolgt trotzdem in (erwarteter) konstanter Zeit.

Eine Hashtabelle bei der Arbeit

Eingabe:

Hashtabelle

Index	Key	Value
0		
1		
2		
3		
4		
5		
6		

Eine Hashtabelle bei der Arbeit

UNI FREIBURG

Eingabe: `('parrot', 'dead')`

Hashtabelle

Index	Key	Value
0		
1		
2		
3		
4		
5		
6		

Eine Hashtabelle bei der Arbeit

UNI FREIBURG

Eingabe: `('parrot', 'dead')`
hash(`'parrot'`)=4

Hashtabelle

Index	Key	Value
0		
1		
2		
3		
4		
5		
6		

Eine Hashtabelle bei der Arbeit

Eingabe:

Hashtabelle

Index	Key	Value
0		
1		
2		
3		
4	`'parrot'`	`'dead'`
5		
6		

Eine Hashtabelle bei der Arbeit

UNI FREIBURG

Eingabe: `('spam', 'tasty')`

Hashtabelle

Index	Key	Value
0		
1		
2		
3		
4	`'parrot'`	`'dead'`
5		
6		

Eine Hashtabelle bei der Arbeit

Eingabe: `('spam', 'tasty')`
hash(`'spam'`)=0

Hashtabelle

Index	Key	Value
0		
1		
2		
3		
4	`'parrot'`	`'dead'`
5		
6		

Eine Hashtabelle bei der Arbeit

Eingabe:

Hashtabelle

Index	Key	Value
0	`'spam'`	`'tasty'`
1		
2		
3		
4	`'parrot'`	`'dead'`
5		
6		

Eine Hashtabelle bei der Arbeit

Eingabe: `('zehn', 10)`

Hashtabelle

Index	Key	Value
0	`'spam'`	`'tasty'`
1		
2		
3		
4	`'parrot'`	`'dead'`
5		
6		

Eine Hashtabelle bei der Arbeit

Eingabe: `('zehn', 10)`
hash(`'zehn'`)=4

Hashtabelle

Index	Key	Value
0	`'spam'`	`'tasty'`
1		
2		
3		
4	`'parrot'`	`'dead'`
5		
6		

Eine Hashtabelle bei der Arbeit

Eingabe: `('zehn', 10)`
hash(`'zehn'`)=4 **Konflikt!**

Hashtabelle

Index	Key	Value
0	`'spam'`	`'tasty'`
1		
2		
3		
4	`'parrot'`	`'dead'`
5		
6		

Eine Hashtabelle bei der Arbeit

Eingabe:

Hashtabelle

Index	Key	Value
0	`'spam'`	`'tasty'`
1		
2		
3		
4	`'parrot'`	`'dead'`
5	`'zehn'`	10
6		

Eine Hashtabelle bei der Arbeit

Anfrage: `'parrot'`

Hashtabelle

Index	Key	Value
0	`'spam'`	`'tasty'`
1		
2		
3		
4	`'parrot'`	`'dead'`
5	`'zehn'`	10
6		

Eine Hashtabelle bei der Arbeit

Anfrage: `'parrot'`
hash(`'parrot'`)=4

Hashtabelle

Index	Key	Value
0	`'spam'`	`'tasty'`
1		
2		
3		
4	`'parrot'`	`'dead'`
5	`'zehn'`	10
6		

Eine Hashtabelle bei der Arbeit

Anfrage: `'parrot'`
hash(`'parrot'`)=4
Ausgabe:`'dead'`

Hashtabelle

Index	Key	Value
0	`'spam'`	`'tasty'`
1		
2		
3		
4	`'parrot'`	`'dead'`
5	`'zehn'`	10
6		

Eine Hashtabelle bei der Arbeit

Anfrage: `'zehn'`

Hashtabelle

Index	Key	Value
0	`'spam'`	`'tasty'`
1		
2		
3		
4	`'parrot'`	`'dead'`
5	`'zehn'`	10
6		

Eine Hashtabelle bei der Arbeit

Anfrage: `'zehn'`
hash(`'zehn'`)=4

Hashtabelle

Index	Key	Value
0	`'spam'`	`'tasty'`
1		
2		
3		
4	`'parrot'`	`'dead'`
5	`'zehn'`	10
6		

Eine Hashtabelle bei der Arbeit

Anfrage: `'zehn'`
hash(`'zehn'`)=4

Hashtabelle

Index	Key	Value
0	`'spam'`	`'tasty'`
1		
2		
3		
4	`'parrot'`	`'dead'`
5	`'zehn'`	10
6		

Eine Hashtabelle bei der Arbeit

Anfrage: `'zehn'`
hash(`'zehn'`)=4
Ausgabe:10

Hashtabelle

Index	Key	Value
0	`'spam'`	`'tasty'`
1		
2		
3		
4	`'parrot'`	`'dead'`
5	`'zehn'`	10
6		

Einige Konsequenzen …

- Schlüssel müssen hash-bar sein und auf Gleichheit getestet werden können.

Einige Konsequenzen …

- Schlüssel müssen hash-bar sein und auf Gleichheit getestet werden können.
- Objekte, die als Schlüssel in einem Dictionary verwendet werden, dürfen nicht verändert werden. Sonst ändert sich der Hashwert und das Objekt wird nicht mehr gefunden.

Veränderliche Dictionary-Keys (1)

UNI FREIBURG

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 (1)

UNI FREIBURG

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?
```

Illegal!

`mydict[mylist]` liefert schon eine Fehlermeldung!

Veränderliche Dictionary-Keys (2)

UNI FREIBURG

- Um solche Problem zu vermeiden, sollten in Python nur *unveränderliche* Objekte, die aus Tupeln, Strings und Zahlen konstruiert sind, als Dictionary-Schlüssel verwendet werden.

Veränderliche Dictionary-Keys (2)

UNI FREIBURG

- Um solche Problem zu vermeiden, sollten in Python nur *unveränderliche* Objekte, die aus Tupeln, Strings und Zahlen konstruiert sind, als Dictionary-Schlüssel verwendet werden.
- Verboten sind also Listen und Dictionaries sowie Objekte, die Listen oder Dictionaries beinhalten bzw deren Attribute veränderlich sind.

Veränderliche Dictionary-Keys (2)

UNI FREIBURG

- Um solche Problem zu vermeiden, sollten in Python nur *unveränderliche* Objekte, die aus Tupeln, Strings und Zahlen konstruiert sind, als Dictionary-Schlüssel verwendet werden.
- Verboten sind also Listen und Dictionaries sowie Objekte, die Listen oder Dictionaries beinhalten bzw deren Attribute veränderlich sind.
- Selbstdefinierte Klassen, deren Instanzen als Dictionary-Schlüssel verwendet werden, sollten als `frozen` definiert werden, sodass die Attribute nach der Initialisierung nicht verändert werden können:

```
@dataclass(frozen=True)
```

Veränderliche Dictionary-Keys (2)

UNI FREIBURG

- Um solche Problem zu vermeiden, sollten in Python nur *unveränderliche* Objekte, die aus Tupeln, Strings und Zahlen konstruiert sind, als Dictionary-Schlüssel verwendet werden.
- Verboten sind also Listen und Dictionaries sowie Objekte, die Listen oder Dictionaries beinhalten bzw deren Attribute veränderlich sind.
- Selbstdefinierte Klassen, deren Instanzen als Dictionary-Schlüssel verwendet werden, sollten als `frozen` definiert werden, sodass die Attribute nach der Initialisierung nicht verändert werden können:

```
@dataclass(frozen=True)
```

- Für die *Werte* sind beliebige Objekte zulässig; die Einschränkung gilt nur für Schlüssel!

Veränderliche Dictionary-Keys (3)

UNI FREIBURG

Python-Interpreter

```
>>> mydict = {("silly", "walk"): [1, 2, 3]}
```

Veränderliche Dictionary-Keys (3)

UNI FREIBURG

Python-Interpreter

```
>>> mydict = {("silly", "walk"): [1, 2, 3]}
>>> mydict[[10, 20]] = "spam"
```

Veränderliche Dictionary-Keys (3)

UNI FREIBURG

Python-Interpreter

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

Veränderliche Dictionary-Keys (3)

UNI FREIBURG

Python-Interpreter

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

Veränderliche Dictionary-Keys (3)

UNI FREIBURG

Python-Interpreter

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

Veränderliche Dictionary-Keys (4)

UNI FREIBURG


```
@dataclass(frozen=True)
class Time():
    hours: int
    minutes: int

morning = Time(6, 30)
noon    = Time(12, 00)

d[morning] = "breakfast"
d[noon]    = "lunch"
```

Dictionaries für Keyword Parameter

UNI FREIBURG

- Eine Funktion kann Keyword Parameter der Form `par=wert` akzeptieren.
- Falls der letzte formale Parameter der Funktion die Form `**kwargs` hat, so akzeptiert die Funktion beliebige Keyword Parameter.
- Im Funktionsrumpf kann `kwargs` wie ein Dictionary verwendet werden.

Python-Interpreter

```
>>> def echo(**kwargs):
...   for k,v in kwargs.items():
...     print(str(k) + " = " + str(v))
...
>>> echo(a=42, b='foo')
a = 42
b = foo
```

UNI FREIBURG

Mengen

Mengen

UNI FREIBURG

- Der Datentyp Menge ist ein Container-Datentyp. Eine Menge enthält (endlich viele) Elemente. Die Reihenfolge der Elemente spielt keine Rolle.

Mengen

UNI FREIBURG

- Der Datentyp Menge ist ein Container-Datentyp. Eine Menge enthält (endlich viele) Elemente. Die Reihenfolge der Elemente spielt keine Rolle.
- Grundoperationen auf dem Datentyp Menge:

Mengen

UNI FREIBURG

- Der Datentyp Menge ist ein Container-Datentyp. Eine Menge enthält (endlich viele) Elemente. Die Reihenfolge der Elemente spielt keine Rolle.
- Grundoperationen auf dem Datentyp Menge:
  - Einfügen eines Elements,

Mengen

UNI FREIBURG

- Der Datentyp Menge ist ein Container-Datentyp. Eine Menge enthält (endlich viele) Elemente. Die Reihenfolge der Elemente spielt keine Rolle.
- Grundoperationen auf dem Datentyp Menge:
  - Einfügen eines Elements,
  - Entfernen eines Elements,

Mengen

UNI FREIBURG

- Der Datentyp Menge ist ein Container-Datentyp. Eine Menge enthält (endlich viele) Elemente. Die Reihenfolge der Elemente spielt keine Rolle.
- Grundoperationen auf dem Datentyp Menge:
  - Einfügen eines Elements,
  - Entfernen eines Elements,
  - Test ob Element enthalten ist.

Mengen

UNI FREIBURG

- Der Datentyp Menge ist ein Container-Datentyp. Eine Menge enthält (endlich viele) Elemente. Die Reihenfolge der Elemente spielt keine Rolle.
- Grundoperationen auf dem Datentyp Menge:
  - Einfügen eines Elements,
  - Entfernen eines Elements,
  - Test ob Element enthalten ist.
- Voraussetzungen

Mengen

UNI FREIBURG

- Der Datentyp Menge ist ein Container-Datentyp. Eine Menge enthält (endlich viele) Elemente. Die Reihenfolge der Elemente spielt keine Rolle.
- Grundoperationen auf dem Datentyp Menge:
  - Einfügen eines Elements,
  - Entfernen eines Elements,
  - Test ob Element enthalten ist.
- Voraussetzungen
  - Elemente müssen hash-bar sein!

Mengen

UNI FREIBURG

- Der Datentyp Menge ist ein Container-Datentyp. Eine Menge enthält (endlich viele) Elemente. Die Reihenfolge der Elemente spielt keine Rolle.
- Grundoperationen auf dem Datentyp Menge:
  - Einfügen eines Elements,
  - Entfernen eines Elements,
  - Test ob Element enthalten ist.
- Voraussetzungen
  - Elemente müssen hash-bar sein!
  - Elemente müssen auf Gleichheit getestet werden können!

Mengen

UNI FREIBURG

- Der Datentyp Menge ist ein Container-Datentyp. Eine Menge enthält (endlich viele) Elemente. Die Reihenfolge der Elemente spielt keine Rolle.
- Grundoperationen auf dem Datentyp Menge:
  - Einfügen eines Elements,
  - Entfernen eines Elements,
  - Test ob Element enthalten ist.
- Voraussetzungen
  - Elemente müssen hash-bar sein!
  - Elemente müssen auf Gleichheit getestet werden können!
  - Elemente sollten unveränderlich (immutable) sein!

Mengen

UNI FREIBURG

- Der Datentyp Menge ist ein Container-Datentyp. Eine Menge enthält (endlich viele) Elemente. Die Reihenfolge der Elemente spielt keine Rolle.
- Grundoperationen auf dem Datentyp Menge:
  - Einfügen eines Elements,
  - Entfernen eines Elements,
  - Test ob Element enthalten ist.
- Voraussetzungen
  - Elemente müssen hash-bar sein!
  - Elemente müssen auf Gleichheit getestet werden können!
  - Elemente sollten unveränderlich (immutable) sein!
- Einfügen und Entfernen sind idempotent; eine Menge kann also nicht dasselbe Element ‚mehrmals' enthalten ($\Rightarrow$ Multimenge).

Mengen (Fortsetzung)

UNI FREIBURG

- Mengen können duch Listen implementiert werden. Dann ist die mittlere Zeit ein Element zu finden, *linear* in der Größe der Menge.

Mengen (Fortsetzung)

UNI FREIBURG

- Mengen können duch Listen implementiert werden. Dann ist die mittlere Zeit ein Element zu finden, *linear* in der Größe der Menge.
- Mengen können duch Binärbäume implementiert werden. Dann ist die mittlere Zeit ein Element zu finden *logarithmisch* in der Größe der Menge und wir brauchen eine Ordnung auf den Elementen.

Mengen (Fortsetzung)

UNI FREIBURG

- Mengen können duch Listen implementiert werden. Dann ist die mittlere Zeit ein Element zu finden, *linear* in der Größe der Menge.
- Mengen können duch Binärbäume implementiert werden. Dann ist die mittlere Zeit ein Element zu finden *logarithmisch* in der Größe der Menge und wir brauchen eine Ordnung auf den Elementen.
- Mengen können durch Dicts implementiert werden, wobei die Elemente die Schlüssel sind und der Wert immer `None` ist (konstante Zugriffszeit).

Mengen (Fortsetzung)

- Mengen können duch Listen implementiert werden. Dann ist die mittlere Zeit ein Element zu finden, *linear* in der Größe der Menge.
- Mengen können duch Binärbäume implementiert werden. Dann ist die mittlere Zeit ein Element zu finden *logarithmisch* in der Größe der Menge und wir brauchen eine Ordnung auf den Elementen.
- Mengen können durch Dicts implementiert werden, wobei die Elemente die Schlüssel sind und der Wert immer `None` ist (konstante Zugriffszeit).
- Es gibt spezielle Datentypen für Mengen in Python, die alle Mengenoperationen unterstützen.

Mengen (Fortsetzung)

UNI FREIBURG

- Mengen können duch Listen implementiert werden. Dann ist die mittlere Zeit ein Element zu finden, *linear* in der Größe der Menge.
- Mengen können duch Binärbäume implementiert werden. Dann ist die mittlere Zeit ein Element zu finden *logarithmisch* in der Größe der Menge und wir brauchen eine Ordnung auf den Elementen.
- Mengen können durch Dicts implementiert werden, wobei die Elemente die Schlüssel sind und der Wert immer `None` ist (konstante Zugriffszeit).
- Es gibt spezielle Datentypen für Mengen in Python, die alle Mengenoperationen unterstützen.
- Sie sind ebenfalls mit Hilfe von Hashtabellen realisiert.

Mengen: `set` und `frozenset`

- Voraussetzung: Mengenelemente müssen *hashbar* sein (wie die Schlüssel bei Dictionaries).

Mengen: `set` und `frozenset`

UNI FREIBURG

- Voraussetzung: Mengenelemente müssen *hashbar* sein (wie die Schlüssel bei Dictionaries).
- Es gibt veränderliche Mengen (`set`) und unveränderliche Mengen (`frozenset`):

Mengen: `set` und `frozenset`

- Voraussetzung: Mengenelemente müssen *hashbar* sein (wie die Schlüssel bei Dictionaries).
- Es gibt veränderliche Mengen (`set`) und unveränderliche Mengen (`frozenset`):
  - `frozenset`s sind unveränderlich $\rightsquigarrow$ hashbar,

Mengen: `set` und `frozenset`

- Voraussetzung: Mengenelemente müssen *hashbar* sein (wie die Schlüssel bei Dictionaries).
- Es gibt veränderliche Mengen (`set`) und unveränderliche Mengen (`frozenset`):
  - `frozenset`s sind unveränderlich $\rightsquigarrow$ hashbar,
  - Insbesondere können `frozenset`s auch als Elemente von `set`s und `frozenset`s sowie als Schlüssel von Dictionaries 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

UNI FREIBURG

- `{elem1, ..., elemN}`
  Erzeugt die veränderliche Menge $\{\texttt{elem1}, \ldots, \texttt{elemN}\}$.

Konstruktion von Mengen

- `{elem1, ..., elemN}`
  Erzeugt die veränderliche Menge $\{\texttt{elem1}, \ldots, \texttt{elemN}\}$.
- `set()`
  Erzeugt eine veränderliche leere Menge.

Konstruktion von Mengen

- `{elem1, ..., elemN}`
  Erzeugt die veränderliche Menge $\{\texttt{elem1}, \ldots, \texttt{elemN}\}$.
- `set()`
  Erzeugt eine veränderliche leere Menge.
- `set(iterable)`
  Erzeugt eine veränderliche Menge aus Elementen von `iterable`.

Konstruktion von Mengen

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

Konstruktion von Mengen

- `{elem1, ..., elemN}`
  Erzeugt die veränderliche Menge $\{\texttt{elem1}, \ldots, \texttt{elemN}\}$.
- `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`.

Konstruktion von Mengen

UNI FREIBURG

- `{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`.
- Das `iterable` darf nur *hashbare* Objekte (z.B. keine Listen!) enthalten.

Konstruktion von Mengen: Beispiele (1)

UNI FREIBURG

Python-Interpreter

```
>>> set("spamspam")
```

Konstruktion von Mengen: Beispiele (1)

UNI FREIBURG

Python-Interpreter

```
>>> set("spamspam")
{'a', 'p', 's', 'm'}
```

Konstruktion von Mengen: Beispiele (1)

UNI FREIBURG

Python-Interpreter

```
>>> set("spamspam")
{'a', 'p', 's', 'm'}
>>> frozenset("spamspam")
```

Konstruktion von Mengen: Beispiele (1)

UNI FREIBURG

Python-Interpreter

```
>>> set("spamspam")
{'a', 'p', 's', 'm'}
>>> frozenset("spamspam")
frozenset({'a', 'p', 's', 'm'})
```

Konstruktion von Mengen: Beispiele (1)

UNI FREIBURG

Python-Interpreter

```
>>> set("spamspam")
{'a', 'p', 's', 'm'}
>>> frozenset("spamspam")
frozenset({'a', 'p', 's', 'm'})
>>> set(["spam", 1, [2, 3]])
```

Konstruktion von Mengen: Beispiele (1)

UNI FREIBURG

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'
```

Konstruktion von Mengen: Beispiele (1)

UNI FREIBURG

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)))
```

Konstruktion von Mengen: Beispiele (1)

UNI FREIBURG

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'}
```

Konstruktion von Mengen: Beispiele (1)

UNI FREIBURG

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})
```

Konstruktion von Mengen: Beispiele (1)

UNI FREIBURG

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"])
```

Konstruktion von Mengen: Beispiele (2)

Python-Interpreter

```
>>> s = set(["jam", "spam"])
>>> set([1, 2, 3, s])
```

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'
```

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)])
```

Konstruktion von Mengen: Beispiele (2)

UNI FREIBURG

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

UNI FREIBURG

- `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:`
  Iteration über Mengen.
- `s.copy()`
  Liefert eine (flache) Kopie der Menge `s`.

Mengen: Einfügen und Entfernen von Elementen

UNI FREIBURG

- `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`.

Viele weitere Operationen

- `union`, `intersection`, `difference`, `symmetric_difference`
- <=, < (Test auf Teilmenge)
- ==, `!=` (Test auf Mengengleichheit)

Zusammenfassung

- `dicts` sind Abbildungen von Schlüsseln auf Werte.

Zusammenfassung

- `dicts` sind Abbildungen von Schlüsseln auf Werte.
- Der Zugriff auf Elemente von `dicts` erfolgt (fast) in konstanter Zeit

Zusammenfassung

- `dicts` sind Abbildungen von Schlüsseln auf Werte.
- Der Zugriff auf Elemente von `dicts` erfolgt (fast) in konstanter Zeit
- `dicts` sind veränderlich.

Zusammenfassung

- `dicts` sind Abbildungen von Schlüsseln auf Werte.
- Der Zugriff auf Elemente von `dicts` erfolgt (fast) in konstanter Zeit
- `dicts` sind veränderlich.
- Die Typen `set` und `frozenset` implementieren Mengen mit allen erwarteten Operationen.

Zusammenfassung

UNI FREIBURG

- `dict`s sind Abbildungen von Schlüsseln auf Werte.
- Der Zugriff auf Elemente von `dict`s erfolgt (fast) in konstanter Zeit
- `dict`s sind veränderlich.
- Die Typen `set` und `frozenset` implementieren Mengen mit allen erwarteten Operationen.
- `set`s sind veränderliche Strukturen, `frozenset`s sind nicht veränderlich.