Übungen 3: Erklärungen und Hilfestellungen

Allgemeines zu Dictionary-Methoden

In Python gibt es verschiedenen Datentypen: Strings, Listen, Dictionaries usw. Diese Typen unterscheiden sich u.a. durch die sie charakterisierenden Methoden. Wenn neue Typen aus bestehenden Typen abgeleitet werden, erben diese alle Methoden des Supertyps. Es bietet sich für die Aufgaben 1- 11, in denen es ja darum geht, einen aus **`dict`** abgeleiteten Typen namens **`Lexicon`** zu definieren, also an, sich zunächst einmal die Methoden, sprich die typ- bzw. klassenspezifischen Funktionen von **`dict`** anzusehen. Diese können dann

1. auf alle Objektinstanzen von **`Lexicon`** angewendet und
2. bei der Definition von **`Lexicon`**-spezifischen Methoden eingesetzt werden.

Die Attribute von **`dict`**:

```
>>> dir(dict)
['__class__', '__cmp__', '__contains__', '__delattr__', '__delitem__',
'__doc__', '__eq__', '__ge__', '__getattribute__', '__getitem__', '__gt__',
'__hash__', '__init__', '__iter__', '__le__', '__len__', '__lt__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__setitem__', '__str__', 'clear', 'copy', 'fromkeys', 'get', 'has_key',
'items', 'iteritems', 'iterkeys', 'itervalues', 'keys', 'pop', 'popitem',
'setdefault', 'update', 'values']
```

Nicht alles, was in dieser Liste zu finden ist, ist eine Methode, und nicht alle Methoden werden erläutert. Uns geht es um einige grundlegende Erklärungen, und dafür betrachten wir die farbig unterlegten Attribute.

Wir stellen zunächst fest, dass bestimmte Methoden die Form **`__methodenname__`** haben, andere dagegen nur **`methodenname`**. Die grün unterlegten Methoden gehören zur ersten Sorte, und obwohl Sie vielleicht glauben, diese Methoden noch nicht zu kennen, haben Sie sie bereits häufig angewendet, beispielsweise in Übung 2 – allerdings in alternativer Notation. Sehen wir uns Ausschnitte davon an, was Python uns zu diesen Methoden mitteilt:

```
>>> help(dict.__getitem__)
x.__getitem__(y) <==> x[y]

>>> help(dict.__contains__)
D.__contains__(k) -> True if D has a key k, else False

>>> help(dict.__delitem__)
x.__delitem__(y) <==> del x[y]

>>> help(dict.__setitem__)
x.__setitem__(i, y) <==> x[i]=y
```

Der folgende Screenshot zeigt die Äquivalenzen zu den Ihnen bekannten Schreibweisen:

```
>>> lex = {'a':'1','b':'2','c':'3'}

>>> lex.__getitem__('b')
'2'
>>> lex['b']
'2'

>>> lex.__contains__('b')
True
>>> lex.has_key('b')
True
>>> 'b' in lex
True
```

```
>>> lex.__setitem__('d','4')
>>> lex
{'a': '1', 'c': '3', 'b': '2', 'd': '4'}
>>> lex.__delitem__('d')
>>> lex
{'a': '1', 'c': '3', 'b': '2'}

>>> lex['d'] = '4'
>>> lex
{'a': '1', 'c': '3', 'b': '2', 'd': '4'}
>>> del lex['d']
>>> lex
{'a': '1', 'c': '3', 'b': '2'}
```

Wie Sie hier sehen können, sind einige der **`__methodenname__`**-Methoden insofern grundlegend, als sie die Basis für andere Methoden mit vereinfachter Eingabe liefern (Stichwort *syntactic sugar*). So ist **`__getitems__`** beispielsweise die Grundlage für **`get()`** und die Referenzierung des *values* über den *key* als Indexausdruck wie in **`lex[key] == value`**. Die Methode **`__contains__`** ist die Grundlage für **`has_key()`** und **`key in dict`** usw.

[1] Da es **`has_key()`** in Python 3.0 nicht mehr geben wird, sondern nur noch **`key in dict`**, sollten wir bereits jetzt letzeres verwenden.

Für unsere Aufgaben sind allerdings nur die blau unterlegten Methoden relevant (und von diesen auch nicht alle). Über die **help**-Funktion können Sie sich anzeigen lassen, was die Methoden bewirken (hier wieder Ausschnitte):

```
>>> help(lex.clear)
    D.clear() -> None.  Remove all items from D.
>>> help(lex.get)
    D.get(k[,d]) -> D[k] if k in D, else d.
>>> help(lex.fromkeys)
    dict.fromkeys(S[,v]) -> New dict with keys from S and values equal to v.
```

Zugegeben: diese Hilfen sind für Einsteiger wenig hilfreich. Um z.B. zu verstehen, was die Methode **fromkeys()** bewirkt, muss man wissen, dass die Variable 'S' in der Erläuterung für *sequence* steht. Im Nachfolgenden sehen Sie deshalb detailliertere Beschreibungen derjenigen Methoden (die **dict**-Methoden **update()** und **fromkeys()**, die String-Methode **ljust()** und die *built-in-function* **zip**, die auf Sequenzen angewendet werden kann), die für die jeweilige Aufgabe relevant sind.

Hilfestellungen zu einzelnen Aufgaben

Aufgabe 1 : `pairlist()`

Verwenden Sie hier die Funktion **zip()** und die Methode **split()**

zip()

Nimmt zwei Sequenzen (z.B. Zeichenketten oder Listen) als Argumente und gibt eine Tupelliste zurück, deren Elemente sich aus den Elemente der Sequenzen konstituieren derart, dass der Reihe nach aus den Elementen der ersten und der zweiten Sequenz Paare gebildet werden.

```
>>> zip([1,2,3,4],[5,6,7,8])
[(1, 5), (2, 6), (3, 7), (4, 8)]
>>> zip('a,@!','b:*?')
[('a', 'b'), (',', ':'), ('@', '*'), ('!', '?')]
```

Aufgabe 2: die Klasse `Lexicon`

Es ist für diese Klasse nicht nötig, eine Initialisierungsmethode zu definieren.

Aufgabe 3: `merge()`

Wenn Sie ein Lexikon **wb** um Daten aus einem weiteren Lexikon **wb1** erweitern wollen, könnten Sie dafür die **dict**-Methode **update()** verwenden. Im Falle von *key*-Identität bei unterschiedlichen *values* werden mit **update()** aber alle bestehenden *key-value*-Paare in **wb** durch die entsprechenden Paare in **wb1** überschrieben.

update()

Nimmt ein Argument, das in ein Dictionary konvertierbar sein muss, und fügt dieses dem bestehenden Lexikon hinzu.

```
>>> lex          #Name des bestehenden Dictionaries
{'light': 'adj', 'the': 'det', 'dog': 'noun'}
>>> lex.update({'my':'det','cat':'noun'})
>>> lex          #Upgedatetes Dictionary
{'light': 'adj', 'the': 'det', 'my': 'det', 'dog': 'noun', 'cat': 'noun'}
```

In den Fällen, in denen derselbe *key* mit unterschiedlichen *values* vertreten ist, wird das ursprüngliche *key-value*-Paar mit dem neuen überschrieben. Wie Sie oben sehen können, hat der *key* **light** in **lex** den *value* **adj**, was bei entsprechendem Argument von **update** aber überschrieben wird:

```
>>> lex.update({'light':'noun'})
>>> lex
{'light': 'noun', 'the': 'det', 'my': 'det', 'dog': 'noun', 'cat': 'noun'}
```

Die Methode **update()** lässt sich wie folgt emulieren:

```
>>> lex = {'the':'det','dog':'noun','light':'adj'}
>>> def update_emulation(D,D1):
      for key in D1:
         D[key] = D1[key]
>>> update_emulation(lex, {'in':'prep'})
>>> lex
{'light': 'adj', 'the': 'det', 'dog': 'noun', 'in': 'prep'}
```

Aufgabe 4: `from_wordlist()`

`fromkeys()`

Nimmt zwei Argumente: eine Sequenz (z.B. eine Liste) von *keys* und den *value*, den diese *keys* erhalten sollen. Diese Methode ist günstig, wenn man 'auf einen Schlag' ein Dictionary erstellen möchte mit *keys*, die denselben *value* haben.

```
>>> newlex = dict.fromkeys(['rather','very','awfully'],'adv')
>>> newlex
{'rather': 'adv', 'very': 'adv', 'awfully': 'adv'}
```

Man kann `fromkeys()` gut in Kombination mit `update` nutzen, um ein bestehendes Dictionary zu erweitern. Auch dabei muss man – wie im Screenshot zu sehen ist – `fromkeys()` als `dict`-Methode kennzeichnen:

```
>>> newlex.update(dict.fromkeys(['and','or','but'],'conj'))
>>> newlex
{'and': 'conj', 'very': 'adv', 'rather': 'adv', 'or': 'conj', 'but': 'conj',
'awfully': 'adv'}
```

Auch `fromkeys()` lässt sich emulieren:

```
>>> def fromkeys_emulation(d,keysequence,value):
    for key in keysequence:
          d[key] = value
>>> fromkeys_emulation(newlex,['kicks','sing'],'verb')
>>> newlex
{'and': 'conj', 'very': 'adv', 'kicks': 'verb', 'sings': 'verb', 'rather':
'adv', 'or': 'conj', 'but': 'conj', 'awfully': 'adv'}
```

Aufgabe 5: `get_cat()`

Sehen Sie bei Bedarf die Lösungen zu *Übungen 2* : Input/Output, Teil 1: Tastatureingaben

Aufgabe 6: `get_cat_table()`

Sehen Sie bei Bedarf den Text über *List Comprehension.*

Überlegen Sie genau, was hier zurückgeben wird und in welchen Schritten Sie dieses Objekt beschreiben.

Input ist ein Dictionary wie das folgende:

```
{'boy': 'noun', 'cried': 'verb', 'slept': 'verb', 'the': 'det', 'my': 'det', 'girl': 'noun'}
```

Output ist ebenfalls ein Dictionary:

```
{'noun': ['boy', 'girl'], 'verb': ['cried', 'slept'], 'det': ['the', 'my']}
```

Sie können diese Methode mit einer *List Comprehension* definieren, deren Outputfunktion selber eine *List Comprehension* ist. Als Hilfestellungen die folgenden Hinweise:

Hinweis 1: Sie können aus jedem Dictionary per *List Comprehension* eine Tupelliste erzeugen, in der das erste Element des Tupels der *value* des jeweiligen Dictionary-Items ist; das zweite der *key*:

```
>>> X = {'a':1, 'b':2, 'c':3, 'd':4}
>>> [(value,key) for (key,value) in X.items()]
[(1, 'a'), (3, 'c'), (2, 'b'), (4, 'd')]
```

Hinweis 2: Um eine Liste aus Tupeln in ein Dictionary zu konvertieren, können Sie z.B. `dict()` verwenden:

```
>>> dict([('1', 'a'), ('3', 'c'), ('2', 'b'), ('4', 'd')])
{'1': 'a', '3': 'c', '2': 'b', '4': 'd'}
```

Hinweis 3: Wenn Sie ein Dictionary mit *key-value*-Paaren haben, können Sie per *List Comprehension* wie folgt eine Liste von *keys* erzeugen, die den gleichen *value* haben:

```
>>> X
{'boy': 'noun', 'the': 'det', 'my': 'det', 'girl': 'noun', 'cat': 'noun'}
>>> [key for key in X if X[key] == 'noun']
['boy', 'girl', 'cat']
>>> [key for key in X if X[key] == 'det']
['the', 'my']
```

Hinweis 4: Die *built-in-function* `set()` gibt für Strings, Listen, Tupel etc. Mengen der Elemente zurück:

```
>>> set('abacadbebf')
set(['a', 'c', 'b', 'e', 'd', 'f'])
```

Aufgabe 7: `print_table()`

`ljust()`

Das 'l' in `ljust()` steht für *left*, entsprechend gibt es auch ein `rjust()` und ein `center()`. Alle drei sind String-Methoden, die die Ausrichtung von Zeichenketten beim Drucken steuern. Das Argument ist ein `int`, das die Feldbreite angibt, innerhalb derer die Zeichenkette ausgerichtet ist:

```
>>> liste = ['aa','bbbb','cccccc','dddddddd']

>>> for element  in liste:
    print "%s" %element.ljust(10)
aa
bbbb
cccccc
dddddddd

>>> for element in liste:
    print "%s" %element.rjust(10)
        aa
      bbbb
    cccccc
  dddddddd

>>> for element in liste:
    print "%s" %element.center(10)
    aa
   bbbb
  cccccc
 dddddddd
```

Aufgabe 8: `print_cat_table()`

Wenn Sie, wie in der Sitzung am 04.09. kurz gefragt, folgendes eingeben:

```
>>> wb.get_cat_table.print_table()
```

erhalten Sie eine Fehlermeldung folgenden Inhaltes:

```
AttributeError: 'function' object has no attribute 'print_table'
```

So, wie es hier steht, ist ausgedrückt, dass eine Methode auf eine Methode angewendet werden soll. Da letzere keine Attribute hat (also auch keine Methoden) kann das so nicht gehen. Geben Sie stattdessen ein

```
>>> wb.get_cat_table().print_table()
```

funktioniert es, da Sie hier die Methode auf das Ergebnis der Methodenanwendung anwenden. Voraussetzung ist allerdings, dass dieses als Instanz des Typs `Lexicon` ausgewiesen ist, sonst heißt es

```
AttributeError: 'list' <str, dict> object has no attribute 'get_cat_table'
```

Aufgabe 9: `statistics()`

Sie sollten auch hier für die Ausgabe `ljust()` verwenden.

Aufgaben 10 & 11: `save()` & `load()`

Sehen Sie bei Bedarf die Lösungen von *Übungen 2* : Input/Output, Teil 2: Dateien schreiben und lesen