5. Datenstrukturen

Dieses Kapitel beschreibt einige Dinge, die schon vorkamen, detaillierter und stellt auch ein paar neue Dinge vor.

5.1. Mehr zu Listen

Der Datentyp `list` hat noch ein paar andere Methoden. Hier sind alle Methoden von Listenobjekten:

`list.append`(x)

Hängt ein neues Element an das Ende der Liste an. Äquivalent zu `a[len(a):] = [x]`.

`list.extend`(L)

Erweitert die Liste, indem es alle Elemente der gegebenen Liste anhängt. Äquivalent zu `a[len(a):] = L`.

`list.insert`(i, x)

Fügt ein Element an der gegebenen Position ein. Das erste Argument ist der Index des Elements, vor dem eingefügt werden soll, so fügt `a.insert(0, x)` am Anfang der Liste ein und `a.insert(len(a), x)` ist äquivalent zu `a.append(x)`.

`list.remove`(x)

Entfernt das erste Element, dessen Wert x ist. Es gibt eine Fehlermeldung, wenn solch ein Element nicht existiert.

`list.pop`([i])

Entfernt das Element an der gegebenen Position und gibt es zurück. Ist kein Index gegeben, entfernt `a.pop()` das letzte Element der Liste und gibt es zurück. (Die eckigen Klammern um das i in der Methodensignatur herum, zeigen an, dass das Argument optional ist und nicht, dass man dort eckige Klammern eintippen sollte. Du wirst diese Notation des öfteren in der Referenz der Pythonbibliothek sehen.

`list.index`(x)

Gibt den Index des ersten Elements in der Liste zurück, dessen Wert x ist. Es gibt eine Fehlermeldung, wenn solch ein Element nicht existiert.

`list.count`(x)

Gibt zurück, wie oft x in der Liste vorkommt.

`list.sort`()

Sortiert die Elemente der Liste im selben Exemplar (in place).

`list.reverse()`

Kehrt die Reihenfolge der Listenelemente im selben Exemplar (in place) um.

Ein Beispiel, das die meisten Methoden von Listen benutzt:

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> print(a.count(333), a.count(66.25), a.count('x'))
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]
>>> a.sort()
>>> a
[-1, 1, 66.25, 333, 333, 1234.5]
```

Vielleicht ist dir aufgefallen, dass bei Methoden wie `insert`, `remove` oder `sort`, die die Liste verändern, keinen Rückgabewert gedruckt wird – sie geben `None` zurück. [1] Dies ist ein Designprinzip für alle veränderlichen Datenstrukturen in Python.

5.1.1. Benutzung von Listen als Stack

Die Methoden von Listen, machen es sehr einfach eine Liste als Stapel (Stack) zu benutzen, bei dem das zuletzt hinzugekommene als Erstes abgerufen wird (“last-in, first-out”). Um ein Element auf den Stack zu legen, benutzt man **`append()`**. Um ein Element abzurufen, benutzt man **`pop()`** ohne expliziten Index. Zum Beispiel:

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

5.1.2. Benutzung von Listen als Queue

Listen lassen sich auch bequem als Schlange (Queue) benutzen, wo das zuerst hinzugekommene Element auch zuerst abgerufen wird (“first-in, first-out”). Allerdings sind Listen nicht effizient für diesen Zweck. Während **`append()`** und **`pop()`** am Ende der Liste

schnell sind, sind `insert()` und `pop()` am Anfang der Liste langsam (da alle anderen Elemente um eine Stelle verschoben werden muessen).

Um eine Queue zu implementieren benutzt man `collections.deque`, die so entworfen wurde, um beidseitig schnelle appends und pops bereitzustellen. Zum Beispiel:

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # Terry kommt an
>>> queue.append("Graham")          # Graham kommt an
>>> queue.popleft()                 # Der Erste geht jetzt
'Eric'
>>> queue.popleft()                 # Der Zweite geht jetzt
'John'
>>> queue                           # Verbleibende Schlange in der
>>>                                 # Reihenfolge der Ankunft
deque(['Michael', 'Terry', 'Graham'])
```

5.1.3. List Comprehensions

List Comprehensions bieten einen prägnanten Weg, um Listen zu erzeugen. Übliche Anwendungen sind solche, in denen man Listen erstellt, in denen jedes Element das Ergebnis eines Verfahrens ist, das auf jedes Mitglied einer Sequenz oder einem iterierbaren Objekt angewendet wird oder solche, in denen eine Teilfolge von Elementen, die eine bestimmte Bedingung erfüllen, erstellt wird.

Zum Beispiel, wenn wir eine Liste von Quadratzahlen erstellen wollen, wie:

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Dann können wir das gleiche mit:

```
squares = [x**2 for x in range(10)]
```

erreichen. Dies ist auch äquivalent zu `squares = list(map(lambda x: x**2, range(10)))`, aber es ist kürzer und lesbarer.

Jede List Comprehension besteht aus eckigen Klammern, die einen Ausdruck gefolgt von einer `for`-Klausel, enthalten. Danach sind beliebig viele `for`- oder `if`-Klauseln zulässig. Das Ergebnis ist eine neue Liste, deren Elemente durch das Auswerten des Ausdrucks im Kontext der `for`- und `if`-Klauseln, die darauf folgen, erzeugt werden. Zum Beispiel kombiniert diese List Comprehension die Elemente zweier Listen, wenn sie nicht gleich sind:

```
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

was äquivalent zu folgendem ist:

```
>>> combs = []
```

```
>>> for x in [1,2,3]:
...     for y in [3,1,4]:
...         if x != y:
...             combs.append((x, y))
...
>>> combs
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Beachte, dass die Reihenfolge der `for` und `if` Anweisungen in beiden Codestücken gleich ist.

Falls der Ausdruck ein Tupel ist (z.B. `(x, y)` im vorigen Beispiel), muss er geklammert werden.

```
>>> vec = [-4, -2, 0, 2, 4]
>>> # erzeuge eine neue Liste mit den verdoppelten Werten
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # filtere die negativen Zahlen heraus
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> # wende eine Funktion auf alle Elemente an
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # rufe eine Methode auf allen Elementen auf
>>> freshfruit = ['  banana', '  loganberry ', 'passion fruit  ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> # erstelle eine Liste von 2-Tupeln der Form (zahl, quadrat)
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # das Tupel muss geklammert werden, sonst wird ein Fehler erzeugt
>>> [x, x**2 for x in range(6)]
  File "<stdin>", line 1, in ?
    [x, x**2 for x in range(6)]
               ^
SyntaxError: invalid syntax
>>> # verflache eine Liste durch eine LC mit zwei 'for'
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

List Comprehensions können auch komplexe Ausdrücke und verschachtelte Funktionen enthalten:

```
>>> from math import pi
>>> [str(round(pi, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

5.1.4. Verschachtelte List Comprehensions

Der erste Ausruck in einer List Comprehension kann ein beliebiger Ausdruck sein, auch andere List Comprehensions.

Beim folgenden Beispiel wird eine 3x4 Matrix mit drei Listen der Länge vier implementiert:

```
>>> matrix = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]
```

Die folgende List Comprehension vertauscht Listen und Spalten:

```
>>> [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Wie wir im vorherigen Abschnitt gesehen haben, wird die verschachtelte List Comprehension im Kontext des nachfolgenden `for` ausgewertet, damit ist das Beispiel äquivalent zu folgendem:

```
>>> transposed = []
>>> for i in range(4):
...     transposed.append([row[i] for row in matrix])
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

was wiederum gleich zu folgendem ist:

```
>>> transposed = []
>>> for i in range(4):
...     # the following 3 lines implement the nested listcomp
...     transposed_row = []
...     for row in matrix:
...         transposed_row.append(row[i])
...     transposed.append(transposed_row)
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

In Produktivcode sollte man aber eingebaute Funktionen komplexen Anweisungen vorziehen. Die Funktion `zip()` würde in diesem Fall gute Dienste leisten:

```
>>> list(zip(*mat))
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
```

Details zum Sternchen sind unter Argumentlisten auspacken zu finden.

5.2. Die `del` -Anweisung

Es gibt einen Weg ein Listenelement durch seinen Index, statt durch seinen Wert zu löschen: Die `del` -Anweisung. Sie unterscheidet sich von der `pop()` -Methode, da sie keinen Wert zurückgibt. Die `del` -Anweisung kann auch dazu benutzt werden Abschnitte einer Liste zu löschen oder sie ganz zu leeren (was wir vorhin durch die Zuweisung einer leeren Liste an einen Abschnitt getan haben). Zum Beispiel:

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

`del` kann auch dazu benutzt werden, ganze Variablen zu löschen:

```
>>> del a
```

Danach den Namen `a` zu referenzieren führt zu einer Fehlermeldung (zumindest bis dem Namen ein anderer Wert zugewiesen wird). Später werden werden wir noch andere Einsatzmöglichkeiten besprechen.

5.3. Tupel und Sequenzen

Wir haben gesehen, dass Listen und Zeichenketten viele Eigenschaften, wie Slicing und Indizierung, gemein haben. Beide sind Exemplare von Sequenzdatentypen (siehe Sequence Types). Da sich Python weiterentwickelt können auch noch andere Sequenzdatentypen hinzukommen. Es gibt aber noch einen weiteren standardmäßigen Sequenzdatentyp: Das Tupel.

Ein Tupel besteht aus mehreren Werten, die durch Kommas von einander getrennt sind, beispielsweise:

```
>>> t = 12345, 54321, 'Hallo!'
>>> t[0]
12345
>>> t
(12345, 54321, 'Hallo!')
>>> # Tupel können verschachtelt werden:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'Hallo!'), (1, 2, 3, 4, 5))
>>> # Tupel sind unveränderlich:
... t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # aber sie können veränderliche Objekte enthalten:
... v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

Wie man sehen kann, werden die ausgegebenen Tupel immer von Klammern umgeben, sodass verschachtelte Tupel richtig interpretiert werden. Sie können mit oder ohne Klammern eingegeben werden, obwohl Klammern trotzdem sehr oft benötigt werden (wenn das Tupel Teil eines größeren Ausdrucks ist). Es ist nicht möglich den individuellen Elementen etwas zuzuweisen, aber es ist möglich Tupel zu erstellen, die veränderliche Objekte, wie beispielsweise Listen, enthalten.

Obwohl Tupel ähnlich zu Listen erscheinen, werden sie oft an verschiedenen Situationen und zu einem anderen Zweck verwendet. Tupel sind unveränderlich und enthalten üblicherweise eine heterogene Sequenz von Elementen, auf die über unpacking (siehe unten) oder über einen Index (oder sogar über Attribute im Fall von `namedtuples`) zugegriffen wird. Lisen sind veränderlich und ihre Elemente sind üblicherweise homogen und der Zugriff erfolgt durch das iterieren über die Liste.

Ein spezielles Problem ergibt sich in der Darstellung von Tupeln, die 0 oder 1 Elemente haben: Die Syntax hat ein paar Eigenheiten um das Problem zu lösen. Leere Tupel lassen sich mit einem leeren Klammerpaar darstellen und ein Tupel mit einem Element wird erstellt, indem dem Wert ein Komma nachgestellt wird, es reicht jedoch nicht, das Element nur in Klammern zu schreiben. Hässlich aber effektiv. Zum Beispiel:

```
>>> empty = ()
>>> singleton = 'Hallo',    # <-- das angehängte Komma nicht vergessen
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('Hallo',)
```

Die Anweisung `t = 12345, 54321, 'Hallo!'` ist ein Beispiel für das Tupel packen (tuple packing): Die Werte `12345`, `54321`, `'Hallo!'` werden zusammen in ein Tupel gepackt. Das Gegenteil ist ebenso möglich:

```
>>> x, y, z = t
```

Das wird passenderweise Sequenz auspacken (sequence unpacking) genannt und funktioniert mit jeder Sequenz auf der rechten Seite der Zuweisung. Die Anzahl der Namen auf der linken Seite muss genauso groß sein, wie die Länge der Sequenz. Eine Mehrfachzuweisung ist eigentlich nur eine Kombination von Tupel packen und dem Auspacken der Sequenz.

5.4. Mengen

Python enthält auch einen Datentyp für Mengen (sets). Eine Menge ist eine ungeordnete Sammlung ohne doppelte Elemente. Sie werden vor allem dazu benutzt, um zu testen, ob ein Element in der Menge vertreten ist und doppelte Einträge zu beseitigen. Mengenobjekte unterstützen ebenfalls mathematische Operationen wie Vereinigungsmenge, Schnittmenge, Differenz und symmetrische Differenz.

Geschweifte Klammern oder die Funktion `set()` können dazu genutzt werden Mengen zu erzeugen. Wichtig: Um eine leere Menge zu erzeugen muss man `set()` benutzen, `{}` ist dagegen nicht möglich. Letzteres erzeugt ein leeres Dictionary, eine Datenstruktur, die wir im nächsten Abschnitt besprechen.

Hier eine kurze Demonstration:

```
>>> basket = {'Apfel', 'Orange', 'Apfel', 'Birne', 'Orange', 'Banane'}
>>> print(basket)                      # zeigt, dass die Duplikate entfernt wurden
{'Orange', 'Birne', 'Apfel', 'Banane'}
>>> 'Orange' in basket                 # schnelles Testen auf Mitgliedschaft
True
>>> 'Fingerhirse' in basket
False
```

```
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # einzelne Buchstaben in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                              # in a aber nicht in b
{'r', 'd', 'b'}
>>> a | b                              # in a oder b
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                              # sowohl in a, als auch in b
{'a', 'c'}
>>> a ^ b                              # entweder in a oder b
{'r', 'd', 'b', 'm', 'z', 'l'}
```

Wie für Listen gibt es auch eine “Set Comprehension”-Syntax:

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'r', 'd'}
```

5.5. Dictionaries

Ein weiterer nützlicher Datentyp, der in Python eingebaut ist, ist das Dictionary (siehe Mapping Types). Dictionaries sind in manch anderen Sprachen als “assoziativer Speicher” oder “assoziative Arrays” zu finden. Anders als Sequenzen, die über Zahlen indizierbar sind, sind Dictionaries durch Schlüssel (keys), als die jeder unveränderbare Typ dienen kann, indizierbar; aus Zeichenketten und Zahlen kann immer solch ein Schlüssel gebildet werden. Tupel können als Schlüssel benutzt werden, wenn sie nur aus Zeichenketten, Zahlen oder Tupel bestehen; enthält ein Tupel direkt oder indirekt ein veränderbares Objekt, kann es nicht als Schlüssel genutzt werden. Listen können nicht als Schlüssel benutzt werden, da sie direkt veränderbar sind, sei es durch Indexzuweisung, Abschnittszuweisung oder Methoden wie `append()` and `extend()`.

Am Besten stellt man sich Dictionaries als ungeordnete Menge von Schlüssel: Wert-Paaren vor, mit der Anforderung, dass die Schlüssel innerhalb eines Dictionaries eindeutig sind. Ein Paar von geschweiften Klammern erstellt ein leeres Dictionary: `{}`. Schreibt man eine Reihe von Komma-getrennten Schlüssel-Wert-Paaren in die Klammern, fügt man diese als Anfangspaare dem Dictionary hinzu; dies ist ebenfalls die Art und Weise, wie Dictionaries ausgegeben werden.

Die Hauptoperationen, die an einem Dictionary durchgeführt werden, sind die Ablage eines Wertes unter einem Schlüssel und der Abruf eines Wertes mit dem gegebenen Schlüssel. Es ist auch möglich ein Schlüssel-Wert-Paar per `del` zu löschen. Legt man einen Wert unter einem Schlüssel ab, der schon benutzt wird, überschreibt man den alten Wert, der vorher mit diesem Schlüssel verknüpft war. Einen Wert mit einem nicht-existenten Schlüssel abrufen zu wollen, erzeugt eine Fehlermeldung.

Der Aufruf `list(d.keys())` auf ein Dictionary gibt eine Liste aller Schlüssel in zufälliger Reihenfolge zurück (will man sie sortiert haben, verwendet man einfach die

Funktion `sorted(d.keys())` stattdessen). [2] Um zu überprüfen ob ein einzelner Schlüssel im Dictionary ist, lässt sich das Schlüsselwort **`in`** benutzen.

Hier ein kleines Beispiel wie man Dictionaries benutzt:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> list(tel.keys())
['irv', 'guido', 'jack']
>>> sorted(tel.keys())
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

Der **`dict()`**-Konstruktor erstellt Dictionaries direkt von Sequenzen von Schlüssel-Wert-Paare:

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

Außerdem können “Dict Comprehensions” benutzt werden, um Dictionaries von willkürlichen Schlüssel und Wert Ausdrücken zu erstellen:

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

Sind die Schlüssel einfache Zeichenketten, ist es manchmal einfacher, die Paare als Schlüsselwort-Argumente anzugeben:

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

5.6. Schleifentechniken

Wenn man über Dictionaries iteriert, lassen sich der Schlüssel und der entsprechende Wert gleichzeitig durch die Methode **`items()`** abrufen.

```
>>> knights = {'Gallahad': 'der Reine', 'Robin': 'der Mutige'}
>>> for k, v in knights.items():
...     print(k, v)
...
Gallahad der Reine
Robin der Mutige
```

Iteriert man über eine Sequenz, lassen sich der Index und das entsprechende Objekt gleichzeitig über die Funktion **`enumerate()`** abrufen.

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print(i, v)
...
```

```
0 tic
1 tac
2 toe
```

Um über mehrere Sequenzen gleichzeitig zu iterieren, können die Einträge mit Hilfe der `zip()`-Funktion gruppiert werden.

```
>>> questions = ['Name', 'Auftrag']
>>> answers = ['Lancelot', 'die Suche nach dem Heiligen Gral']
>>> for q, a in zip(questions, answers):
...     print('Was ist dein {0}?  Er ist {1}.'.format(q, a))
...
Was ist dein Name?  Er ist Lancelot.
Was ist dein Auftrag?  Er ist die Suche nach dem Heiligen Gral.
```

Um über eine Sequenz in umgekehrter Reihenfolge zu iterieren, gibt man die Sequenz zuerst in richtiger Reihenfolge an und ruft dann die Funktion `reversed()` auf.

```
>>> for i in reversed(range(1, 10, 2)):
...     print(i)
...
9
7
5
3
1
```

Um über eine Sequenz in sortierter Reihenfolge zu iterieren, gibt es die Funktion `sorted()`, die eine neue, sortierte Liste zurückgibt, die ursprüngliche Sequenz, aber nicht anrührt.

```
>>> basket = ['Apfel', 'Orange', 'Apfel', 'Birne', 'Orange', 'Banane']
>>> for f in sorted(set(basket)):
...     print(f)
...
Apfel
Banane
Birne
Orange
```

5.7. Mehr zu Bedingungen

Die Bedingungen, die in `while`- und `if`-Anweisungen benutzt werden, können jegliche Operatoren enthalten, nicht nur Vergleiche.

Die Vergleichsoperatoren `in` und `not in` überprüfen, ob ein Wert in einer Sequenz (nicht) vorkommt. Die Operatoren `is` und `is not` vergleichen auf Objektidentität, d.h. ob zwei Objekte dieselben sind; dies ist aber nur wichtig für veränderbare Objekte wie Listen. Alle Vergleichsoperatoren haben dieselbe Priorität, die geringer als die von numerischen Operatoren ist.

Vergleiche können aneinandergereiht werden. Zum Beispiel überprüft `a < b == c`, ob `a` kleiner als `b` ist und darüberhinaus `b` gleich `c` ist.

Vergleiche können kombiniert werden, indem man die boolschen Operatoren `and` und `or` benutzt. Das Ergebnis des Vergleiches (oder jedes anderen boolschen Ausdrucks) kann mit `not` negiert werden. Sie haben eine geringere Priorität als

Vergleichsoperatoren; von ihnen hat `not` die höchste und `or` die niedrigste Priorität, sodass `A and not B or C` äquivalent zu `(A and (not B)) or C` ist. Wie üblich können auch hier Klammern benutzt werden, um die gewünschte Gruppierung auszudrücken.

Die boolschen Operatoren `and` und `or` sind sogenannte Kurzschluss- (short-circuit) Operatoren: Ihre Argumente werden von links nach rechts ausgewertet und die Auswertung wird abgebrochen, sobald das Ergebnis feststeht. Zum Beispiel, wenn `A` und `C` wahr, aber `B` unwahr ist, wird `C` von `A and B and C` nicht ausgewertet. Werden sie als genereller Wert und nicht als Boolean benutzt, ist der Rückgabewert eines Kurzschluss-Operators das zuletzt ausgewertete Argument.

Es ist möglich das Ergebnis eines Vergleiches oder eines anderen boolschen Ausdruck einer Variablen zuzuweisen. Zum Beispiel:

```
>>> string1, string2, string3 = '', 'Trondheim', 'Hammer Tanz'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```

Wichtig ist, dass in Python, anders als in C, Zuweisungen nicht innerhalb eines Ausdrucks vorkommen können. C-Programmierer mögen darüber murren, aber diese Einschränkung vermeidet eine in C übliche Fehlerklasse, in einem Ausdruck `=` , statt des beabsichtigten `==`, zu schreiben.

5.8. Vergleich von Sequenzen mit anderen Typen

Sequenzobjekte können mit anderen Objekten desselben Sequenztyps verglichen werden. Der Vergleich benutzt eine lexikographische Ordnung: Zuerst werden die ersten beiden Elemente verglichen, unterscheiden sie sich, bestimmt das das Ergebnis des Vergleiches. Sind sie gleich, werden die nächsten zwei Elemente verglichen, und so weiter, bis sich eine der beiden erschöpft. Sind zwei Elemente, die verglichen werden sollen wiederum Sequenzen desselben Sequenztyps, wird der lexikographische Vergleich rekursiv durchgeführt. Sind alle Elemente zweier Sequenzen gleich, werden die Sequenzen für gleich befunden. Ist eine Sequenz eine anfängliche Teilfolge der anderen, so ist die kürzere die kleinere (geringere). Die Lexikographische Ordnung von Zeichenketten benutzt die Nummer des Unicode-Codepoints, um einzelne Zeichen zu ordnen. Ein paar Beispiele für Vergleiche zwischen Sequenzen des gleichen Typs:

```
(1, 2, 3)              < (1, 2, 4)
[1, 2, 3]              < [1, 2, 4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1, 2, 3, 4)           < (1, 2, 4)
(1, 2)                 < (1, 2, -1)
(1, 2, 3)             == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab'))   < (1, 2, ('abc', 'a'), 4)
```

Das Vergleichen von Objekten verschiedenen Typs durch `<` oder `>` ist erlaubt, sofern die Objekte passende Vergleichsmethoden haben. Zum Beispiel werden numerische Typen anhand ihres numerischen Wertes verglichen, sodass 0 0.0 gleicht, usw. Andernfalls wird der Interpreter eine `TypeError`-Ausnahme verursachen, statt eine willkürliche Ordnung bereitzustellen.

[1] Andere Sprachen geben möglicherweise das veränderte Objekt selbst zurück, was die Verkettung von Methoden erlaubt, wie z.B. `d->insert("a")->remove("b")->sort();`

[2] Beim Aufruf von `d.keys()` wird ein *dictionary view*-Objekt zurückgeben. Es unterstützt Operationen wie Mitgliedschaftsprüfung (membership testing) und Iteration, aber sein Inhalt ist abhängig vom ursprünglichen Dictionary – es ist nur eine Ansicht (*view*).