Informatik I

21. Eingebaute Sequenztypen

Jan-Georg Smaus

Albert-Ludwigs-Universität Freiburg

1. Februar 2011

Informatik I

1. Februar 2011 — 21. Eingebaute Sequenztypen

Danksagung

Dieses Kapitel wurde mit wenigen Änderungen aus den Materialien von Malte Helmert, Robert Mattmüller, Gabi Röger und Felix Steffenhagen übernommen, die bei diversen Python-Kursen in Freiburg zum Einsatz kamen.

21.1 Allgemeines

- Strings
- Tupel und Listen
- Tupel
- Sequenzen

Sequenzen

In diesem Kapitel befassen wir uns mit einigen von Pythons Sequenztypen:

- Strings: `str`
- (Unveränderliche) Tupel: `tuple`
- (Veränderliche) Listen: `list`

Außerdem lernen wir `for`-Schleifen besser kennen.

Beispiel zu Sequenzen

Python-Interpreter

```
>>> first_name = "John"
>>> last_name = 'Gambolputty'
>>> name = first_name + " " + last_name
>>> print(name)
John Gambolputty
>>> print(name.split())
['John', 'Gambolputty']
>>> primes = [2, 3, 5, 7]
>>> print(primes[1], sum(primes))
3 17
>>> squares = (1, 4, 9, 16, 25)
>>> print(squares[1:4])
(4, 9, 16)
```


Strings

- Strings sind uns in kleineren Beispielen schon begegnet.
- Strings werden meistens `"auf diese Weise"` angegeben. Es gibt noch alternative Schreibweisen.

Tupel und Listen

- Tupel und Listen sind Container für andere Objekte.
- Tupel werden in runden, Listen in eckigen Klammern notiert: `(2, 1, "Risiko")` vs. `["red", "green", "blue"]`.
- Tupel und Listen können beliebige Objekte enthalten, natürlich auch andere Tupel und Listen: `([18, 20, 22, "Null"], [("spam", [])])`
- Der Hauptunterschied zwischen Tupeln und Listen:
  - Listen sind veränderlich (mutable). Man kann Elemente anhängen, einfügen oder entfernen.
  - Tupel sind unveränderlich (immutable). Ein Tupel ändert sich nie, es enthält immer dieselben Objekte in derselben Reihenfolge. (Allerdings können sich die enthaltenen Objekte verändern, z.B. bei Tupeln von Listen.)

Mehr zu Tupeln

- Die Klammern um Tupel sind optional, sofern sie nicht gebraucht werden um Mehrdeutigkeiten aufzulösen:

Python-Interpreter

```
>>> mytuple = 2, 4, 5
>>> print(mytuple)
(2, 4, 5)
>>> mylist = [(1, 2), (3, 4)] # Klammern notwendig
```

- Achtung Anomalie: Einelementige Tupel schreibt man `("so",)`.

Simultane Zuweisung

Es ist in Python möglich, mehrere Variablen simultan zuzuweisen:

Python-Interpreter

```
>>> a, b = 2, 3
```

Hierbei werden Tupel komponentenweise zugewiesen. Man nennt dies Tuple Unpacking.

Mehr zu Tuple Unpacking

- Tuple Unpacking funktioniert auch mit Listen und Strings und lässt sich sogar schachteln:

Python-Interpreter

```
>>> [a, (b, c), (d, e), f] = (42, (6, 9), "do", [1, 2])
>>> print(a, "*", b, "*", c, "*", d, "*", e, "*", f)
42 * 6 * 9 * d * o * [1, 2]
```


Sequenzen

- Strings, Tupel und Listen haben etwas gemeinsam: Sie enthalten andere Dinge in einer bestimmten Reihenfolge und erlauben direkten Zugriff auf die einzelnen Komponenten mittels Indizierung.
- Typen mit dieser Eigenschaft bezeichnet man als Sequenztypen, ihre Instanzen als Sequenzen.

21.2 Unterstützte Operationen

- Verkettung
- Wiederholung
- Indizierung
- Mitgliedschaftstest
- Slicing
- Iteration
- Builtins für `for`-Schleifen
- Zusammenfassung

Unterstützte Operationen

Sequenztypen unterstützen die folgenden Operationen:

- Verkettung: `"Gambol" + "putty" == "Gambolputty"`
- Wiederholung: `2 * "spam" == "spam" * 2 == "spamspam"`
- Indizierung: `"Python"[1] == "y"`
- Mitgliedschaftstest: `17 in [11,13,17,19]`
- Slicing: `"Monty Python's Flying Circus"[6:12] == "Python"`
- Iteration: `for x in "egg"`

Verkettung

Sequenztypen unterstützen die folgenden Operationen:

- Verkettung: `"Gambol" + "putty" == "Gambolputty"`
- Wiederholung: `2 * "spam" == "spam" * 2 == "spamspam"`
- Indizierung: `"Python"[1] == "y"`
- Mitgliedschaftstest: `17 in [11,13,17,19]`
- Slicing: `"Monty Python's Flying Circus"[6:12] == "Python"`
- Iteration: `for x in "egg"`

Verkettung

Python-Interpreter

```
>>> print("Gambol" + "putty")
Gambolputty
>>> mylist = ["spam", "egg"]
>>> print(["spam"] + mylist)
['spam', 'spam', 'egg']
```

Verkettung II

Python-Interpreter

```
>>> primes = (2, 3, 5, 7)
>>> print(primes + primes)
(2, 3, 5, 7, 2, 3, 5, 7)
>>> print(mylist + primes)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError:  can only concatenate list (not "tuple") to list
>>> print(mylist + list(primes))
['spam', 'egg', 2, 3, 5, 7]
```


Wiederholung

Sequenztypen unterstützen die folgenden Operationen:

- Verkettung: `"Gambol" + "putty" == "Gambolputty"`
- Wiederholung: `2 * "spam" == "spam" * 2 == "spamspam"`
- Indizierung: `"Python"[1] == "y"`
- Mitgliedschaftstest: `17 in [11,13,17,19]`
- Slicing: `"Monty Python's Flying Circus"[6:12] == "Python"`
- Iteration: `for x in "egg"`

Wiederholung

Python-Interpreter

```
>>> print("*" * 20)
********************
>>> print([None, 2, 3] * 3)
[None, 2, 3, None, 2, 3, None, 2, 3]
>>> print(2 * ("parrot", ["is", "dead"]))
('parrot', ['is', 'dead'], 'parrot', ['is', 'dead'])
```


Indizierung

Sequenztypen unterstützen die folgenden Operationen:

- Verkettung: `"Gambol" + "putty" == "Gambolputty"`
- Wiederholung: `2 * "spam" == "spam" * 2 == "spamspam"`
- Indizierung: `"Python"[1] == "y"`
- Slicing: `"Monty Python's Flying Circus"[6:12] == "Python"`
- Iteration: `for x in "egg"`

Indizierung

- Sequenzen können von vorne und von hinten indiziert werden.
- Bei Indizierung von vorne hat das vorderste Element Index 0.
- Zur Indizierung von hinten verwendet man negative Indizes. Dabei hat das hinterste Element den Index -1.

Indizierung: Beispiele

Python-Interpreter

```
>>> primes = (2, 3, 5, 7, 11, 13)
>>> print(primes[1], primes[-1])
3 13
>>> animal = "parrot"
>>> animal[-2]
'o'
>>> animal[10]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError:  string index out of range
```


Wo sind die Zeichen?

In Python gibt es keinen eigenen Typ für Zeichen (chars).
Für Python ist ein Zeichen einfach ein String der Länge 1.

Python-Interpreter

```
>>> food = "spam"
>>> food
'spam'
>>> food[0]
's'
>>> type(food)
<class 'str'>
>>> type(food[0])
<class 'str'>
>>> food[0][0][0][0][0]
's'
```


Indizierung: Zuweisung an Indizes I

- Listen kann man per Zuweisung an Indizes verändern:

Python-Interpreter

```
>>> primes = [2, 3, 6, 7, 11]
>>> primes[2] = 5
>>> print(primes)
[2, 3, 5, 7, 11]
>>> primes[-1] = 101
>>> print(primes)
[2, 3, 5, 7, 101]
```

- Auch hier müssen die entsprechenden Indizes existieren.

Indizierung: Zuweisung an Indizes II

- Tupel und Strings sind unveränderlich:

Python-Interpreter

```
>>> food = "ham"
>>> food[0] = "j"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError:  'str' object does not support item assignment
>>> pair = (10, 3)
>>> pair[1] = 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError:  'tuple' object does not support item assignment
```


Mitgliedschaftstest

Sequenztypen unterstützen die folgenden Operationen:

- Verkettung: `"Gambol" + "putty" == "Gambolputty"`
- Wiederholung: `2 * "spam" == "spam" * 2 == "spamspam"`
- Indizierung: `"Python"[1] == "y"`
- Mitgliedschaftstest: `17 in [11,13,17,19]`
- Slicing: `"Monty Python's Flying Circus"[6:12] == "Python"`
- Iteration: `for x in "egg"`

Mitgliedschaftstest: Der `in`-Operator

- `item in seq` (`seq` ist ein Tupel oder eine Liste):
  Liefert True, wenn `seq` das Element `item` enthält.
  Laufzeit hängt linear von der Länge der Liste ab.
- `substring in string` (`string` ist ein String):
  Liefert True, wenn `string` den Teilstring `substring` enthält.

Python-Interpreter

```
>>> print(2 in [1, 4, 2])
True
>>> if "spam" in ("ham", "eggs", "sausage"):
...     print("tasty")
...
>>> print("m" in "spam", "ham" in "spam", "pam" in "spam")
True False True
```


Slicing

Sequenztypen unterstützen die folgenden Operationen:

- Verkettung: `"Gambol" + "putty" == "Gambolputty"`
- Wiederholung: `2 * "spam" == "spam" * 2 == "spamspam"`
- Indizierung: `"Python"[1] == "y"`
- Mitgliedschaftstest: `17 in [11,13,17,19]`
- Slicing: `"Monty Python's Flying Circus"[6:12] == "Python"`
- Iteration: `for x in "egg"`

Slicing

- Slicing ist das Ausschneiden von "Scheiben" aus einer Sequenz:

Python-Interpreter

```
>>> primes = [2, 3, 5, 7, 11, 13]
>>> print(primes[1:4])
[3, 5, 7]
>>> print(primes[:2])
[2, 3]
>>> print("egg, sausage and bacon"[-5:])
bacon
```


Slicing: Erklärung

- `seq[i:j]` liefert den Bereich $[i,j)$, also die Elemente an den Positionen $i, i+1, \ldots, j-1$:
  `("do", "re", 5, 7)[1:3] == ("re", 5)`
- Lässt man i weg, beginnt der Bereich an Position 0:
  `("do", "re", 5, 7)[:3] == ("do", "re", 5)`
- Lässt man j weg, endet der Bereich am Ende der Folge:
  `("do", "re", 5, 7)[1:] == ("re", 5, 7)`
- Lässt man beide weg, erhält man eine Kopie der gesamten Folge:
  `("do", "re", 5, 7)[:] == ("do", "re", 5, 7)`

Slicing: Erklärung II

- Beim Slicing gibt es keine Index-Fehler: Bereiche jenseits des Endes der Folge sind einfach leer:

Python-Interpreter

```
>>> "spam"[2:10]
'am'
>>> "spam"[-6:3]
'spa'
>>> "spam"[7:]
''
```

- Auch beim Slicing kann man "von hinten zählen".
  So erhält man die drei letzten Elemente einer Folge z.B. mit `seq[-3:]`.

Slicing: Zuweisungen an Slices I

- Bei Listen kann man auch Slice-Zuweisungen durchführen, d.h. einen Teil einer Liste durch eine andere Sequenz ersetzen:

Python-Interpreter

```
>>> dish = ["ham", "sausage", "eggs", "bacon"]
>>> dish[1:3] = ["spam", "spam"]
>>> print(dish)
["ham", "spam", "spam", "bacon"]
>>> dish[:1] = ["spam"]
>>> print(dish)
["spam", "spam", "spam", "bacon"]
```

Slicing: Zuweisungen an Slices II

- Die zugewiesene Sequenz muss nicht gleich lang sein wie der zu ersetzende Bereich. Beide dürfen sogar leer sein:

Python-Interpreter

```
>>> print(dish)
["spam", "spam", "spam", "bacon"]
>>> dish[1:4] = ["baked beans"]
>>> print(dish)
["spam", "baked beans"]
>>> dish[1:1] = ["sausage", "spam", "spam"]
>>> print(dish)
["spam", "sausage", "spam", "spam", "baked beans"]
>>> dish[2:4] = []
>>> print(dish)
["spam", "sausage", "baked beans"]
```


Slicing und Listen: Die del-Anweisung

- Statt einem Slice eine leere Sequenz zuzuweisen, kann man auch die `del`-Anweisung verwenden, die einzelne Elemente oder Slices enfernt:

Python-Interpreter

```
>>> primes = [2, 3, 5, 7, 11, "spam", 13]
>>> del primes[-2]
>>> primes
[2, 3, 5, 7, 11, 13]
>>> months = ["april", "may", "grune", "sectober", "june"]
>>> del months[2:4]
>>> months
['april', 'may', 'june']
```


Iteration

Sequenztypen unterstützen die folgenden Operationen:

- Verkettung: `"Gambol" + "putty" == "Gambolputty"`
- Wiederholung: `2 * "spam" == "spam" * 2 == "spamspam"`
- Indizierung: `"Python"[1] == "y"`
- Slicing: `"Monty Python's Flying Circus"[6:12] == "Python"`
- Iteration: `for x in "egg"`

Iteration

- Zum Durchlaufen von Sequenzen verwendet man `for`-Schleifen:

Python-Interpreter

```
>>> primes = [2, 3, 5, 7]
>>> product = 1
>>> for number in primes:
...     product = product * number
...
...  print(product)
210
```

Iteration II

- `for` funktioniert mit allen Sequenztypen:

Python-Interpreter

```
>>> for character in "spam":
...     print(character * 2)
...
ss
pp
aa
mm
>>> for ingredient in ("spam", "spam", "egg"):
...     if ingredient == "spam":
...       print("tasty!")
...
tasty!
tasty!
```


Iteration: Mehrere Schleifenvariablen

- Wenn man eine Sequenz von Sequenzen durchläuft, kann man mehrere Schleifenvariablen gleichzeitig binden:

Python-Interpreter

```
>>> couples = [("Justus", "Lys"),
...            ("Peter", "Kelly"),
...            ("Bob", "Liz")]
>>> for x, y in couples:
...     print(x, "ist cool;", y, "nervt.")
...
Justus ist cool; Lys nervt.
Peter ist cool; Kelly nervt.
Bob ist cool; Liz nervt.
```

- Dies ist ein Spezialfall des früher gesehenen Tuple Unpacking.

break, continue, else

Im Zusammenhang mit Schleifen sind die folgenden drei Anweisungen interessant:

- `break` beendet eine Schleife vorzeitig.
- `continue` beendet die aktuelle Schleifeniteration vorzeitig, d.h. springt zum Schleifenkopf und setzt die Schleifenvariable(n) auf den nächsten Wert.
- Außerdem können Schleifen (so wie `if`-Abfragen) einen `else`-Zweig aufweisen. Dieser wird nach Beendigung der Schleife ausgeführt, und zwar genau dann, wenn die Schleife nicht mit `break` verlassen wurde.

`break`, `continue` und `else` funktionieren genauso bei den bereits gesehenen `while`-Schleifen.

break, continue und else: Beispiel

break-continue-else.py

```
foods_and_amounts = [("sausage", 2), ("eggs", 0),
                     ("spam", 2), ("ham", 1)]

for food, amount in foods_and_amounts:
  if amount == 0:
    continue
  if food == "spam":
    print(amount, "tasty piece(s) of spam.")
    break
else:
  print("No spam!")

# Ausgabe:
# 2 tasty piece(s) of spam.
```

Listen während der Iteration ändern

- Innerhalb einer Schleife sollte das durchlaufene Objekt nicht seine Größe ändern. Ansonsten kommt es zu verwirrenden Ergebnissen:

Python-Interpreter

```
>>> numbers = [3, 5, 7]
>>> for n in numbers:
...     print(n)
...     if n == 3:
...       del numbers[0]
...
3
7
>>> print(numbers)
[5, 7]
```


Listen während der Iteration ändern II

- Abhilfe kann man schaffen, indem man eine Kopie der Liste durchläuft:

Python-Interpreter

```
>>> numbers = [3, 5, 7]
>>> for n in numbers[:]:
...     print(n)
...     if n == 3:
...       del numbers[0]
...
3
5
7
>>> print(numbers)
[5, 7]
```


Nützliche Funktionen im Zusammenhang mit `for`-Schleifen

Einige builtins tauchen häufig im Zusammenhang mit `for`-Schleifen auf und sollen hier nicht unerwähnt bleiben:

- `range`
- `enumerate`

`range`

- Bereichsobjekte sind spezielle iterierbare Objekte, die bestimmte Listen/Mengen von `int`s darstellen, und die vor allem für Schleifendurchläufe gedacht sind.
- `range` erzeugt solche Bereichsobjekte:
  - `range(stop)` ergibt `0, 1, ..., stop-1`
  - `range(start, stop)` ergibt `start, start+1, ..., stop-1`

  `range` spart gegenüber einer "echten" Liste Speicherplatz, da gerade keine Liste angelegt werden muss.

range: Beispiele

Python-Interpreter

```
>>> range(5)
range(0, 5)
>>> list(range(5))
[0, 1, 2, 3, 4]
>>> for i in range(3, 6):
...     print(i, "** 3 =", i ** 3)
...
3 ** 3 = 27
4 ** 3 = 64
5 ** 3 = 125
```


enumerate

- Manchmal möchte man beim Durchlaufen einer Sequenz wissen, an welcher Position man gerade ist.
- Dazu dient die Funktion `enumerate`, die eine Sequenz als Argument erhält und eine Folge von Paaren `(index, element)` liefert:

Python-Interpreter

```
>>> for i, char in enumerate("egg"):
...     print("An Position", i, "steht ein", char)
...
An Position 0 steht ein e
An Position 1 steht ein g
An Position 2 steht ein g
```

- Auch `enumerate` erzeugt keine "richtige" Liste, sondern ist vornehmlich für `for`-Schleifen gedacht.

Zusammenfassung

- Wir haben Sequenztypen kennengelernt: Tupel, Listen, und Strings.
- Unterschied zwischen Tupel und Listen: Tupel sind unveränderlich, Listen sind veränderlich.
- Mit Sequenzen kann man allerhand Dinge tun: Verkettung, Wiederholung, Indizierung, Mitgliedschaftstest, Slicing, und Iteration.
- Python-Listen sind zu unterscheiden von verlinkten Listen: Python-Listen erlauben direkten Zugriff, aber Einfüge- und Entfernoperationen können teuer sein.
- Es wäre möglich, auch für verlinkte Listen die Iteration mittels `in` zu implementieren, aber wir werden es nicht tun.