

Informatik I: Einführung in die Programmierung

10. Sequenzen, for-Schleifen, Objekte und Identität

Albert-Ludwigs-Universität Freiburg



**UNI
FREIBURG**

Bernhard Nebel

10. November 2017

- Strings
- Tupel und Listen
- Tupel Unpacking

Sequenzen

Strings
Tupel und Listen
Tupel Unpacking

Operationen auf Sequenzen

Iteration

Objekte und
Identität

Wir beschäftigen uns jetzt mit Pythons Sequenztypen:

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

Außerdem lernen wir `for`-Schleifen kennen.

Sequenzen

Strings
Tupel und Listen
Tupel Unpacking

Operationen
auf
Sequenzen

Iteration

Objekte und
Identität

Python-Interpreter

```
>>> first_name = "Johann"
>>> last_name = 'Gambolputty'
>>> name = first_name + " " + last_name
>>> print(name)
Johann Gambolputty
>>> print(name.split())
['Johann', '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)
```

Sequenzen

Strings
Tupel und Listen
Tupel Unpacking

Operationen
auf
Sequenzen

Iteration

Objekte und
Identität



- Strings sind uns in kleineren Beispielen schon begegnet.
- Strings sind in Python grundsätzlich Unicode-Strings (d.h. sie entsprechen damit den Strings von Java).
- Strings werden meistens "auf diese Weise" angegeben, es gibt aber viele alternative Schreibweisen.

Sequenzen

Strings

Tupel und Listen

Tupel Unpacking

Operationen
auf

Sequenzen

Iteration

Objekte und
Identität



- **Tupel** und **Listen** sind Container für andere Objekte (grob vergleichbar mit Vektoren in C++/Java).
- 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.)

Sequenzen

Strings

Tupel und Listen

Tupel Unpacking

Operationen
auf

Sequenzen

Iteration

Objekte und
Identität

- 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",).
- Bei `a, b = 2, 3` werden *Tupel* komponentenweise zugewiesen; man spricht auch von **Tuple Unpacking**.

Sequenzen

Strings

Tupel und Listen

Tupel Unpacking

Operationen
auf
Sequenzen

Iteration

Objekte und
Identität

- 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, 3])
>>> print(a, "*", b, "*", c, "*", d, "*", e, "*", f)
42 * 6 * 9 * d * o * [1, 2, 3]
```

Sequenzen

Strings

Tupel und Listen

Tupel Unpacking

Operationen
auf

Sequenzen

Iteration

Objekte und
Identität

2 Operationen auf Sequenzen



- Verkettung
- Wiederholung
- Indizierung
- Mitgliedschaftstest
- Slicing
- Typkonversion
- Weitere Sequenz-Funktionen

Sequenzen

Operationen
auf
Sequenzen

Verkettung

Wiederholung

Indizierung

Mitgliedschaftstest

Slicing

Typkonversion

Weitere Sequenz-
Funktionen

Iteration

Objekte und
Identität



- 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**.

Sequenztypen unterstützen die folgenden Operationen:

- Verkettung: `"Gambol" + "putty" == "Gambolputty"`
- Wiederholung: `2 * "spam" == "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"`

Sequenzen

Operationen
auf
Sequenzen

Verkettung

Wiederholung

Indizierung

Mitgliedschaftstest

Slicing

Typkonversion

Weitere Sequenz-
Funktionen

Iteration

Objekte und
Identität

Python-Interpreter

```
>>> print("Gambol" + "putty")
Gambolputty
>>> mylist = ["spam", "egg"]
>>> print(["spam"] + mylist)
['spam', 'spam', 'egg']
>>> primes = (2, 3, 5, 7)
>>> print(primes + primes)
(2, 3, 5, 7, 2, 3, 5, 7)
>>> print(mylist + primes)
Traceback (most recent call last): ...
TypeError: can only concatenate list (not "tuple") to
list
>>> print(mylist + list(primes))
['spam', 'egg', 2, 3, 5, 7]
```

Sequenzen

Operationen

auf

Sequenzen

Verkettung

Wiederholung

Indizierung

Mitgliedschaftstest

Slicing

Typkonversion

Weitere Sequenz-
Funktionen

Iteration

Objekte und

Identität

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'])
```

Sequenzen

Operationen
auf
Sequenzen

Verkettung

Wiederholung

Indizierung

Mitgliedschaftstest

Slicing

Typkonversion

Weitere Sequenz-
Funktionen

Iteration

Objekte und
Identität



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

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): ...
IndexError: string index out of range
```

Sequenzen

Operationen
auf
Sequenzen

Verkettung

Wiederholung

Indizierung

Mitgliedschaftstest

Slicing

Typkonversion

Weitere Sequenz-
Funktionen

Iteration

Objekte und
Identität

Wo sind die Zeichen?



- In Python gibt es keinen eigenen Datentyp 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'
```

Sequenzen

Operationen
auf
Sequenzen

Verkettung

Wiederholung

Indizierung

Mitgliedschaftstest

Slicing

Typkonversion

Weitere Sequenz-
Funktionen

Iteration

Objekte und
Identität

- 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.

Sequenzen

Operationen
auf
Sequenzen

Verkettung

Wiederholung

Indizierung

Mitgliedschaftstest

Slicing

Typkonversion

Weitere Sequenz-
Funktionen

Iteration

Objekte und
Identität

- Tupel und Strings sind unveränderlich:

Python-Interpreter

```
>>> food = "ham"
```

```
>>> food[0] = "j"
```

```
Traceback (most recent call last): ...
```

```
TypeError: 'str' object does not support item  
assignment
```

```
>>> pair = (10, 3)
```

```
>>> pair[1] = 4
```

```
Traceback (most recent call last): ...
```

```
TypeError: 'tuple' object doesn't support item  
assignment
```

Sequenzen

Operationen
auf
Sequenzen

Verkettung

Wiederholung

Indizierung

Mitgliedschaftstest

Slicing

Typkonversion

Weitere Sequenz-
Funktionen

Iteration

Objekte und
Identität



- `item in seq` (`seq` ist ein Tupel oder eine Liste):
Liefert `True`, wenn `seq` das Element `item` enthält.
- `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
```

Sequenzen

Operationen
auf
Sequenzen

Verkettung

Wiederholung

Indizierung

Mitgliedschaftstest

Slicing

Typkonversion

Weitere Sequenz-

Funktionen

Iteration

Objekte und
Identität

- *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
```

Sequenzen

Operationen
auf
Sequenzen

Verkettung

Wiederholung

Indizierung

Mitgliedschaftstest

Slicing

Typkonversion

Weitere Sequenz-
Funktionen

Iteration

Objekte und
Identität

- `seq[i:j]` liefert den Bereich $[i,j)$, also die Elemente an den Positionen $i, i+1, \dots, j-1$:
`("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)`

Sequenzen

Operationen
auf
Sequenzen

Verkettung

Wiederholung

Indizierung

Mitgliedschaftstest

Slicing

Typkonversion

Weitere Sequenz-
Funktionen

Iteration

Objekte und
Identität

- 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:]`.

Sequenzen

Operationen
auf
Sequenzen

Verkettung

Wiederholung

Indizierung

Mitgliedschaftstest

Slicing

Typkonversion

Weitere Sequenz-
Funktionen

Iteration

Objekte und
Identität

- Beim sogenannten *erweiterten Slicing* kann man zusätzlich noch eine Schrittweite angeben:

Python-Interpreter

```
>>> zahlen = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> zahlen[1:7:2]
[1, 3, 5]
>>> zahlen[1:8:2]
[1, 3, 5, 7]
>>> zahlen[7:2:-1]
[7, 6, 5, 4, 3]
>>> zahlen[::-1]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Sequenzen

Operationen
auf
Sequenzen

Verkettung

Wiederholung

Indizierung

Mitgliedschaftstest

Slicing

Typkonversion

Weitere Sequenz-
Funktionen

Iteration

Objekte und
Identität



- 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']
```

Sequenzen

Operationen
auf
Sequenzen

Verkettung

Wiederholung

Indizierung

Mitgliedschaftstest

Slicing

Typkonversion

Weitere Sequenz-
Funktionen

Iteration

Objekte und
Identität

Slicing: Zuweisungen an Slices (2)



- Die zugewiesene Sequenz muss nicht gleich lang sein wie der zu ersetzende Bereich. Beide dürfen 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']
```

- Bei Slices mit Schrittweite muss beides gleich lang sein.

Sequenzen

Operationen
auf
Sequenzen

Verkettung

Wiederholung

Indizierung

Mitgliedschaftstest

Slicing

Typkonversion

Weitere Sequenz-
Funktionen

Iteration

Objekte und
Identität



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

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']
```

Sequenzen

Operationen
auf
Sequenzen

Verkettung
Wiederholung
Indizierung
Mitgliedschaftstest
Slicing
Typkonversion
Weitere Sequenz-
Funktionen

Iteration

Objekte und
Identität

list, **tuple**, und **str** konvertieren zwischen den Sequenztypen (aber nicht immer so wie man hofft).

Python-Interpreter

```
>>> tuple([0, 1, 2])
(0, 1, 2)
>>> list(('spam', 'egg'))
['spam', 'egg']
>>> list('spam')
['s', 'p', 'a', 'm']
>>> tuple('spam')
('s', 'p', 'a', 'm')
>>> str(['a', 'b', 'c'])
"['a', 'b', 'c']"
```

Sequenzen

Operationen
auf
Sequenzen

Verkettung

Wiederholung

Indizierung

Mitgliedschaftstest

Slicing

Typkonversion

Weitere Sequenz-
Funktionen

Iteration

Objekte und
Identität



- `sum(seq)`:
Berechnet die Summe einer Zahlensequenz.
- `min(seq), min(x, y, ...)`:
Berechnet das Minimum einer Sequenz (erste Form)
bzw. der Argumente (zweite Form).
 - Sequenzen werden lexikographisch verglichen.
 - Der Versuch, das Minimum konzeptuell unvergleichbarer Typen (etwa Zahlen und Listen) zu bilden, führt zu einem `TypeError`.
- `max(seq), max(x, y, ...)`: \rightsquigarrow analog zu `min`

Python-Interpreter

```
>>> max([1, 23, 42, 5])
42
>>> sum([1, 23, 42, 5])
71
```

Sequenzen

Operationen
auf
Sequenzen

Verkettung

Wiederholung

Indizierung

Mitgliedschaftstest

Slicing

Typkonversion

Weitere Sequenz-
Funktionen

Iteration

Objekte und
Identität

- `any(seq)`:
Äquivalent zu `elem1 or elem2 or elem3 or ...`,
wobei `elemi` die Elemente von `seq` sind und nur `True`
oder `False` zurück geliefert wird.
- `all(seq)`: \rightsquigarrow analog zu `any`

Sequenzen

Operationen
auf
Sequenzen

Verkettung

Wiederholung

Indizierung

Mitgliedschaftstest

Slicing

Typkonversion

Weitere Sequenz-
Funktionen

Iteration

Objekte und
Identität

- `len(seq)`:
Berechnet die Länge einer Sequenz.
- `sorted(seq)`:
Liefert eine Liste, die dieselben Elemente hat wie `seq`,
aber (stabil) sortiert ist.

Sequenzen

Operationen
auf
Sequenzen

Verkettung

Wiederholung

Indizierung

Mitgliedschaftstest

Slicing

Typkonversion

Weitere Sequenz-
Funktionen

Iteration

Objekte und
Identität

3 Iteration

- Mehrere Variablen
- Nützliche Funktionen

Sequenzen

Operationen
auf
Sequenzen

Iteration

Mehrere Variablen
Nützliche
Funktionen

Objekte und
Identität

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

Python-Interpreter

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

Visualisierung

Sequenzen

Operationen
auf
Sequenzen

Iteration

Mehrere Variablen
Nützliche
Funktionen

Objekte und
Identität

- 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!
```

Sequenzen

Operationen
auf
Sequenzen

Iteration

Mehrere Variablen

Nützliche
Funktionen

Objekte und
Identität



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

Python-Interpreter

```
>>> menus = [("egg", "spam"), ("ham", "spam"),  
...          ("beans", "bacon")]  
>>> for x, y in menus:  
...     print(x, "is yuk, but", y, "is tasty.")  
...  
egg is yuk, but spam is tasty.  
ham is yuk, but spam is tasty.  
beans is yuk, but beacon is tasty.
```

- Dies ist ein Spezialfall des früher gesehenen Tuple Unpacking.

Sequenzen

Operationen
auf
Sequenzen

Iteration

Mehrere Variablen
Nützliche
Funktionen

Objekte und
Identität

Im Zusammenhang mit Schleifen sind die folgenden drei Anweisungen interessant:

- `break` beendet eine Schleife vorzeitig wie bei `while`-Schleifen.
- `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 ebenso bei den bereits gesehenen `while`-Schleifen.

Sequenzen

Operationen
auf
Sequenzen

Iteration

Mehrere Variablen
Nützliche
Funktionen

Objekte und
Identität

break, continue und else: Beispiel



```
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.
```

Sequenzen

Operationen
auf
Sequenzen

Iteration

Mehrere Variablen
Nützliche
Funktionen

Objekte und
Identität



- 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]
```

Sequenzen

Operationen
auf
Sequenzen

Iteration

Mehrere Variablen

Nützliche
Funktionen

Objekte und
Identität

Listen während der Iteration ändern (2)



- 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]
```

Sequenzen

Operationen
auf
Sequenzen

Iteration

Mehrere Variablen

Nützliche
Funktionen

Objekte und
Identität

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

- `range`
- `enumerate`
- `zip`
- `reversed`

Sequenzen

Operationen
auf
Sequenzen

Iteration

Mehrere Variablen

Nützliche
Funktionen

Objekte und
Identität

- Bereichsobjekte sind spezielle iterierbare Objekte, die bestimmte Listen/Mengen von ints 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(start, stop, step) ergibt start, start + step, start + 2 * step, ..., stop-1

range spart gegenüber einer ‚echten‘ Liste Speicherplatz, da gerade *keine* Liste angelegt werden muss. Es wird ein sog. **Iterator** erzeugt.

Sequenzen

Operationen
auf
Sequenzen

Iteration

Mehrere Variablen

Nützliche
Funktionen

Objekte und
Identität

Python-Interpreter

```
>>> range(5)
range(0, 5)
>>> range(3, 30, 10)
range(3, 30, 10)
>>> list(range(3, 30, 10))
[3, 13, 23]
>>> for i in range(3, 6):
...     print(i, "** 3 =", i ** 3)
...
3 ** 3 = 27
4 ** 3 = 64
5 ** 3 = 125
```

Sequenzen

Operationen
auf
Sequenzen

Iteration

Mehrere Variablen

Nützliche
Funktionen

Objekte und
Identität



- 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 einen Iterator. Ist vornehmlich für `for`-Schleifen gedacht.

Sequenzen

Operationen
auf
Sequenzen

Iteration

Mehrere Variablen

Nützliche
Funktionen

Objekte und
Identität

- Die Funktion `zip` nimmt eine oder mehrere Sequenzen und liefert eine Liste von Tupeln mit korrespondierenden Elementen.
- Auch `zip` erzeugt keine ‚richtige‘ Liste, sondern einen Iterator; will man daraus eine Liste erzeugen, muss man explizit den Listen-Konstruktor aufrufen.

Sequenzen

Operationen
auf
Sequenzen

Iteration

Mehrere Variablen

Nützliche
Funktionen

Objekte und
Identität

Python-Interpreter

```
>>> meat = ["spam", "ham", "beacon"]
>>> sidedish = ["spam", "pasta", "chips"]
>>> print(list(zip(meat,sidedish)))
[('spam', 'spam'), ('ham', 'pasta'), ('beacon',
'chips')]
```

- Besonders nützlich ist `zip`, um mehrere Sequenzen parallel zu durchlaufen:

Python-Interpreter

```
>>> for x, y, z in zip("ham", "spam", range(5, 10)):  
...     print(x, y, z)  
...  
h s 5  
a p 6  
m a 7
```

- Sind die Eingabesequenzen unterschiedlich lang, ist das Ergebnis so lang wie die kürzeste Eingabe.

Sequenzen

Operationen
auf
Sequenzen

Iteration

Mehrere Variablen

Nützliche
Funktionen

Objekte und
Identität

- Will man eine Sequenz in umgekehrter Richtung durchlaufen, kann man `reversed` benutzen.
- Erzeugt wie `enumerate` einen Iterator.

Python-Interpreter

```
>>> for x in reversed("ham")
: ...     print(x)
...
m
a
h
```

Sequenzen

Operationen
auf
Sequenzen

Iteration

Mehrere Variablen

Nützliche
Funktionen

Objekte und
Identität

- Objekte
- Objekte und Variablen
- Identität
- Identität und Gleichheit
- Identität von Literalen
- Erweiterte Zuweisungen: Änderbare und nicht änderbare Strukturen
- Zyklische Datenstrukturen

Sequenzen

Operationen
auf
Sequenzen

Iteration

**Objekte und
Identität**

Objekte

Objekte und
Variablen

Identität

Identität und
Gleichheit

Identität von
Literalen

Erweiterte
Zuweisungen:
Änderbare und
nicht änderbare
Strukturen

Zyklische
Datenstrukturen



- Man kann es nicht länger verschweigen: Alle *Werte* sind in Wirklichkeit *Objekte*.
- Damit ist gemeint, dass sie nicht nur aus reinen *Daten* bestehen, sondern auch assoziierte *Attribute* und *Methoden* haben, auf die mit der Punktnotation `ausdruck.attribut` zugegriffen werden kann:

Python-Interpreter

```
>>> x = complex(10, 3)
>>> x.real, x.imag
10.0 3.0
>>> "spam".index("a")
2
>>> (10 + 10).__neg__()
-20
```

- Später mehr dazu ...

Sequenzen

Operationen
auf
Sequenzen

Iteration

Objekte und
Identität

Objekte

Objekte und
Variablen

Identität

Identität und
Gleichheit

Identität von
Literalen

Erweiterte
Zuweisungen:
Änderbare und
nicht änderbare
Strukturen

Zyklische
Datenstrukturen

- Was bewirkt `x = <ausdruck>`?
 - Die naive Antwort lautet: ‚Der Variablen `x` wird der Wert `<ausdruck>` zugewiesen.‘
 - Eine *bessere*, weil zutreffendere Antwort, lautet aber eher umgekehrt: ‚Dem durch `<ausdruck>` bezeichneten Objekt wird der Name `x` zugeordnet.‘ Entscheidend ist dabei, dass **dasselbe Objekt** unter **mehreren Namen** bekannt sein kann:

Python-Interpreter

```
>>> food = ["spam", "eggs", "bacon"]
>>> lunch = food
>>> del lunch[0]
>>> print(lunch)
['eggs', 'bacon']
>>> print(food)
['eggs', 'bacon']
```

Visualisierung

Sequenzen

Operationen
auf
Sequenzen

Iteration

Objekte und
Identität

Objekte

Objekte und
Variablen

Identität

Identität und
Gleichheit

Identität von
Literalen

Erweiterte
Zuweisungen:
Änderbare und
nicht änderbare
Strukturen

Zyklische
Datenstrukturen

- Identität lässt sich mit den Operatoren `is` und `is not` testen:
- `x is y` ist `True`, wenn `x` und `y` **dasselbe Objekt** bezeichnen, und ansonsten `False` (`is not` umgekehrt):

Python-Interpreter

```
>>> x = ["ham", "spam", "jam"]
>>> y = ["ham", "spam", "jam"]
>>> z = y
>>> x is y, x is z, y is z
(False, False, True)
>>> x is not y, x is not z, y is not z
(True, True, False)
>>> del y[1]
>>> x, y, z
(['ham', 'spam', 'jam'], ['ham', 'jam'], ['ham', 'jam'])
```

Sequenzen

Operationen
auf
Sequenzen

Iteration

Objekte und
Identität

Objekte
Objekte und
Variablen

Identität
Identität und
Gleichheit
Identität von
Literalen

Erweiterte
Zuweisungen:
Änderbare und
nicht änderbare
Strukturen

Zyklische
Datenstrukturen

- `id(x)` liefert ein `int`, das eine Art ‚Sozialversicherungsnummer‘ für das durch `x` bezeichnete Objekt ist: Zu keinem Zeitpunkt während der Ausführung eines Programms haben zwei Objekte die gleiche `id`.
- `x is y` ist äquivalent zu `id(x) == id(y)`.

Python-Interpreter

```
>>> x = ["ham", "spam", "jam"]
>>> y = ["ham", "spam", "jam"]
>>> z = y
>>> id(x), id(y), id(z)
(1076928940, 1076804076, 1076804076)
```

Sequenzen

Operationen
auf
Sequenzen

Iteration

Objekte und
Identität

Objekte

Objekte und
Variablen

Identität

Identität und
Gleichheit
Identität von
Literalen

Erweiterte
Zuweisungen:
Änderbare und
nicht änderbare
Strukturen

Zyklische
Datenstrukturen



Zu jedem Zeitpunkt haben alle Objekte unterschiedliche ids.
Es ist allerdings möglich, dass die id eines alten Objektes
wiederverwendet wird, nachdem es nicht mehr benötigt wird:

```
x = [1, 2, 3]
y = [4, 5, 6]
my_id = id(x)
x = [7, 8, 9]
# Das alte Objekt wird nicht mehr benötigt
# => my_id wird frei.
z = [10, 11, 12]
# my_id und id(z) könnten jetzt gleich sein,
# falls Implementierung id wiederverwendet.
```

Sequenzen

Operationen
auf
Sequenzen

Iteration

Objekte und
Identität

Objekte

Objekte und
Variablen

Identität

Identität und
Gleichheit
Identität von
Literalen

Erweiterte
Zuweisungen:
Änderbare und
nicht änderbare
Strukturen

Zyklische
Datenstrukturen

- Wir haben es bisher nur bei Strings gesehen, aber man kann Listen und Tupel auch auf Gleichheit testen. Der Unterschied zum Identitätstest ist wichtig:

Python-Interpreter

```
>>> x = ["ham", "spam", "jam"]
>>> y = ["ham", "spam", "jam"]
>>> x == y, x is y
(True, False)
```

- Bei *Gleichheit* wird getestet, ob *x* und *y* den gleichen Typ haben, gleich lang sind und korrespondierende Elemente gleich sind (die Definition ist rekursiv).
- Bei *Identität* wird getestet, ob *x* und *y* dasselbe Objekt bezeichnen.
- Der Gleichheitstest ist verbreiteter; z.B. testet der `in`-Operator immer auf (strukturelle) Gleichheit.

Sequenzen

Operationen
auf
Sequenzen

Iteration

Objekte und
Identität

Objekte

Objekte und
Variablen

Identität

**Identität und
Gleichheit**

Identität von
Literalen

Erweiterte
Zuweisungen:
Änderbare und
nicht änderbare
Strukturen

Zyklische
Datenstrukturen

Jetzt können wir auch genauer sagen, was es mit veränderlichen (*mutable*) und unveränderlichen (*immutable*) Datentypen auf sich hat:

- Instanzen von veränderlichen Datentypen können modifiziert werden. Daher muss man bei Zuweisungen wie $x = y$ aufpassen:
Operationen auf x beeinflussen auch y .
 - Beispiel: Listen (`list`)
- Instanzen von unveränderlichen Datentypen können nicht modifiziert werden. Daher sind Zuweisungen wie $x = y$ völlig unkritisch:
Da man das durch x bezeichnete Objekt nicht verändern kann, besteht keine Gefahr für y .
 - Beispiele: Zahlen (`int`, `float`, `complex`), Strings (`str`), Tupel (`tuple`)

Sequenzen

Operationen
auf
Sequenzen

Iteration

Objekte und
Identität

Objekte

Objekte und
Variablen

Identität

Identität und
Gleichheit

Identität von
Literalen

Erweiterte
Zuweisungen:
Änderbare und
nicht änderbare
Strukturen

Zyklische
Datenstrukturen

- Bei veränderlichen Datentypen wird jedesmal ein neues Objekt erzeugt, wenn ein Literal ausgewertet wird:

```
def meine_liste():  
    return []  
a = []  
b = []  
c = meine_liste()  
d = meine_liste()  
# id(a), id(b), id(c) und id(d)  
# sind garantiert unterschiedlich.
```

Sequenzen

Operationen
auf
Sequenzen

Iteration

Objekte und
Identität

Objekte

Objekte und
Variablen

Identität

Identität und
Gleichheit

**Identität von
Literalen**

Erweiterte
Zuweisungen:
Änderbare und
nicht änderbare
Strukturen

Zyklische
Datenstrukturen

- Bei unveränderlichen Datentypen darf Python ein existierendes Objekt jederzeit ‚wiederverwenden‘, um Speicherplatz zu sparen, muss aber nicht.

```
def mein_tupel():  
    return ()  
a, b, c, d = (), (), mein_tupel(), mein_tupel()  
# a, b, c, d eventuell (nicht garantiert!) identisch.  
  
a = 2  
b = 2      # a und b sind vielleicht identisch.  
c = a     # a und c sind garantiert identisch.  
d = 1 + 1 # a und d sind vielleicht identisch.
```

- Wegen dieser Unsicherheit ist es meistens falsch, unveränderliche Objekte mit `is` zu vergleichen.

Sequenzen

Operationen
auf
Sequenzen

Iteration

Objekte und
Identität

Objekte

Objekte und
Variablen

Identität

Identität und
Gleichheit

**Identität von
Literalen**

Erweiterte
Zuweisungen:
Änderbare und
nicht änderbare
Strukturen

Zyklische
Datenstrukturen

Eine Anmerkung zu None:

- Der Typ `NoneType` hat nur einen einzigen Wert (der der Name `None` zugeordnet ist). Daher ist es egal, ob ein Vergleich mit `None` per Gleichheit oder per Identität erfolgt.
- Es hat sich eingebürgert, Vergleiche mit `None` immer als `x is None` bzw. `x is not None` und nicht als `x == None` bzw. `x != None` zu schreiben.
- Der Vergleich per Identität ist auch (geringfügig) effizienter.

Sequenzen

Operationen
auf
Sequenzen

Iteration

Objekte und
Identität

Objekte

Objekte und
Variablen

Identität

Identität und
Gleichheit

Identität von
Literalen

Erweiterte
Zuweisungen:
Änderbare und
nicht änderbare
Strukturen

Zyklische
Datenstrukturen



- Bei änderbaren Strukturen wird die Änderungsoperation benutzt!

Python-Interpreter

```
>>> x = [ 0, 1 , 2]
>>> y = x
>>> x += [ 3 ]
>>> print(x)
[0, 1, 2, 3 ]
>>> print(y)
[0, 1, 2, 3 ]
>>> x = x + [ 4 ]
>>> print(y)
[0, 1, 2, 3 ]
```

Visualisierung

Sequenzen

Operationen
auf
Sequenzen

Iteration

Objekte und
Identität

Objekte

Objekte und
Variablen

Identität

Identität und
Gleichheit

Identität von
Literalen

Erweiterte
Zuweisungen:
Änderbare und
nicht änderbare
Strukturen

Zyklische
Datenstrukturen

- Bei nicht änderbaren Strukturen wird einfach in eine normale Zuweisung expandiert.

Python-Interpreter

```
>>> nationality = "Spanish"
>>> institution = nationality
>>> institution += " Inquisition"
>>> print(institution)
Spanish Inquisition
>>> print(nationality)
Spanish
```

Sequenzen

Operationen
auf
Sequenzen

Iteration

Objekte und
Identität

Objekte

Objekte und
Variablen

Identität

Identität und
Gleichheit

Identität von
Literalen

Erweiterte
Zuweisungen:
Änderbare und
nicht änderbare
Strukturen

Zyklische
Datenstrukturen

- **Zyklische** Datenstrukturen sind solche, in denen ein Teil der Struktur identisch mit sich selbst ist!
- Dies kann zu merkwürdigen Fehlern führen! Benutzen Sie solche Strukturen nur, wenn Sie wirklich wissen, was Sie tun!

Python-Interpreter

```
>>> l1 = [1, 3, 5]
>>> l1[2] = l1
>>> print(l1)
[1, 3, [1, 3, [...]]]
```

Visualisierung

Sequenzen

Operationen
auf
Sequenzen

Iteration

Objekte und
Identität

Objekte

Objekte und
Variablen

Identität

Identität und
Gleichheit

Identität von
Literalen

Erweiterte
Zuweisungen:
Änderbare und
nicht änderbare
Strukturen

**Zyklische
Datenstrukturen**



- Es gibt drei Sequenztypen in Python: Strings, Tupel und Listen
- Operationen auf Sequenzen: Verkettung, Wiederholung, Indizierung, Mitgliedschaftstest, Slicing, Iteration
- Zur Iteration dient die `for`-Schleife
- Diese kann, ebenso wie die `while`-Schleife, durch einen optionalen `else`-Zweig ergänzt werden.
- Wichtige Hilfsfunktionen für `for`-Schleifen sind: `range`, `enumerate`, `zip` und `reversed`.
- Alle Werte sind tatsächlich **Objekte**.
- Die Struktur von **änderbaren** Objekten ist veränderlich ... egal über welchen Namen wir zugreifen!
- **Zyklische** Datenstrukturen sollten normalerweise vermieden werden!

Sequenzen

Operationen
auf
Sequenzen

Iteration

Objekte und
Identität

Objekte

Objekte und
Variablen

Identität

Identität und
Gleichheit

Identität von
Literalen

Erweiterte
Zuweisungen:
Änderbare und
nicht änderbare
Strukturen

Zyklische
Datenstrukturen