Informatik I: Einführung in die Programmierung

6. Python-Programme; Sequenzen

Albert-Ludwigs-Universität Freiburg

Prof. Dr. Peter Thiemann

29. Oktober 2025

1 Programme schreiben

Zeilenumbruch

- Umbrechen, wenn Zeilen zu lang.
- Implizite Fortsetzung mit öffnenden Klammern und Einrückung (siehe PEP8):

Lange Zeilen

```
foo = long_function_name(var_one, var_two,
                         var_three, var_four)
def long_function_name(
        var_one, var_two, var_three,
        var_four):
    print(var_one)
```

Kommentare im Programmtext

- Kommentiere dein Programm!
- Programme werden öfter gelesen als geschrieben!
- Auch der Programmierer selbst vergisst…
- Nicht das Offensichtliche kommentieren, sondern Hintergrundinformationen:
  Warum ist das Programm so geschrieben und nicht anders?
- Möglichst in Englisch kommentieren.

Syntax von Kommentaren

- Der Rest einer Zeile nach # ist Kommentar.
- Blockkommentare: Zeilen, die jeweils mit # beginnen und genauso wie die restlichen Zeilen eingerückt sind beziehen sich auf die folgenden Zeilen.

Block-Kommentare

```
def fib(n : int) -> int:
    # this is a double recursive function
    # runtime is exponential in the argument
    if n == 0:
```

- Fließtext-Kommentare kommentieren einzelne Zeilen.

Schlechte und gute Kommentare

```
x = x + 1 # Increment x (BAD)
y = y + 1 # Compensate for border (GOOD)
```

docstring-Kommentare

- #-Kommentare sind nur für den Leser.
- docstring-Kommentare geben dem Programmierer Informationen.
- Ist der erste Ausdruck in einer Funktion `f` oder einem Programm (Modul) ein String, so wird dieser der *docstring* der Funktion, der beim Aufruf von `help(f)` ausgegeben wird.
- Konvention: Benutze den mit drei "-Zeichen eingefassten String, der über mehrere Zeilen gehen kann.

docstring

```
def fib(n):
    """Computes the n-th Fibonacci number.
The argument must be a positive integer.
"""
    ...
```

2 Sequenzen

- Strings
- Listen und Tupel
- Tuple Unpacking

Sequenzen

Sequenztypen in Python

- Strings — `str`
- Tupel — `tuple`
- Listen — `list`

Programmieren mit Sequenzen

- Gemeinsame Operationen
- Kontrollfluss: Iteration (`for`-Schleifen)

Strings

- Kennen wir schon…

Listen und Tupel

- Sowohl ein Tupel als auch eine Liste ist eine Sequenz von Objekten.
- Tupel werden in runden, Listen in eckigen Klammern notiert:
  `(2, 1, 0)` vs. `["red", "green", "blue"]`.
- Tupel und Listen können beliebige Objekte enthalten, natürlich auch andere Tupel und Listen:
  `([18, 20, 22, "Null"], [("spam", [])])`
- Die Typannotation für ein Tupel bzw. eine Liste soll auch den Typ der Elemente (als Typparameter in eckigen Klammern) benennen:

```
st  : tuple[str,int,bool] = ("red", 0, True)
fl  : list[float] = [3.1415, 1.4142, 2.71828]
ill : list[list[int]] = [[42], [32, 16, 8]]
```

Mehr zu Tupeln

- Klammern um Tupel können weggelassen werden, sofern dadurch keine Mehrdeutigkeit entsteht:

```
>>> mytuple = 2, 4, 5
>>> print(mytuple)
(2, 4, 5)
>>> mylist = [(1, 2), (3, 4)] # Klammern notwendig
>>> onetuple = (42,)
>>> print(onetuple)
(42,)
```

- Ausnahme: Ein-elementige Tupel schreiben sich so `(42,)`.

Tuple Unpacking

- Die Anweisung `a, b = 2, 3` ist eine komponentenweise Zuweisung von *Tupeln* (Tuple Unpacking < Pattern Matching).
- Gleichwertig zu `a = 2` gefolgt von `b = 3`.
- Tuple Unpacking funktioniert auch mit Listen und Strings und lässt sich sogar schachteln:

```
>>> [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]
```

3 Operationen auf Sequenzen

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

Sequenzen

- Strings, Tupel und Listen haben etwas gemeinsam: Sie enthalten untergeordnete Objekte in einer bestimmten Reihenfolge und erlauben direkten Zugriff auf die einzelnen Komponenten mittels Indizierung.
- Typen mit dieser Eigenschaft heißen Sequenztypen, ihre Instanzen 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"`

Verkettung

```
>>> 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):
  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


```
>>> print("*" * 20)
********************
>>> print([None, 2, 3] * 3)
[None, 2, 3, None, 2, 3, None, 2, 3]
>>> print(2 * ("Artur", ["est", "mort"]))
('Artur', ['est', 'mort'], 'Artur', ['est', 'mort'])
```

Indizierung

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

```
>>> 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
```

Test auf Mitgliedschaft: Der `in`-Operator

- **`item in`** *seq* (*seq* ist ein Tupel oder eine Liste):
  `True`, wenn `seq` das Element `item` enthält.
- **`substring in string`** (`string` ist ein String):
  `True`, wenn `string` den Teilstring `substring` enthält.

```
>>> 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

Ausschneiden von ‚Scheiben‘ aus einer Sequenz


```
>>> 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 $i, i+1, \ldots, j-1$:

```
>>> assert ("do", "re", 5, 7)[1:3] == ("re", 5)
```

- Ohne *i* beginnt der Bereich an Position 0:

```
>>> assert ("do", "re", 5, 7)[:3] == ("do", "re", 5)
```

- Ohne *j* endet der Bereich am Ende der Folge:

```
>>> assert ("do", "re", 5, 7)[1:] == ("re", 5, 7)
```

- Der slice Operator `[:]` liefert eine Kopie der Folge:

```
>>> assert ("do", "re", 5, 7)[:] == ("do", "re", 5, 7)
```

Slicing: Erklärung (2)

- Keine Indexfehler beim Slicing. Bereiche ausserhalb der Folge sind leer.

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

- Auch Slicing kann ‚von hinten zählen‘.
  Z.B. liefert seq`[-3:]` die drei letzten Elemente.

Typkonversion

list und tuple konvertieren zwischen den Sequenztypen. str liefert Druckversion.

```
>>> 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']"
>>> "".join(['a', 'b', 'c'])
'abc'
```

Weitere Sequenzoperationen 1

- `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`

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

Weitere Sequenzoperationen 2

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

Weitere Sequenzoperationen 3

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

4 Iteration

- Nützliche Funktionen
- Beispiele

Iteration

Durchlaufen von Sequenzen mit der `for`-Schleife


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

Visualisierung

Iteration (2)

`for` funktioniert mit allen Sequenztypen

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

Syntax der for-Schleife und Terminologie

```
1 for var in expr:
2     suite
```

- `for` und `in` sind Schlüsselworte
- Zeile 1: Schleifenkopf
- Zeile 2-: Schleifenrumpf `suite` eingerückter Block von Anweisungen
- Schleifenvariable: *var* im Schleifenkopf
- (Schleifen-) Iteration: ein Durchlauf (Ausführung) des Schleifenrumpfs

`break`, `continue`, `else`

Die drei folgenden Anweisungen beeinflussen den Ablauf der Schleife:

- `break` im Schleifenrumpf beendet die Schleife vorzeitig.
- `continue` im Schleifenrumpf beendet die aktuelle Schleifeniteration vorzeitig, d.h. springt zum Schleifenkopf und setzt die Schleifenvariable auf den nächsten Wert.
- Schleifen können einen `else`-Block haben. Dieser wird nach Beendigung der Schleife ausgeführt, und zwar genau dann, wenn die Schleife *nicht* mit `break` verlassen wurde.

break, continue und else: Beispiel

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

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

Nützliche Funktionen im Zusammenhang mit `for`-Schleifen

Einige Funktionen tauchen häufig im Zusammenhang mit `for`-Schleifen auf:

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

range

- Konzeptuell erzeugt `range` eine Folge von Indizes für Schleifendurchläufe:
  - `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, ..., start + i * step`
    solange `stop - (start + i * step) > 0` (für `step > 0`).
  - Entsprechendes gilt für `step < 0`.
- `range` erzeugt *keine* Liste oder Tupel, sondern einen sog. Iterator (später).

range: Beispiele

```
>>> 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
```

zip (1)

- Die Funktion `zip` nimmt eine oder mehrere Sequenzen und liefert eine Sequenz von Tupeln mit korrespondierenden Elementen.
- Auch `zip` erzeugt keine Liste, sondern einen Iterator; `list` erzeugt daraus eine richtige Liste.

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

zip (2)

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

```
>>> for xyz in zip("ham", "spam", range(5, 10)):
...     x, y, z = xyz
...     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.

reversed

- Die Funktion `reversed` ermöglicht das Durchlaufen einer Sequenz in umgekehrter Richtung.

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

Beispiel Iteration (I)

Fakultätsfunktion

Zu einer positiven ganzen Zahl soll die Fakultät berechnet werden.

$$0! = 1 \qquad (n+1)! = (n+1) \cdot n! \qquad (1)$$

Schritt 1: Bezeichner und Datentypen

Entwickle eine Funktion `factorial`, die die Fakultät einer positiven ganzen Zahl berechnet. Eingabe ist

- `n : int` (mit `n >= 0`)

Ausgabe ist ein `int`.

Nächste Schritte

Schritt 2: Funktionsgerüst

```
def factorial(
        n : int # assume n >= 0
        ) -> int:
    # fill in
    return
```

Schritt 3: Beispiele

```
assert factorial(0) == 1
assert factorial(1) == 1
assert factorial(3) == 6
```

Ergebnis

```
def factorial(
        n : int
        ) -> int:
    result = 1
    # result == 0!
    for i in range(n):
        # result == i!
        result = (i + 1) * result
        # result == (i+1)!
    return result
```

Beispiel Iteration (II)

Produkt einer Liste

Aus einer Liste von ganzen Zahlen soll das Produkt berechnet werden.

```
product([]) == 1
product([ x[0], ..., x[i], x[i+1] ])
            == product([ x[0], ..., x[i] ]) * x[i+1]
```

Schritt 1: Bezeichner und Datentypen

Entwickle eine Funktion `product`, die das Produkt einer Liste von ganzen Zahlen berechnet. Eingabe ist

- `xs : list[int]`

Ausgabe ist wieder eine Zahl `int`, das Produkt der Elemente der Eingabe.

Nächste Schritte

Schritt 2: Funktionsgerüst

```
def product(
        xs : list[int]
        ) -> int:
    # fill in
    return
```

Schritt 3: Beispiele

```
assert(product([]) == 1)
assert(product([42]) == 42)
assert(product([3,2,1]) == 6)
assert(product([1,-1,1]) == -1)
```

Neuer Schritt: Gerüst zur Verarbeitung von Sequenzen I

Ist ein Argument eine Sequenz (Liste, Tupel, String, ...), dann ist es naheliegend, dass diese Sequenz durchlaufen wird.

```
def product(
        xs : list[int]
        ) -> int:
    # fill in
    for i in range(len(xs)):
        ... # fill in action for each element xs[i]
    return
```

Ergebnis

```
def product(xs : list[int]) -> int:
    result = 1     # result == product([])
    for i in range(len(xs)):
        # result == xs[0] * xs[1] * ... * xs[i-1]
        result = result * xs[i]
        # result == xs[0] * xs[1] * ... * xs[i-1] * xs[i]
        # i = i + 1
    return result

assert(product([]) == 1)
assert(product([42]) == 42)
assert(product([3,2,1]) == 6)
assert(product([1,-1,1]) == -1)
```

Alternatives Gerüst zur Verarbeitung von Sequenzen

- Schleifenvariable durchläuft direkt die Elemente der Sequenz, statt die Indizes

```
def product(
        xs : list[int]
        ) -> int:
    result = 1
    for x in xs:
        result = result * x
    return result

assert(product([]) == 1)
assert(product([42]) == 42)
assert(product([3,2,1]) == 6)
assert(product([1,-1,1]) == -1)
```

Zusammenfassung

- Sequenzen: Oberbegriff für Strings, Tupel und Listen
- Die Typen von Tupeln und Listen haben Typparameter, mit denen der Typ der Elemente angegeben wird.
- Listen sind veränderlich, Tupel nicht
- Zuweisung an mehrere Variable mit Tuple unpacking
- Sequenzoperationen: Verkettung, Wiederholung, Indizierung, Mitgliedschaft, Slicing und Iteration
- Iteration mit der `for`-Schleife
- Checkliste für Programmierung mit Iteration