

Informatik I: Einführung in die Programmierung

6. Python-Programme schreiben, kommentieren, starten und entwickeln

Albert-Ludwigs-Universität Freiburg



Prof. Dr. Peter Thiemann

09. November 2021

1 Programme



Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

Operationen
auf
Sequenzen

Iteration

- Programme = konkretisierte Algorithmen?
- Ja, aber nicht immer!
- Folge von Anweisungen und Ausdrücken, die einen bestimmten Zweck erfüllen sollen.
 - Interaktion mit der Umwelt (Benutzer, Sensoren, Dateien)
 - Unter Umständen nicht terminierend (OS, Sensorknoten, ...)
 - Auf jeden Fall meistens länger als 4 Zeilen!

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

Operationen
auf
Sequenzen

Iteration

2 Programme schreiben



Programme

**Programme
schreiben**

Programme
starten

Programme
entwickeln

Sequenzen

Operationen
auf
Sequenzen

Iteration

- 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)
```

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

Operationen
auf
Sequenzen

Iteration

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

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

Operationen
auf
Sequenzen

Iteration

- 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)
```

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

Operationen
auf
Sequenzen

Iteration

- #-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.  
    """  
  
    ...
```

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

Operationen
auf
Sequenzen

Iteration

- Programme werden in Dateien abgelegt.
- Als Dateiname wähle *Modulname*.py
- Die Dateierweiterung .py zeigt an, dass es sich um ein Python-Programm handelt.
- *Windows*: Wähle immer *Alle Dateien* beim Sichern damit nicht .txt angehängt wird.

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

Operationen
auf
Sequenzen

Iteration

3 Programme starten



Programme

Programme
schreiben

**Programme
starten**

Programme
entwickeln

Sequenzen

Operationen
auf
Sequenzen

Iteration

5 Wege ein Programm zu starten



- Starten mit explizitem Aufruf von Python3
- Starten durch Import
- Starten in einer IDE

Beispielprogramm: `example.py`

```
print("Hello world")
```

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

Operationen
auf
Sequenzen

Iteration

Shell

```
# python3 example.py
```

```
Hello world
```

- Voraussetzungen:

- Die Datei `example.py` liegt im aktuellen Ordner.
- Die Pfad-Variable (PATH) wurde so gesetzt, dass der Python-Interpreter gefunden wird.

- Wird normalerweise bei der Installation geleistet.

- Kann „per Hand“ nachgetragen werden:

- *Windows*: Systemsteuerung → System und Sicherheit → Erweiterte Systemeinstellungen → Erweitert → Umgebungsvariablen
- *Unix*: Setzen der PATH-Variablen im entsprechenden Login-Skript oder in der Shell-Konfigurationsdatei (z.B. `~/.bash_profile`)

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

Operationen
auf
Sequenzen

Iteration

- Nach dem Start von Python im Ordner, in dem `example.py` liegt:

Python-Interpreter

```
>>> import example
Hello world
```

- *Beachte:* Angabe ohne die Dateierweiterung!
- Die Anweisungen in der Datei werden nur beim ersten Import ausgeführt.

Python-Interpreter

```
>>> import example
Hello world
>>> import example
>>>
```

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

Operationen
auf
Sequenzen

Iteration

4 Programme entwickeln



■ IDE

Programme

Programme
schreiben

Programme
starten

**Programme
entwickeln**

IDE

Sequenzen

Operationen
auf
Sequenzen

Iteration

IDE = Integrated development environment



Editor aufrufen, Programm in der Shell starten, wieder Editor starten, ...

IDEs sind einsetzbar für:

- Projektverwaltung
- Programm editieren
- Ausführen
- Testen und *Debuggen*
- Dokumentation erzeugen
- ...

Gibt es in den verschiedensten Komplexitäts- und Qualitätsabstufungen.

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

IDE

Sequenzen

Operationen
auf
Sequenzen

Iteration

- Programme bearbeiten mit **Texteditor** (nicht Word!), möglichst mit integrierter Syntaxprüfung.
- Oder mit IDE: IDLE oder VisualStudioCode oder ...
- Werden Zeilen zu lang, müssen sie **umgebrochen** werden.
- **Kommentare** sind hilfreich, um das Programm zu verstehen.
- Es gibt Block-, Fließtext und `docstring`-Kommentare

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

IDE

Sequenzen

Operationen
auf
Sequenzen

Iteration

5 Sequenzen



- Strings
- Listen und Tupel
- Tupel Unpacking

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

Strings

Listen und Tupel

Tupel Unpacking

Operationen
auf
Sequenzen

Iteration

Pythons Sequenztypen

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

Programmieren mit Sequenzen

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

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

Strings

Listen und Tupel

Tupel Unpacking

Operationen
auf
Sequenzen

Iteration

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)
```

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

Strings
Listen und Tupel
Tupel Unpacking

Operationen
auf
Sequenzen

Iteration

$s1 + s2$	Verkettung von Sequenzen
$s[0]$	Indizierung ab 0 (Zugriff aufs erste Element)
$s[1:3]$	Teilsequenz (vom 2. bis 4. Element)

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

Strings

Listen und Tupel

Tupel Unpacking

Operationen
auf
Sequenzen

Iteration

- Kennen wir schon...

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

Strings

Listen und Tupel

Tupel Unpacking

Operationen
auf
Sequenzen

Iteration

- Sowohl ein **Tupel** als auch eine **Liste** ist eine **Sequenz von Objekten**.
- 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", [])])

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen
Strings

Listen und Tupel
Tupel Unpacking

Operationen
auf
Sequenzen

Iteration

Hauptunterschied zwischen Listen und Tupeln

- **Listen** sind *veränderlich* (mutable).
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.)

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen
Strings

Listen und Tupel
Tupel Unpacking

Operationen
auf
Sequenzen

Iteration

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

Python-Interpreter

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

- **Ausnahme:** Ein-elementige Tupel schreiben sich so ("I'm just a ...",).

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen
Strings

Listen und Tupel
Tupel Unpacking

Operationen
auf
Sequenzen

Iteration

- 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:

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]
```

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

Strings

Listen und Tupel

Tupel Unpacking

Operationen
auf
Sequenzen

Iteration

6 Operationen auf Sequenzen



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

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

**Operationen
auf
Sequenzen**

Verkettung

Wiederholung

Indizierung

Mitgliedschaftstest

Slicing

Typkonversion

Weitere Sequenz-
Funktionen

- 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" == "spamsam"`

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"`

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

Operationen
auf
Sequenzen

Verkettung
Wiederholung
Indizierung
Mitgliedschaftstest
Slicing

Typkonversion
Weitere Sequenz-
Funktionen

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]
```

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

Operationen
auf
Sequenzen

Verkettung
Wiederholung
Indizierung
Mitgliedschaftstest
Slicing
Typkonversion
Weitere Sequenz-
Funktionen

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'])
```

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

Operationen
auf
Sequenzen

Verkettung

Wiederholung

Indizierung

Mitgliedschaftstest

Slicing

Typkonversion

Weitere Sequenz-
Funktionen

- Sequenzen können von vorne und von hinten indiziert werden.
- Bei Indizierung von vorne hat das erste Element Index 0.
- Zur Indizierung von hinten dienen 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
```

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

Operationen
auf
Sequenzen

Verkettung

Wiederholung

Indizierung

Mitgliedschaftstest

Slicing

Typkonversion

Weitere Sequenz-
Funktionen

Wo sind die Zeichen?

- Python hat keinen eigenen Datentyp für Zeichen (*chars*).
Für Python ist ein Zeichen 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'
```

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

Operationen
auf
Sequenzen

Verkettung
Wiederholung
Indizierung
Mitgliedschaftstest
Slicing

Typkonversion
Weitere Sequenz-
Funktionen

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.

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
```

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

Operationen
auf
Sequenzen

Verkettung

Wiederholung

Indizierung

Mitgliedschaftstest

Slicing

Typkonversion

Weitere Sequenz-
Funktionen

- *Slicing*: 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
```

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

Operationen
auf
Sequenzen

Verkettung
Wiederholung
Indizierung
Mitgliedschaftstest

Slicing

Typkonversion
Weitere Sequenz-
Funktionen

- `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)`
- Ohne i beginnt der Bereich an Position 0:
`("do", "re", 5, 7)[:3] == ("do", "re", 5)`
- Ohne j endet der Bereich am Ende der Folge:
`("do", "re", 5, 7)[1:] == ("re", 5, 7)`
- Der slice Operator `[:]` liefert eine **Kopie** der Folge:
`("do", "re", 5, 7)[:]` == `("do", "re", 5, 7)`

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

Operationen
auf
Sequenzen

Verkettung
Wiederholung
Indizierung
Mitgliedschaftstest

Slicing

Typkonversion
Weitere Sequenz-
Funktionen

Slicing: Erklärung (2)



- Beim Slicing gibt es keine Indexfehler. Bereiche jenseits des Endes der Folge sind leer.

Python-Interpreter

```
>>> "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.

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

Operationen
auf
Sequenzen

Verkettung

Wiederholung

Indizierung

Mitgliedschaftstest

Slicing

Typkonversion

Weitere Sequenz-
Funktionen

list und **tuple** konvertieren zwischen den Sequenztypen. **str** wirkt anders.

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

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

Operationen
auf
Sequenzen

Verkettung
Wiederholung
Indizierung
Mitgliedschaftstest
Slicing

Typkonversion
Weitere Sequenz-
Funktionen

- `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
```

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

Operationen
auf
Sequenzen

Verkettung

Wiederholung

Indizierung

Mitgliedschaftstest

Slicing

Typkonversion

Weitere Sequenz-
Funktionen

- `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`, aber mit `elem1 and elem2 and elem3 and ...`

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

Operationen
auf
Sequenzen

Verkettung

Wiederholung

Indizierung

Mitgliedschaftstest

Slicing

Typkonversion

Weitere Sequenz-
Funktionen

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

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

Operationen
auf
Sequenzen

Verkettung

Wiederholung

Indizierung

Mitgliedschaftstest

Slicing

Typkonversion

Weitere Sequenz-
Funktionen

■ Nützliche Funktionen

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

Operationen
auf
Sequenzen

Iteration

Nützliche
Funktionen

- Durchlaufen von Sequenzen mit for-Schleifen:

Python-Interpreter

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

Visualisierung

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

Operationen
auf
Sequenzen

Iteration

Nützliche
Funktionen

Iteration (2)

- 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!
```

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

Operationen
auf
Sequenzen

Iteration

Nützliche
Funktionen

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

- Zeile 1: **Schleifenkopf**
- Zeile 2-: **Schleifenrumpf** *block* eine oder mehrere Anweisungen
- **Schleifenvariable**: *var* im Schleifenkopf
- **Schleifeniteration**: ein Durchlauf (Ausführung) des Schleifenrumpfs

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

Operationen
auf
Sequenzen

Iteration

Nützliche
Funktionen

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-Zweig** haben. Dieser wird nach Beendigung der Schleife ausgeführt, und zwar genau dann, wenn die Schleife *nicht* mit **break** verlassen wurde.

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

Operationen
auf
Sequenzen

Iteration

Nützliche
Funktionen

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!")
```

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

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

Operationen
auf
Sequenzen

Iteration

Nützliche
Funktionen

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

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

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

Operationen
auf
Sequenzen

Iteration

Nützliche
Funktionen

- Konzeptuell erzeugt range eine Folge von Indexen 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, ..., stop-1
- range erzeugt *keine* Liste, sondern einen sog. **Iterator** (später).

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

Operationen
auf
Sequenzen

Iteration

Nützliche
Funktionen

range: Beispiele



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
```

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

Operationen
auf
Sequenzen

Iteration

Nützliche
Funktionen

zip (1)

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

Python-Interpreter

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

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

Operationen
auf
Sequenzen

Iteration

Nützliche
Funktionen

zip (2)

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

Python-Interpreter

```
>>> 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.

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

Operationen
auf
Sequenzen

Iteration

Nützliche
Funktionen

reversed

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

Python-Interpreter

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

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

Operationen
auf
Sequenzen

Iteration

Nützliche
Funktionen

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 `fact`, die die Fakultät einer positiven ganzen Zahl berechnet. Eingabe ist

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

Ausgabe ist ein `int`.

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

Operationen
auf
Sequenzen

Iteration

Nützliche
Funktionen

Schritt 2: Funktionsgerüst

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

Schritt 3: Beispiele

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

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

Operationen
auf
Sequenzen

Iteration

Nützliche
Funktionen

```
def fact(  
    n : int  
    ) -> int:  
    result = 1  
    for i in range(1, n+1):  
        result = result * i  
    return result
```

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

Operationen
auf
Sequenzen

Iteration

Nützliche
Funktionen

Produkt einer Sequenz

Aus einer Sequenz von Zahlen soll das Produkt der Zahlen berechnet werden.

Schritt 1: Bezeichner und Datentypen

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

- `xs : Sequence[Number]`

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

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

Operationen
auf
Sequenzen

Iteration

Nützliche
Funktionen

Für diese Typannotation muß ein Import erfolgen:

- `from numbers import Number`

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

Operationen
auf
Sequenzen

Iteration

Nützliche
Funktionen

Nächste Schritte

Schritt 2: Funktionsgerüst

```
def product(  
    xs : Sequence[Number]  
    ) -> Number:  
    # 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)
```

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

Operationen
auf
Sequenzen

Iteration

Nützliche
Funktionen

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

```
def product(  
    xs : Sequence[Number]  
    ) -> Number:  
    # fill in  
    for x in xs:  
        ... # fill in action for each element  
    return
```

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

Operationen
auf
Sequenzen

Iteration

Nützliche
Funktionen

```
def product(  
    xs : Sequence[Number]  
    ) -> Number:  
    result = 1    # product([])  
    for x in xs:  
        result = result * x  
    return result
```

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

Operationen
auf
Sequenzen

Iteration

Nützliche
Funktionen

- **Sequenzen**: Oberbegriff für Strings, Tupel und Listen
- 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

Programme

Programme
schreiben

Programme
starten

Programme
entwickeln

Sequenzen

Operationen
auf
Sequenzen

Iteration

Nützliche
Funktionen