Informatik I: Einführung in die Programmierung

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

Albert-Ludwigs-Universität Freiburg

Prof. Dr. Peter Thiemann

24. November 2020

1 Programme

Programme

- 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!

2 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 geben:
  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 Benutzer Informationen.
- Ist der erste Ausdruck in einer Funktion oder einem Programm (Modul) ein String, so wird dieser der *docstring*, der beim Aufruf von `help` 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 speichern

- 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.

3 Programme starten

5 Wege ein Programm zu starten

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

Beispielprogramm: example.py

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

Starten mit explizitem Aufruf von Python3

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`-Variable im entsprechenden Login-Skript oder in der Shell-Konfigurationsdatei (z.B. `~/.bash_profile`)

Starten als Skript

Shell

```
# example.py
Hello world
```

- Voraussetzungen:
  - Die Datei `example.py` liegt im aktuellen Ordner.
  - *Windows*: .py wurde als Standard-Dateierweiterung für Python registriert.
  - *Unix*: Die erste Zeile in der Datei `example.py` ist:
    `#!/usr/bin/env python3`
    und die Datei hat das `x`-Bit (ausführbare Datei) gesetzt.

Starten durch Klicken

- Wenn `.py` als Standard-Dateierweiterung für Python registriert ist (geht eigentlich bei allen Plattformen mit Desktop-Oberfläche), kann die Datei durch Klicken (oder Doppelklicken) gestartet werden.
- Leider wird nur kurz das Shell-Fenster geöffnet, mit Ende des Programms verschwindet es wieder.
- *Abhilfe*: Am Ende der Datei die Anweisung `input()` in das Programm schreiben.
- *Allerdings*:
  - Bei Fehlern verschwindet das Fenster
  - Beim Aufruf können keine Parameter übergeben werden.
- Für fertig entwickelte Programme mit GUI geeignet.

Starten durch Import

- 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
>>>
```

4 Programme entwickeln

- IDE
- IDLE

IDE = Integrated development environment

Editor aufrufen, Programm in der Shell starten, wieder Editor starten, ...
IDEs sind einsetztbar für:

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

Gibt es in den verschiedensten Komplexitäts- und Qualitätsabstufungen.

Pythons IDE: IDLE

Wohlmöglich benannt nach Eric Idle.

- Ist 100% in Python geschrieben und benutzt die *tkinter* GUI (graphical user interface).
- Läuft auf allen Plattformen.
- Multi-Fenster-Texteditor mit Syntaxkennzeichnung, multipler Zurücknahme, smarter Einrückung.
- Enthält ein Fenster mit Python-Shell.
- Rudimentäre Debug-Möglichkeiten.
- Beschreibung siehe: `http://docs.python.org/3/library/idle.html`.

IDLE Demo

Zusammenfassung

- 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

5 Sequenzen

- Strings
- Tupel und Listen
- Tupel Unpacking

Sequenzen

Pythons Sequenztypen

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

Programmieren mit Sequenzen

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

Beispiel zu Sequenzen

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)
```

Sequenzoperationen

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

Strings

- Strings in Python enthalten Unicode-Zeichen.
- Strings werden meistens `"auf diese Weise"` angegeben, es gibt aber viele alternative Schreibweisen.

Tupel und Listen

- Ein Tupel bzw. 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", [])])`

Hauptunterschied zwischen Tupeln und Listen

- 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.)

Mehr zu Tupeln

- 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: Einelementige Tupel schreiben sich `("so",)`.

Tuple Unpacking

- Bei `a, b = 2, 3` werden *Tupel* komponentenweise zugewiesen (Tuple Unpacking $<$ Pattern Matching).
- 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]
```

6 Operationen auf Sequenzen

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

Sequenzen

- Strings, Tupel und Listen haben etwas gemeinsam: Sie enthalten andere 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

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]
```

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

- 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
```

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'
```

Mitgliedschaftstest: 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
```

Slicing

- *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
```

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)`
- 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)`

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.

Typkonversion

list, tuple, und str konvertieren zwischen den Sequenztypen.

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'
```

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`

Python-Interpreter

```
>>> 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.

7 Iteration

- Nützliche Funktionen

Iteration

- 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

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!
tasty!
```

Syntax und Terminologie

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

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

break, continue, else

Im Zusammenhang mit Schleifen sind die folgenden drei Anweisungen interessant:

- `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.

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.
```

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 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).

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
```

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')]
```

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.

reversed

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

Python-Interpreter

```
>>> 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 `fac`, 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 fac(
        n : int # assume n >= 0
        ) -> int
    # fill in
    return
```

Schritt 3: Beispiele

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

Ergebnis

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

Beispiel Iteration (II)

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.

Einschub: Typannotation

Für diese Typannotation müssen einige Importe erfolgen:

- `from collections.abc import Sequence`
- `from numbers import Number`

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)
```

Neuer Schritt: Gerüst zur Verarbeitung von Sequenzen

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
```

Ergebnis


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

Zusammenfassung

- 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