Fortgeschrittene Programmierung

Kapitel 2: Python
Kontrollstrukturen und Funktionen

Prof. Dr. Thomas Wieland
WS 2022/2023

Abbildungen z.T. aus: K. Sierra/B. Bates: Java von Kopf bis Fuß. O'Reilly, 2006.

Programmstrukturierung

- Blöcke kennzeichnen zusammengehörige Programmabschnitte
- In Python durch Einrückungen (in Java durch Klammern {})
  - Bei Verschachtelungen in verschiedenen Tiefen
  - Block wird durch Doppelpunkt begonnen

```
if x < y :
    print ("x ist kleiner")
    x = y
```

- Jedes Objekt hat Identität, Typ und Wert

if-Anweisungen

- Bedingungen werden in Python durch die if-Anweisung ausgedrückt (genau wie in Java)

Häufige Fehler

- **Falsche Schreibweise** des Gleichheitsoperators (==) als Zuweisungsoperator (=)
- **Fehlender Doppelpunkt** nach der Bedingung und nach `else`
- Falsche Einrückung bei **Verschachtelungen**

- Alternativen mit if/else

```
if n % 2 == 0:
    print("Zahl gerade")
else:
    print("Zahl ungerade")
```

Mehrere Bedingungen

- **Mehrere Bedingungen** können mit den logischen Operatoren `and`, `or` und `not` kombiniert werden
  - Dabei einzelne Bedingungen in Klammern

```
if (t < 0) or (t > 100):
    print("Falsche Eingabe")
```

- **Verschachtelte Bedingungen** können mit `elif` kombiniert werden

```
if x < 0:
    y = 0
elif (x >= 0) and (x <= a):
    y = x
else:
    y = a
```

- Eine **Mehrfachauswahl** (switch in Java) gibt es in Python nicht!

Schleifen

- Einfache Schleife mit **Anfangsüberprüfung**:

```
while Bedingung:
    Anweisungen
```

- Beispiel:

```
i = 1
while i <= 10:
    print(i*i)
    i += 1
```

```
while mehrBaelle:
    weiterJonglieren()
```

- **break** bewirkt in einer Schleife, dass diese bei dessen Erreichen beendet wird.
- **continue** bewirkt in einer Schleife, dass der Rest des Schleifenkörpers übersprungen und mit der nächsten Iteration fortgefahren wird.

Schleifen mit for

- Schleifen mit for:

```
for variable in objekt:
    Anweisungen
```

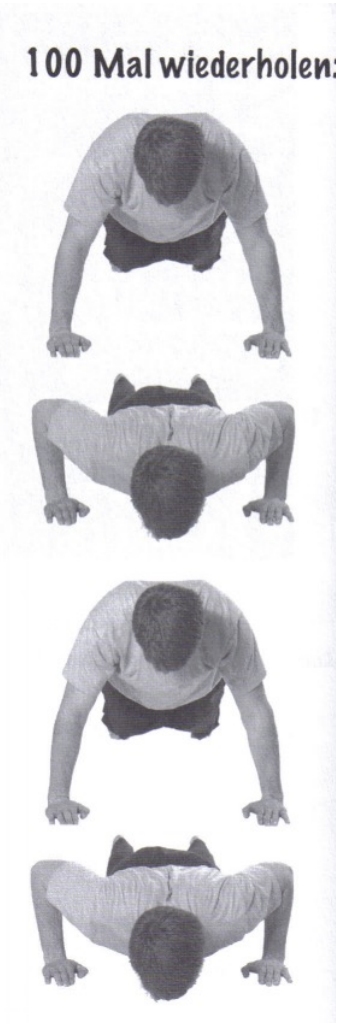
  - **Objekt**: Alle iterierbaren Objekte, z.B. Bereiche, Listen, Strings, Dictionaries

- Beispiel: Summe der Quadratzahlen

```
s = 1
for i in range(1, n+1)
s += i * i
```

- Beispiel: Initialisieren einer Matrix m x n

```
a = [[0 for x in range(n)] for y in range(m)]
for i in range(m):
    for j in range(n):
        a[i][j] = i+j
```



Quelle: K. Sierra/B. Bates: Java von Kopf bis Fuß. O'Reilly, 2006

List Comprehension

- Einfache Listen lassen sich mit `range` etc. erzeugen
- Komplexere Listen mit **list comprehension**

```
>> b = [i**2 for i in range(0,10) if i%2 == 0]
>> b
[0, 4, 16, 36, 64]
```

- Man kann damit auch **geschachtelte Schleifen** nachbilden

```
>> c = [(i,j) for i in range(1,5) if i%2 == 0
              for j in range(1,5) if j%2!=0]
>> c
[(2, 1), (2, 3), (4, 1), (4, 3)]
```

Funktionen

- Definiert mit Schlüsselwort def
- Nach **Funktionsname** und Parameterliste folgt Doppelpunkt
- Der nachfolgende Block wird bei Aufruf ausgeführt
- Bis zum **Ende** des Blocks oder einem return
- **Rückgabewert**: beliebiger Typ
- Auch Funktionen sind **Objekte**!

```
def hello():
    print “Hello World!“
```

Namensraum (Scope)

- Jede Funktion hat für ihre Variablen einen lokalen **Namensraum**
- Gleichnamige **globale** Variablen werden überdeckt
- Mit `global` kann eine Variable global gemacht werden
- `nonlocal` bringt eine Variable in den **nächst höheren Namensraum** (bei verschachtelten Definitionen)

Parameterübergabe

- Nach **Position**: Bei `def f(x, y, z):` erfolgt der Aufruf als `f(a, b, c)`, falls a für x, b für y und c für z gesetzt werden soll
- Nach **Namen**: Aufruf kann auch als `f(a, z = c, y = b)` oder auch `f(y = b, x = a, z = c)` erfolgen
  - Positionale Parameter müssen aber vor den benannten stehen!
- Mit **optionalen Werten**: Ein Parameter kann einen Vorgabewert erhalten: `def f(x, y, z = 3):` Aufruf kann dann ohne diesen sein `f(a, b)` oder mit `f(a, b, c)` oder auch `f(a, y = b, z = 4)`
- **Optionale Parameter**: Eine Funktion kann auch beliebig viele optionale Parameter erhalten, die in ein Tupel kommen

```
def s(x, *tup):
    s = x
    for i in tup: s += i
    return s
```

  - Die Parameter können auch als Dictionary übergeben werden `def m(x, **lex)`

Rückgabewerte

- Bei mehreren Rückgabewerten wird ein **Tupel** zurückgeliefert

```
def return_two():
    return “x“, “y“
h = return_two()
```

- h ist dann (“x“, “y“)

- Listen als Rückgaben sind dann aber nur einzelne Werte

Anonyme Funktionen

- Normale Funktionen haben eine Zugriffsnamen
- Anonyme Funktionen haben keinen Namen bei der Konstruktion, können an beliebigen Stellen im Code stehen und können einen Namen später erhalten
  -> **Lambda-Ausdrücke**

```
inc = lambda i : i+1
inc(3)
4
```

- Syntax: `lambda` *`Argumentenliste : Ausdruck`*

Map und Filter

- `map` wendet eine Funktion auf alle Elemente eines sequentiellen Datentyps (oder mehrerer) an
  - Geht auch mit Lambda-Ausdrücken

```
l = range(10)
m = list(map(lambda x: x*x+1, l))
```

- `filter` liefert alle Elemente einer Sequenz zurück, für welche die gegebene Bedingung wahr ist

```
m = list(filter(lambda x: x%2 == 0, l))
```

Generator

- Generator ist eine Funktion, die nicht ein einzelnes Resultat produziert, sondern eine Folge davon
- Ähnlich zu **statischen Variablen** in Java und C++!
- Statt return wird yield verwendet:

```
def gen(n) :
    z = n
    while True:
        yield z
        z -= 1
```

- Liefert alle Zahlen von n abwärts, eine pro Aufruf
- Nach der **Initialisierung** weitere Aufrufe mit Methode `__next__()`

Iteratoren

- Iterator liefert zu Beginn das erste Element, bei jedem weiteren Aufruf das jeweils nächste
  - Ist der Vorrat erschöpft, kommt die Ausnahme *StopIteration*

```
lst = [1, 2, 3, 4, 5]
it = iter(lst)
while True:
    print(next(it))
```

  - Statt `next(it)` wäre auch `it.__next__()` möglich
- Wird eingesetzt für Lesen in Dateien,
  List Comprehension usw.
- Funktioniert also mit allen sequentiellen Datentypen

Zusammenfassung

- Die **for-Schleife** führt Iterationen n-mal aus, die **while-Schleife** bis zu einer Abbruchbedingung
- Eine Schleife wird durch **break** sofort beendet, durch **continue** wird zum nächsten Schleifendurchgang gesprungen
- Schleifen können beliebig **verschachtelt** sein
- **Funktionen** werden mit def definiert
- **Parameter von Funktionen** können nach Namen und mit Vorgabewerten übergeben werden
- Funktionen können **mehrere Werte** zurückliefern
- Funktionen bilden eigene **Namensräume** für ihre Variablen
- Mit **lambda** können anonyme Funktion definiert werden
- **map und filter** wenden Funktionen auf alle Elemente einer Liste an
- Eine **Generatorfunktion** liefert bei jedem Aufruf einen neuen Wert zurück
- Ein **Iterator** durchläuft einen sequentiellen Datentyp


Aha, eine Funktion kann zwei Rückgaben haben. Dann kann ich ja an Lissy und Sally gleichzeitig schreiben…