Montag, 17.02.2014

Table of Contents

Mit Variablen, Funktionen und Kontrollstrukturen (Verzweigungen, Schleifen) haben wir den **Kern** einer Programmiersprache. Was eine Programmiersprache zusätzlich auszeichnet, sind sogenannte **Datenstrukturen** und **Bibliotheken**.

- Beispiele für **Datenstrukturen**: In Delphi haben wir eine Klasse für **Listen** kennengelernt (→ Hessischer Bildungsserver). Sinnvoll ist auch die Unterstützung von Zeichenketten (**Strings**) oder von **assoziativen Arrays** (→ PHP - in Python nennt man sie **Dictionaries**).
- Beispiele für **Bibliotheken**: Wir benötigen zum Beispiel eine Bibliothek für die Erstellung grafischer Oberflächen (**GUI**). In Python gibt es hier beispielsweise **Tkinter** (= **Tk**-**Inter**face). Oder **Sockets** für den Zugriff auf das Netzwerk (→ The Python Standard Library)

OOP in Python

In Python kann man auch objektorientiert programmieren (, muss man aber nicht!). Wir untersuchen das Bankkonto-Beispiel in Python:

```
class Bankkonto:                                                    # 001
   """Einfache Bankkonto-Klasse"""                                  # 002
                                                                    # 003
   def __init__(self,startbetrag):                                  # 004
      """Konstruktor: erzeugt Bankkonto"""                          # 005
      self.kontostand = startbetrag                                 # 006
                                                                    # 007
   def einzahlung(self, betrag):                                    # 008
      self.kontostand = self.kontostand + betrag                    # 009
                                                                    # 010
   def auszahlung(self, betrag):                                    # 011
      self.kontostand = self.kontostand - betrag                    # 012
                                                                    # 013
   def anzeigen(self):                                              # 014
      print self.kontostand                                         # 015
```

- In Zeile 001 wird eine Klasse mit dem Schlüsselwort `class` vereinbart.

- Zeile 002 ist einfach nur ein Dokumentationsstring.
- Zeile 004 - 006: Da eine Klasse bekanntlich aus **Eigenschaften** und **Methoden** besteht, wird in der `__ init__`-Methode die Eigenschaft `kontostand` initialisiert. Dies geschieht beim Aufruf des Konstruktors: `konto1 = Bankkonto(100).` `konto1` ist ein Objekt der Klasse Bankkonto.

  Aufgabe 1: Welchen Wert hat danach die Eigenschaft `kontostand`?

  Nun muss ich irgendwie ausdrücken, dass der Kontostand nur zu diesem Objekt `konto1` gehört; wenn ich also mehrere Konten vereinbare, muss (!) der Kontostand eine Eigenschaft des jeweiligen Bankkontos sein! Das erreiche ich mit der Schreibweise `self`, die gerade unser jeweilges Bankkonto bezeichnet, im Beispiel also `konto1`. Damit nicht genug: ich muss ja auch den Methoden sagen, auf welchem Bankkonto sie aus- oder einzahlen sollen, und auch hier wird der Parameter `self` verwendet.

> **Hinweis 01:** Man muss den Parameter `self` beim Aufruf der Methode NICHT hinschreiben, das macht Python "automatisch" über den Namen des Objektes. Beispielaufruf hierzu:
>
> ```
> konto1.anzeigen()
> ```

- In Zeile 008, 011 und 014 werden die drei Methoden `einzahlung`, `auszahlung` und `anzeigen` der Klasse `Bankkonto` vereinbart. Die ersten beiden Methoden erwarten **einen** Parameter, die Methode `anzeigen` wird dagegen ohne Parameter aufgerufen.

  Aufgabe 2: Mit welchem Parameter wird die Methode `einzahlung` aufgerufen? Und warum wird sie nur mit *einem* Parameter aufgerufen?

  Fehlt noch ein Test der Klasse `Bankkonto`:

```
# Ein Konto erzeugen
konto1 = Bankkonto(100)

# Kontostand
konto1.anzeigen()   #  Ausgabe: 100

# Auf das konto1 einzahlen
konto1.einzahlung(4711)
konto1.anzeigen()   # Ausgabe: 4811

# Vom konto1 abheben
konto1.auszahlung(123456789)
konto1.anzeigen()   # Ausgabe: -123451978
```

Am besten gefällt mir natürlich

```
>>> help(Bankkonto)
Help on class Bankkonto in module __main__:

class Bankkonto
 |  Einfache Bankkonto-Klasse
 |
 |  Methods defined here:
 |
 |  __init__(self, startbetrag)
 |      Konstruktor: erzeugt Bankkonto
 |
 |  anzeigen(self)
 |
 |  auszahlung(self, betrag)
 |
 |  einzahlung(self, betrag)
```

Aufgaben

- **Aufgabe 3**: Kann man mehrere Konten vereinbaren? Kommen sich die Konten gegenseitig ins Gehege? (Mit Begründung!)
- **Aufgabe 4 Methode transfer**: In der Klasse Bankkonto fehlt noch eine Methode zum Überweisen! Entwickle eine Methode `transfer` mit folgender Signatur:

  ```
  def transfer(self,betrag,konto):
  ```

  Abspeichern unter `bankkonto_ext.py`
- **Aufgabe 5 Vererbung**: Entwerfe und implementiere eine Klasse Taschengeldkonto in Python. (→ Taschengeldkonto: Man darf nicht überziehen!).

  Abspeichern unter `taschengeldkonto.py`. Link zur Vererbung in Python: http://www.wspiegel.de/pykurs/python14.htm
- **Aufgabe 6 Zahlenraten:** Nett ist auch das Zahlenraten-Spiel. Dazu muss man wissen, woher man in Python Zufallszahlen bekommt: hierfür gibt es in Python das Modul `random`. Aber `help(random)` ist diesmal keine gute Idee, da der Hilfetext recht umfangreich ist. Betrachte stattdessen folgendes Beispiel:

  ```
  import random
  zzahl = random.randint(1, 100)
  print "Zahl zwischen 1 und 100: ", zzahl
  ```

  Hierzu wieder die Python-Hilfe:

  ```
  randint(self, a, b)
        Return random integer in range [a, b], including both end points.
  ```

  Abspeichern unter `zahlenraten.py`

- **Aufgabe 7 Logik in Python:**

In der nebenstehenden Übersicht sind die logische Operatoren aufgeführt: **and**, **or**, **xor** und **not** sind einfach, dafür gibt es in Python die Operatoren:
and, **or**, **^** (= **xor**) und **not**

Die Operatoren: **nor**, **nand**,
=> (Implikation)
und **<=>** (Äquivalenz)
sind zu programmieren!

Hinweis zu **xor** in Python:
a **xor** b → `a ^ b` (das „Dach"!)

Wahrheitswerte in Python:
`True` bzw. `False`

→ GROSS- und Kleinschreibung beachten!

Abspeichern unter dem Namen **`logik.py`**

```
 b1     b2      b1 or b2
TRUE   TRUE    TRUE
TRUE   FALSE   TRUE
FALSE  TRUE    TRUE
FALSE  FALSE   FALSE

 b1     b2      b1 nor b2
TRUE   TRUE    FALSE
TRUE   FALSE   FALSE
FALSE  TRUE    FALSE
FALSE  FALSE   TRUE

 b1     b2      b1 and b2
TRUE   TRUE    TRUE
TRUE   FALSE   FALSE
FALSE  TRUE    FALSE
FALSE  FALSE   FALSE

 b1     b2      b1 nand b2
TRUE   TRUE    FALSE
TRUE   FALSE   TRUE
FALSE  TRUE    TRUE
FALSE  FALSE   TRUE

 Implikation
 b1     b2      b1 => b2
TRUE   TRUE    TRUE
TRUE   FALSE   FALSE
FALSE  TRUE    TRUE
FALSE  FALSE   TRUE

 Aequivalenz
 b1     b2      b1 <=> b2
TRUE   TRUE    TRUE
TRUE   FALSE   FALSE
FALSE  TRUE    FALSE
FALSE  FALSE   TRUE

 XOR
 b1     b2      b1 xor b2
TRUE   TRUE    FALSE
TRUE   FALSE   TRUE
FALSE  TRUE    TRUE
FALSE  FALSE   FALSE
```