Der Kochrezept-Vergleich

Ein beliebter Vergleich, der oft gebracht wird, um die Bedeutung des Programmierens zu veranschaulichen: wir haben einen Rezeptschreiber (Programmierer) und einen Koch (Computer). Der Koch ist allerdings sehr pingelig! Anstatt zu schreiben "gebe 3 Eier zum Mehl", müssen wir ihm sagen "Nimm das erste Ei. Klopfe es auf. Gib den Inhalt in die Schüssel. Nimm das zweite Ei. Klopfe es auf . . . "

Je nach Programmiersprache variiert die Auffassungsgabe des Kochs. Unser Python-Koch ist schon recht selbstständig. Jeder, der schon mal in einer Low-Level Sprache wie C, C++ oder Fortran gearbeitet hat, wird das schnell merken. Alle anderen, die bisher noch nicht programmiert haben, seien hier nur einmal darauf hingewiesen, damit sie sich bei später auftretenden Problemen damit trösten können, dass sie in anderen Programmiersprachen *mindestens* genauso viele Schwierigkeiten hätten ;-)

Python als Interpreter Sprache

Python ist im Gegensatz zu Programmiersprachen wie C, C++ oder Fortran eine *Interpretersprache*. Das bedeutet, dass geschriebener Python-Code nicht kompiliert werden muss. Stattdessen wird der Code zur Laufzeit des Programmes vom Python Interpreter abgearbeitet.

Wenn wir ein Programm mit dem Namen hello_world.py starten möchten, müssen wir in der Konsole daher einfach nur

```
python hello_world.py
```

eingeben.

(Für Python 3 (nicht auf unseren Rechnern verfügbar) ist der Befehl

```
python3 hello_world.py
```

Hello World

Um einen ersten Eindruck einer Programmiersprache zu geben, beginnen die meisten Tutorials mit einem simplen Programm, das nur die Worte "Hello World" ausgibt. Auch dieses Tutorium macht da keinen Unterschied.

Wir legen eine Textdatei **hello_world.py** mit folgendem Inhalt an:

```
print "Hello World" # für Python 3: print("Hello World") (mit Klammern!)
```

Das “Hello World” Programm zeigt schon gut, worin Pythons große Stärke liegt: die gute Lesbarkeit des Quellcodes! In nur einer Zeile steht alles, was nötig ist, um das Programm auszuführen.

Was passiert hier genau? In dem Programm wird die print Funktion aufgerufen. Als Argument bekommt sie eine Zeichenkette (einen sogenannten String) geliefert. Dieser String wird dann schließlich beim Ausführen des Programms ausgegeben.

Zum Vergleich “Hello World” in Java:

```
class Main{
    public static void main(String[] args){
        System.out.println("Hello World");
    }
}
```

Datentypen und Variablen

In Python können wir mit folgenden grundlegenden Datentypen arbeiten:

- **Integer**: ganzzahlige Zahlen
- **Float**: Gleitkomma-Zahlen
- **String**: Zeichenketten
- **Boolean**: Logische Werte, können True oder False sein
- **NoneType**: Spezieller Datentyp, der z.B. als Default-Argument in Funktionen verwendet wird (kommt später)

Jeder dieser Datentypen kann nun in einer Variablen gespeichert werden. Dies funktioniert mit einer einfachen Zuweisung wie in folgendem Beispiel:

```
ganze_zahl = 10
komma_zahl = 1.234
komplexe_zahl = 3.123 + 2j
string_variable = "Hallo Welt"
boolsche_variable = True
none_variable = None
```

Hier haben wir vier Variablen mit den Namen *ganze_zahl*, *komma_zahl*, *komplexe_zahl*, *string_variable* und *boolsche_variable* initialisiert. Im Gegensatz zu anderen Programmiersprachen erkennt der Interpreter von alleine, um was für einen Datentyp es sich handelt. Um zu sehen, welchen Datentyp unsere Variablen nun haben, schreiben wir

```
print type(ganze_zahl)
print type(komma_zahl)
print type(komplexe_zahl)
print type(string_variable)
```

```
print type(boolsche_variable)
print type(none_variable)
```

mit den Resultaten

```
<type 'int'>
<type 'float'>
<type 'complex'>
<type 'str'>
<type 'bool'>
<type 'NoneType'>
```

Ein kleiner Hinweis zu Variablennamen: Grundsätzlich ist es möglich, seine Variablen zu nennen, wie man möchte, solange sie mit einem Buchstaben beginnen. Zahlen am Anfang sind nicht erlaubt. Sonderzeichen dürfen nicht im Variablennamen vorkommen. Darüberhinaus ist es Konvention, Variablennamen klein zu schreiben und mit Unterstrich zu trennen.

Kommentare

Ein weiterer **wichtiger** Punkt für Programmiersprachen generell, der sich zwar nicht auf die Ausführung des Codes auswirkt, aber dafür auf das Verständnis des Lesers, sind Kommentare.

Kommentare lassen sich mithilfe des Raute-Zeichens in ein Python-Skript einfügen. Alles was in derselben Zeile hinter der Raute steht, wird nicht vom Python-Interpreter abgearbeitet

```
print 1, 2, 3 # wird ausgeführt, denn die Raute kommt erst nach dem Befehl
# print 4, 5, 6 wird nicht ausgeführt, da zu Beginn der Zeile eine Raute steht
```

Operatoren

Bisher ist in unseren Programmen noch nicht viel passiert. Das wird sich nun ändern!
Mit Operatoren ist es uns nun möglich, Werte in unseren Programmen miteinander zu vergleichen, Variablen neue Werte zuzuweisen, oder mathematische Ausdrücke zu berechnen.

- Zuweisungsoperator =

Diesen Operator haben wir schon im vorherigen Abschnitt kennengelernt. `python x = 3` Dieser Operator weist einer Variablen einen Wert zu.

- Arithmetische Operatoren +, -, *, /, %, **
  Mit diesen Operatoren können wir einfache arithmetische Ausdrücke berech-

nen. `python print 4*3 print 10/5 print 10/4 print 10+3-11 print 14%3 print 10**2` Ausgabe: “‘ 12 2 # in Python 3: 2.0 2 # in Python 2: 2.5 2 2 100

`Wir können diese natürlich auch mit dem Zuweisungs-Operator kombinieren:`python x = 4*3 + 13/5 print(x) `Ausgabe in Python 3:` 14.6 “‘

Aber in Python 2: `14` Was ist hier passiert? Während Python 3 uns das erwartete Ergebnis liefert, wendet Python 2 bei 13/5 die sogenannte Integer-Division an. Das Ergebnis ist also auch wieder ein Integer (13 durch 5 ist 2 Rest 3).

- Vergleichsoperatoren ==, <, <=, >, >=
  Diese Operatoren vergleichen zwei Werte miteinander und geben einen booleschen Wert zurück. `python print 4 + 3 == 7 print -11 < -13` Ausgabe: “‘ True False

“‘

- Logische Operatoren not, and, or Diese Operatoren akzeptieren als Eingabe boolesche Werte und geben einen booleschen Wert zurück.
  not negiert einen booleschen Wert: `python print not False print not True` Ausgabe: `True False` **and** ergibt *True* wenn beide Eingabewerte *True* sind, ansonsten *False* “‘python print True and True print True and False print False and False

print 1+2==3 and 4*4==16 print 1+2==3 and False x = 7 print x == 7 and x+3 == 10 and True `Ausgabe` True False False True False True `**or**` `ergibt _True_ wenn mindestens einer der beiden Eingabewerte _True_ ist.`python print True or True print True or False print False or False x = 5 print x<5 or x>5 `Ausgabe:` True True False False “‘

Schleifen

In Python gibt es zwei Arten von Schleifen:

For-Schleife

Die for-Schleife wird verwendet, wenn man einen Code-Teil eine bestimmte Anzahl von Malen wiederholen möchte.

Z.B.

```
python   for i in range(10):     print i
```

Ausgabe:

```
0   1   2   3   4   5   6   7   8   9
```

Wichtig ist hierbei die Einrückung nach der for-Anweisung (Konvention sind 4 Leerzeichen pro Einrückung). Alles, was nach der for-Anweisung eingerückt geschrieben wird, wird so oft ausgeführt, wie die for-Schleife durchlaufen wird.

While-Schleife

Die While-Schleife wird benutzt, um eine Bedingung zu prüfen, und wenn diese erfüllt ist, den nachfolgenden Code auszuführen

Beispiel:

```
python   summe = 0   i = 0   while summe < 20:     i = i + 1
summe = summe + i
```

Mit *break* aus einer Schleife ausbrechen

Manchmal gibt es Fälle, in denen wir vorzeitig aus einer Schleife ausbrechen wollen. In diesen Fällen hilft uns das Statement *break*

Beispiel:

```
for i in range(10):
    print i
    if i == 4:
        break
```

Ausgabe:

```
0
1
2
3
4
```

Anstatt die komplette Schleife zu durchlaufen, bricht das Programm die Schleife ab, sobald es die break-Anweisung erreicht.

Bedingte Anweisungen (if-statements)

Um ein richtiges Programm schreiben zu können, müssen wir dem Computer mitteilen können, wie er auf bestimmte Situationen reagieren soll. Dafür sind **bedingte Anweisungen** nötig. Die Form dieser **if statements** ist wie folgt:

```
if <Bedingung>:
    <Anweisung 1>
elif <Bedingung 2>:
```

```
    <Anweisung 2>
else:
    <Anweisung 3>
```

ist dabei ein Ausdruck, der einen boolschen Wert zurückgibt. Die Anweisungen sind beliebiger Python-Code.

Beispiel:

```
x = 7

if 0 <= x <= 5:
    print "x lässt sich an einer Hand abzählen"
elif 6 <= x <= 10:
    print "x lässt sich an zwei Händen abzählen"
elif x > 10:
    print "Dieses x ist mir zu groß..."
else:
    print "Dieses x ist mir zu klein..."
```

Was passiert in diesem Beispiel? Zuallererst wird in x der Wert 7 gespeichert. Nun beginnt das if-Statement: die erste boolsche Funktion wird ausgewertet. Da x größer als 5 ist, ergibt der gesamte Ausdruck False. Daher wird der darauf folgende Code nicht ausgewertet. Das Programm springt weiter zum elif-Statement. Da 6 <= 7 <= 10 True ergibt, gibt das Programm den Satz “”x lässt sich an zwei Händen abzählen" aus. Die letzten beiden Bedingungen werden übersprungen da die zweite Bedingung True ergab).

Ein- und Ausgabe

In diesem Kapitel beschäftigen wir uns mit verschiedenen Arten der Ein- und Ausgabe, nämlich:

- Einlesen und Schreiben von Dateien
- Einlesen von Tastatureingaben

Schreiben:

Um eine Datei zu öffnen, benötigen wir den Befehl **open**:

```
datei = open("meine_datei", "w")
```

Das **“w”** steht dabei für *writeable*. Möchte man eine (schon existierende) Datei nur auslesen, benutzt man **“r”** für *readable*. Möchte man an eine bestehende Datei weiteren Text anhängen, muss man **“a”** (für *append*) benutzen.

Als nächstes möchten wir etwas in unsere gerade geöffnete Datei schreiben. Dazu benutzen wir, wie schon im Hello World Programm die print-Funktion:

```
print >> datei, "Dieser Satz steht in der ersten Zeile"
print >> datei, "Und der hier in der zweiten"
```

Anschließend müssen wir die Datei noch schließen.

```
datei.close()
```

In diesem Beispiel gibt es allerdings ein Problem. Angenommen, etwas unvorhergesehenes geschieht, bevor das Programm die Datei schließen kann:

```
datei = open("meine_datei", "w")
print 1/0 # Das wird mit einer Fehlermeldung abbrechen...
datei.close() # diese Zeile wird nie ausgeführt werden
```

In diesem Fall wird f.close() nie ausgeführt. Als Resultat können Speicherprobleme auftreten, oder die Datei wird unlesbar.

Die sichere Variante sieht wie folgt aus:

```
with open("meine_datei", "w") as datei:
    print >> datei, "Dieser Satz steht in der ersten Zeile"
    print >> datei, "Und der hier in der zweiten"
```

Einlesen:

Das Einlesen einer Datei funktioniert ähnlich. Hier können wir zum Einlesen aller Zeilen einen schönen Trick verwenden:

```
with open("meine_datei", "r") as datei:
    for line in datei:
        print line
```

Hier benutzen wir eine for-Schleife, die über alle Zeilen in unserer Datei iteriert. Tatsächlich kann man in Python for-Schleifen sehr vielfältig anwenden. Mehr dazu beim nächsten Mal!

Tastatureingabe:

Mit dem Handling von Tastatureingaben können wir unsere Programme endlich etwas interaktiver gestalten! Die Syntax dafür ist wie folgt:

```
user_input = raw_input("Bitte geben Sie etwas ein: ") # Python 3: input statt raw_input
print "Sie gaben ein:", user_input
```

Zugegeben, das Beispiel ist noch nicht sonderlich spannend, aber wir können nun in Kombination mit if-else Anweisungen Programme schreiben, die auf verschiedene User-Eingaben reagieren können.