# Informatik I

## 13. Python: Programme

Jan-Georg Smaus

Albert-Ludwigs-Universität Freiburg

23. Dezember 2010

# Zuweisungen

# Wert eines Ausdrucks an eine Variable binden

Wir hangeln uns weiter an der Scheme-Präsentation entlang.

# Wert eines Ausdrucks an eine Variable binden

Wir hangeln uns weiter an der Scheme-Präsentation entlang.

Es hieß dort:
Eine Definition ist . . . :
(`define` ⟨*variable*⟩ ⟨*expression*⟩)

- Erster Operand: Name einer Variablen.
- Zweiter Operand: ein Ausdruck.
- Diese Bindung bindet den Namen der Variable an den Wert des Ausdrucks.

# Definitionen in Scheme

```
> (define spam (* 111 111))
> spam
12321
> (define egg (* spam spam))
> egg
151807041
```

# Nun in Python

## Python-Interpreter

```
>>> spam = 111 * 111
>>> spam
12321
>>> egg = spam * spam
>>> egg
151807041
```

# Nun in Python

## Python-Interpreter

```
>>> spam = 111 * 111
>>> spam
12321
>>> egg = spam * spam
>>> egg
151807041
```

Man nennt dies in der imperativen Programierung nicht Definition, sondern Zuweisung (assignment).
Sie verwendet das Symbol $=$ (das somit nicht symmetrisch ist!).

# Definitionen in einem Scheme-Programm

Wir wiederholen den Scheme-Python-Vergleich, doch diesmal in einem richtigen Programm:

**assignment.rkt**

```
(define spam (* 111 111))
spam
(define egg (* spam spam))
egg
```

Ablauf des Programms produziert Ausgabe:

```
12321
151807041
```

# Python-Programme

hello.py

```
# "Richtige" Programme stelle ich in diesem Stil
# dar, nicht als Dialog mit dem Python-Interpreter.
#
# "Richtige" Programme sind komplizierter und
# muessen daher auch mal kommentiert werden. Also
# sollte ich sagen, wie man in Python kommentiert:
#
# Alles, was in einer Zeile auf ein Schweinegatter
# (#) folgt, ist ein Kommentar.

print("Hello, world") # Dies ist das beruehmte
                      # "hello world"-Programm.
```

# Das Python-Programm für das Beispiel

**assignment.py**

```
spam = 111 * 111
print(spam)
egg = spam * spam
print(egg)
```

Das Programm produziert die Ausgabe

```
12321
151807041
```

# Nochmal zu Scheme

Betrachte nun folgendes Programm:

**assignment.rkt**

```
(define spam (* 111 111))
spam
(define egg (* spam spam))
egg
(define spam (* egg egg))
spam
```

Was ist die Ausgabe?

# Nochmal zu Scheme

Betrachte nun folgendes Programm:

**assignment.rkt**

```
(define spam (* 111 111))
spam
(define egg (* spam spam))
egg
(define spam (* egg egg))
spam
```

Was ist die Ausgabe?

```
spam: Für diesen Namen gibt es schon eine Definition.
```

# Und nun wieder Python

**assignment.py**

```
spam = 111 * 111
print(spam)
egg = spam * spam
print(egg)
spam = egg * egg
print(spam)
```

Was ist die Ausgabe?

# Und nun wieder Python

**assignment.py**

```
spam = 111 * 111
print(spam)
egg = spam * spam
print(egg)
spam = egg * egg
print(spam)
```

Was ist die Ausgabe?

```
12321
151807041
23045377697175681
```

# Variablen
## Vergleich funktionale vs. imperative Programmierung

- In funktionalen Programmen kann man eine Variable an den Wert eines Ausdrucks binden, aber die Variable behält diesen Wert dann "für immer" (lokale Variablen mittels `let` sind hierzu kein Widerspruch).
- In imperativen Programmen wird einer Variablen ein Wert zugewiesen. Dieser Wert kann jederzeit durch eine neue Zuweisung überschrieben werden.

# Variablen
## Vergleich funktionale vs. imperative Programmierung

- In funktionalen Programmen kann man eine Variable an den Wert eines Ausdrucks binden, aber die Variable behält diesen Wert dann "für immer" (lokale Variablen mittels `let` sind hierzu kein Widerspruch).
- In imperativen Programmen wird einer Variablen ein Wert zugewiesen. Dieser Wert kann jederzeit durch eine neue Zuweisung überschrieben werden.
- Man sagt: in der imperativen Programmierung gibt es veränderlichen Zustand oder auch destruktive Updates.
- Imperative Programme sind vielleicht intuitiver, aber funktionale Programme lassen sich leichter als korrekt beweisen.

# Variablen
## Vergleich funktionale vs. imperative Programmierung

- In funktionalen Programmen kann man eine Variable an den Wert eines Ausdrucks binden, aber die Variable behält diesen Wert dann "für immer" (lokale Variablen mittels `let` sind hierzu kein Widerspruch).
- In imperativen Programmen wird einer Variablen ein Wert zugewiesen. Dieser Wert kann jederzeit durch eine neue Zuweisung überschrieben werden.
- Man sagt: in der imperativen Programmierung gibt es veränderlichen Zustand oder auch destruktive Updates.
- Imperative Programme sind vielleicht intuitiver, aber funktionale Programme lassen sich leichter als korrekt beweisen.
- Auch wenn dieser Unterschied nur für die Paradigmen in Reinform gilt, so ist er doch fundamental!

# In Scheme

Warum habe ich hier jetzt Programme (im Unterschied zu REPL/Interpreter) eingeführt?

# In Scheme

Warum habe ich hier jetzt Programme (im Unterschied zu REPL/Interpreter) eingeführt?
In REPL/Interpreter lässt sich der Unterschied nicht demonstrieren:

```
> (define spam (* 111 111))
> spam
12321
> (define egg (* spam spam))
> egg
151807041
> (define spam (* egg egg))
> spam
23045377697175681
```

In der REPL in Scheme ist Überschreiben von Definitionen möglich.

# Michael Sperber schreibt:

*Es ist erstaunlich schwierig, eine REPL in allen Aspekten intuitiv, flexibel und konsistent zu gestalten. (Nach ein paar Jahren Scheme-Standardisierung würde ich sagen nahezu unmöglich.)*

*Die Grundidee der REPL ist, daß das Programm inkrementell erweitert und geändert wird — entsprechend sind auch Umdefinitionen möglich.*

# Funktionen

# Interpreter vs. Programm

Man kann Funktionen (Prozeduren) im Interpreter schreiben, aber für diese Vorlesung ist es zweckmäßiger, sie in ein richtiges Programm zu schreiben.

# Programm aufrufen
## Direkt aus der Shell

Man kann ein Programm, sagen wir `spam.py`, in der Shell starten:

**Shell**

```
# python3 spam.py
```

Das Programm wird dann ausgeführt und produziert eine Ausgabe oder auch nicht.

# Programm aufrufen
## Über den Interpreter

Man kann aber auch in der Shell den Interpreter starten:

**Shell**

```
# python3
Python 3.1.2 (r312:79147, Apr 15 2010, 12:35:07)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license"
for more information.
```

und dann das Programm importieren:

**Python-Interpreter**

```
>>> from spam import *
```

Dies erlaubt uns, wie in der Scheme-REPL, Ausdrücke auswerten zu lassen, die die Funktionen des Programms verwenden.
So werden wir es machen.

# Quadrat einer Zahl berechnen

In Scheme:

square.rkt

```
(define square
  (lambda (x)
    (* x x)))
(square 3)
```

# Quadrat einer Zahl berechnen

In Scheme:

square.rkt

```
(define square
  (lambda (x)
    (* x x)))
(square 3)
```

In Python:

square.py

```
def square(x):
    return x*x
print(square(3))
```

Die ersten zwei Zeilen sind die Definition einer Funktion, die das Quadrat einer Zahl berechnet. Die dritte Zeile ist ein Aufruf der `print`-Funktion.

# Quadrat einer Zahl berechnen

**square.py**

```
def square(x):
    return x*x
print(square(3))
```

**Python-Interpreter**

```
>>> from square import *
9
>>> square(10)
100
```

Wir sehen: Der Import führt das Programm aus, was die Ausgabe 9 produziert. Dann kann man im Interpreter weitere Ausdrücke auswerten.

# Ausführung imperativer Programme

Bei imperativen Programmen ist der Zustand veränderlich. Die Programmzeilen sind Befehle, die nacheinander ausgeführt werden. Folgende alternative Definition verdeutlicht dies:

square2.py

```
def square(x):
    res = x*x
    return res
print(square(3))
```

# Noch eine Alternative

square3.py

```
def square(x):
    x = x*x
    return x
print(square(3))
```

Was passiert hier?

# Noch eine Alternative

**square3.py**

```
def square(x):
    x = x*x
    return x
print(square(3))
```

Was passiert hier?

**Python-Interpreter**

```
>>> from square3 import *
9
```

Es funktioniert ohne Problem!

# return

In einer Funktionsdefinition in imperativen Sprachen muss man einer Funktion ausdrücklich befehlen, dass sie einen Wert zurückgeben soll: `return`.

**square.py**

```
def square(x):
    return x*x
print(square(3))
```

# Code hinter dem return

**square4.py**

```
def square(x):
    print("egg")
    return x*x
    print("spam")
print(square(3))
```

**Python-Interpreter**

```
>>> from square4 import *
```

# Code hinter dem return

**square4.py**

```
def square(x):
    print("egg")
    return x*x
    print("spam")
print(square(3))
```

**Python-Interpreter**

```
>>> from square4 import *
egg
9
>>> square(10)
```

# Code hinter dem return

**square4.py**

```
def square(x):
    print("egg")
    return x*x
    print("spam")
print(square(3))
```

**Python-Interpreter**

```
>>> from square4 import *
egg
9
>>> square(10)
egg
100
```

# Bemerkungen zu `return`

- Code, der hinter dem `return` steht, wird nicht ausgeführt (toter Code).
- Wird nicht explizit ein Wert zurückgegeben, ist das Resultat der spezielle Wert `None`.

# Einrückung
## Eine Python-Besonderheit

- Irgendwie muss festgelegt werden, wo eine Funktionsdefinition zu Ende ist. Scheme verwendet hierfür Klammern, andere Sprachen Schlüsselwörter wie z.B. `end`.

# Einrückung
## Eine Python-Besonderheit

- Irgendwie muss festgelegt werden, wo eine Funktionsdefinition zu Ende ist. Scheme verwendet hierfür Klammern, andere Sprachen Schlüsselwörter wie z.B. `end`.
- Python markiert die Blockstruktur eines Programs durch Einrücken.
- Die Regel ist ganz einfach: Alles, was zu der Funktion gehört, muss gegenüber der Funktionsdefinition eingerückt sein.

**Python:**

**Programming**
**the way**
**Guido**
**indented it**

# Nochmals square4

square4.py

```
def square(x):
    print("egg")
    return x*x
    print("spam")
print(square(3))
```

# Andere Einrückung

**square5.py**

```
def square(x):
    print("egg")
    return x * x
  print("spam")
print(square(3))
```

# Andere Einrückung

**square5.py**

```
def square(x):
    print("egg")
    return x * x
  print("spam")
print(square(3))
```

**Python-Interpreter**

```
>>> from square5 import *
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "square5.py", line 4
    print("spam")
                ^
IndentationError: unindent does not match any
outer indentation level
```

# Nochmal andere Einrückung

square6.py

```
def square(x):
    print("egg")
    return x * x
print("spam")
print(square(3))
```

# Nochmal andere Einrückung

**square6.py**

```
def square(x):
    print("egg")
    return x * x
print("spam")
print(square(3))
```

**Python-Interpreter**

```
>>> from square6 import *
spam
egg
9
>>> square(10)
egg
100
```

# Der Editor emacs

Eine Nebenbemerkung: im Python-Mode des Editors emacs springt beim Tippen des Programms

square4.py

```
def square(x):
    print("egg")
    return x*x
    print("spam")
print(square(3))
```

der Cursor nach der `return`-Zeile auf Einrückungsebene 0. D.h., der Editor merkt, dass nach dem `return` die Funktionsdefinition zu Ende sein sollte.

# Zusammenfassung

- Eine Variable an einen Wert zu binden, bezeichnet man in der imperativen Programierung als Zuweisung. Zuweisungen können später durch eine neue Zuweisung überschrieben werden. Bei imperativen Programmen spielt Zeit eine große Rolle.
- Die Python-Syntax zum Schreiben von Funktionen unterscheidet sich stark von der von Scheme.
- Ein nackter Ausdruck, insbesondere ein Funktionsaufruf, im Programm produziert nicht automatisch eine Ausgabe.
- Man muss Funktionen ausdrücklich befehlen, einen Wert zurückzugeben (sonst: `None`).
- Die Blockstruktur wird in Python durch Einrückung gekennzeichnet.