# Einführung in Python

Albert Brandl

8.10.2010

# Überblick

Mehrere Teile

- Einführung in Python (Was ist das? Wer verwendet es? Wie schaut es aus? ==> Folien
- Beispiel-Anwendung: Programm, das große Dateien in Verzeichnisbaum findet
- Alternativ: Aufbereitung einer Datei mit Kino-Daten
- Falls noch Zeit ist: Aufbereitung der Kino-daten fürs Web

# Was ist Python?

- Interpretierte Programmiersprache
- Legt besonders viel Wert auf Lesbarkeit und Verständlichkeit ==> flache Lernkurve
- Geeignet für imperative, objektorientierte und funktionale Programmierung
- Kann damit genauso gut kleine Scripts schreiben wie umfangreiche Programme
- Interaktiver Interpreter hilft beim "Erkunden" der Sprache

# Anwendungen

- EVE Online (Online-Spiel)
- Solar Wolf (SDL-basiertes Weltraumspiel
- ZOPE / Plone
- Impressive (Präsentationssoftware)
- u.v.m - siehe auch
  http://www.python.org/about/success/

# Implementierungen

- CPython: Ursprüngliche Python-Version, nach wie vor am verbreitetsten
- Jython: Setzt auf der Java Virtual Machine auf, erlaubt Zugriff auf Java-Klassen
- IronPython: Verwendet .NET statt C als Basis ==> Windows-lastig
- Stackless: Python-Implementierung, die ohne Stack auskommt (nur den Heap verwendet)

# Besonderheiten der Syntax

Einrückung definiert Blockanfang und -ende

Beispiel: Iteration in bash

```
for var in 1 2 3; do
    echo "Value of var is ${var}."
done
```

Beginn und Ende der Schleife durch "do" und "done" festgelegt

Beispiel: Iteration in C

```
for (int i = 1; i < 4; i++) {
    printf("Value of var is %d", i);
}
```

Beginn und Ende der Schleife durch geschwungene Klammern festgelegt

- Beispiel in Python:

# Vorteile

- Bin gezwungen, die Struktur auch durch Einrückung darzustellen (in anderen Sprachen muss das z.B. durch Coding Guidelines bewerkstelligt werden)
- Kann mir sicher sein, dass eingerückter Code zusammengehört:

Der folgende C-Code liefert nicht das erwartete Ergebnis

```
for (int i=1; i < 4; i++)
    printf("Value of var is %d", i);
printf("alue of 2 * var is %d", 2 * i);
```

- Weniger "Schreibarbeit" (Keine ";" und "{...}" nötig)
- Bessere Lesbarkeit

# Nachteile

- Muss mehrzeilige Statements speziell behandeln ("trailing backslash"):

```
my_variable = another_variable * a_third_variable / sqrt(another_variable
    yet_another_variable
```

Allerdings kann das weggelassen werden, wenn Ausdruck geklammert ist

- Weniger Gestaltungsspielraum beim Formatieren (Nachteil?)

# Datentypen

- Booleans: True, False
- Zahlen: Integers, Float, Comples

Beispiele: -23, 12.4, complex(2,3)

- Strings: Haben ein Encoding (latin1, utf8, ...)

Beispiele: "abc", 'def', "abc'def", "äöü"

- Unicode: Müssen für Ausgabe umgewandelt werden

Beispiele: u'abc', u'öüä', u"abc'def"

# Wichtige Kontrollstrukturen

- Einfache Statements
- Verzweigungen
- Schleifen

# Einfache Statements

Das pass-Statement wird ignoriert

**Beispiel:**

```
if x < 5:
    pass
elif x < 10:
    print x
else:
    print x ** 2
```

Das break-Statement beendet eine Schleife

**Beispiel:**

```
for i in range(10):
    print i ** 2
    if i > 5:
        break
```

# Noch mehr einfache Statements

Das print-Statement gibt den Wert eines Ausdrucks aus

**Beispiel:**

```
print 2 * 5.8
```

Ab Python 3.0 ist print kein Statement mehr, sondern eine Funktion!

Die Zuweisung weist einer oder mehreren Variablen Werte zu

**Beispiele:**

```
a = 5
b = 2.8
a, b = b, a
```

Vor der Zuweisung werden sämtliche Expressions rechts ausgewertet!

# if ... elif ... else

Ähnlich wie in vielen anderen Sprachen

Beispiel:

```
if x < 5:
    print x
elif x < 10:
    print 2 * x
else:
    print 3 * x
```

# for- und while-Schleifen

Eine "for"-Schleife führt den folgenden Block für jedes Element aus.

Die Elemente müssen in einer geordneten Liste o.ä. vorliegen.

**Beispiel:**

```
for i in [1,2,3]:
    print i
```

Eine "while"-Schleife wird ausgeführt, solange die Bedingung erfüllt ist:

```
while x > 0:
     x = x/2
     print x
```

# Funktionen

Funktionen ermöglichen es, Code zu strukturieren.

Die meisten Funktionen haben einen Namen, man kann aber auch anonyme Funktionen definieren

Funktionen haben eine Parameterliste. Es gibt verschiedene Arten von Parametern.

**Beispiel:**

```
def my_function(par1, par2, *other_pars, **keyword_pars):
    pass
```

- par1 und par2 sind "positionale Parameter"
- other_pars enthält eine Liste aller zusätzlich mitgegebenen, namenlosen Parameter
- keyword_pars enthält ein Dictionary von Parametern mit Namen.

# Funktionen, Teil 2

Angenommen, die oben definierte Funktion wird so aufgerufen:

```
result = my_function(7, 12, "a", "b", "c", x1=55, x2=99)
```

Dann haben die Parameter folgende Werte:

- par1 == 7
- par2 == 1
- other_pars == ["a", "b", "c"]
- keyword_pars = {"x1": 55, "x2": 99}

Wenn am Anfang einer Funktion ein String steht, wird der als Dokumentation interpretiert und in der Variablen __doc__ der Funktion gespeichert.

# Weitere Kontrollstrukturen

raise löst eine Exception aus, try...except...finally erlaubt die Behandlung:

```
def do_something(x):
    if x > 5:
        raise ValueError
try:
    do_something(8)
except ValueError:
    handle_error()
```

List comprehensions erlauben die Definition komplexer Listen:

```
>>> print [2 * x for x in range(6) if x % 2 != 0]
[2, 6, 10]
```

- Objektorientierte Programmierung
- Generatoren
- Module
- Metaprogrammierung

# Workshop, Alternative 1

Aufgabenstellung: Finde alle Dateien in einem Verzeichnisbaum > 1024 Bytes

Constraint: Verwende Python, keinen Bash-Oneliner ;-)

# Workshop, Alternative 2

Aufgabenstellung: Bereite eine Textdatei mit den Beginnzeiten von Filmen in unerschiedlichen Kinos auf. Ziel: eine Liste von Beginnzeiten, damit man schnell entscheiden kann, in welchen Film man geht.

Bonus: Stelle diese Informationen über einen Webserver zur Verfügung, mit Suchmaske