Skriptsprachen: Python
Einführung

Jan Krüger, Alexander Sczyrba

Technische Fakultät
Universität Bielefeld

12. Februar 2018

Python

–

Einführung

Eigenschaften, Erweiterungen, Erste Gehversuche

Python – Versionsgeschichte

- Dynamische Programmiersprache, leider oftmals lediglich als Skriptsprache "missbraucht"
- Erste Version 1991

3.5.2 Aktueller *stable* Release

- Viele Performance- und Detailverbesserungen
- Teilweise veränderte Syntax
- Nicht rückwärtskompatibel zu *2.X*
- Nativer Unicode Support

2.7.12

- *2.X* immer noch sehr verbreitet (z.B. Bestandteil von OSX)
- Ab *2.6* Nachimplementierungen von *3.X*-Funktionalität
- Viele Module, die noch nicht (vollständig) *3.X*-kompatibel sind (z.B. FLASK)
- Aufwärtskompatibilität durch Autoconverter → 2to3

Python – Eigenschaften

- Optimiert auf Lesbarkeit
- Wenige Schlüsselwörter
- Multi OS-Support
- Multiparadigmal: imperative, objekt-/aspektorientierte und funktionale Programmierung
- Sehr gut integriertes Hilfesystem

Python – Interna

- Stark objektorientiert $\rightarrow$ Alles ist ein Objekt!
  - Keine Literale
  - Starke Typisierung
  - Funktionen sind Objekte: Können während der Laufzeit überladen werden $\rightarrow$ *Dependency Injection*
- Blöcke werden durch *TAB*-Einrückungen gekennzeichnet
- Duck Typing: Variablen(!) und nicht das dahinterliegende Objekt sind typisiert

Python – Features und Erweiterungen

Python-Shell	Interaktiver Pythonmodus
IDLE	Built-In Entwicklungsumgebung
iPython	Mächtige alternative interaktive Shell mit viel Magic™
Cython	Teile lassen sich in C extrahieren und maschinenoptimiert ausführen

Das erste Skript

Texteditor	vi(m), emacs, gedit, kate → KEIN Wordprocessor
Dateiendung	*.py
Präambel	*#!/usr/bin/env python*
Dateirechte	chmod u+x <DATEINAME>

```
#!/usr/bin/env python
print("Meine␣erste␣Ausgabe")
```

```
$ chmod u+x test.py
$ ./test.py
```

iPython – Alternative Pythonshell

- Tab-Completion
- Integrierte Hilfe über `?` und `??`
- OS Shell: `!<BEFEHL>`
- Integrierter Debugger
- Befehlshistorie
- Tutorial: `%magic`

Das Werkzeug der Wahl zum interaktiven Experimentieren!

Starten `$ ipython` Beenden `> quit` oder `CTRL^d`

Übung – Erste Gehversuche

Überprüfe die folgenden Ausdrücke in der interaktiven Shell (iPython) und versuche zu erklären, was hier weswegen passiert:

```
> 1/7
> 1.0/7.0
> print 1./7

> 0.9999999999999
> print 0.9999999999999

> 0.9999999999999*9/9
> 0.9999999999999/9*9
```

Python

–

Einführung

Objektreferenzen, Namespaces, Callables, GC

Objekte, Referenzen und Instanzen

Python ist streng Objektorientiert

- „primitive“ Datentypen wie z.B. `int` sind Objekte
- Funktionen sind Objekte
- So gut wie ALLES(!!!) in Python ist ein Objekt

Objekte, Referenzen und Instanzen

```
> 23        # Instanz vom Typ int

> a = 23    # a ist eine Referenz auf eine Instanz
            # vom Typ int mit dem Wert 23

> b = a     # b referenziert auf die gleiche
            # Instanz wie a

> b = 46
```

Welche Werte enthalten a und b ?

Objekte, Referenzen und Instanzen

Grundlegende Funktionen, die ihr kennen müsst:

- dir() Listet alle Objekteigenschaften (Namespaces, Callables) auf
- type() Gibt den Typ eines Objekts wieder
- help() Eingebaute Hilfe

Diese werden auch als `builtin_function_or_method` bezeichnet.

Übung – Wer bin ich und was kann ich?

Gegeben sei der folgende Ausdruck:

```
> text = "Hallo␣Welt!"
```

Wende die Befehle `type()`, `dir()`, und `help()` auf die Variable `text` an. Was passiert hier?

Schaue dir mit Hilfe von `help()` die Hilfe zu Funktionen/Typen an.

Namespaces und Callables

Erstellt euch eine komplexe Zahl `c=1+2j`. Schaut euch den Real- und den Imaginärteil dieser Zahl mittels `real` und `imag` an und abschließend ihre konjugierte komplexe Zahl.

```
> imag(c)
> c.imag

> c.real

> c.conjugate
> c.conjugate()
```

Namespaces und Callables

Erstellt euch eine komplexe Zahl `c=1+2j`. Schaut euch den Real- und den Imaginärteil dieser Zahl mittels `real` und `imag` an und abschließend ihre konjugierte komplexe Zahl.

```
> imag(c)          # ->NameError
> c.imag           # Namespace

> c.real           # Namespace

> c.conjugate      # Das Funktionsobjekt -> Callable
> c.conjugate()    # Aufruf
```

Namespaces vergleichbar mit Feldern (Java) und Callables mit Funktion ($\rightarrow$ müssen `__cal__()`-Funktion implementieren)

Identitätsvergleich vs. Inhaltvergleich

```
> a = "Hello World!"
> b = a
> c = "Hello World!"

> a == b
True
> a == c
True
> a is b      # id(a) == id(b)
True
> a is c      # id(a) == id(c)
False
```

- Identität eines Objekts : `id()`
- Identitätsvergleichs-operator : `is`
- Gleichheitsvergleichs-operator: `==`

Garbage Collection

Python hat ähnlich wie *Java* eine *Garbarge Collection*, die sich ums Aufräumen von nicht benötigten Instanzen kümmert. Eine Instanz kann vom GC gelöscht werden, wenn keine Referenz mehr auf sie zeigt.

```
> a = 23
> a = 46   # keine Referenz mehr auf Instanz
           # (int :: 23)
           # -> kann vom GC aus dem Speicher
           # entfernt werden

> del(a)   # del() entfernt Referenzen explizit
```