

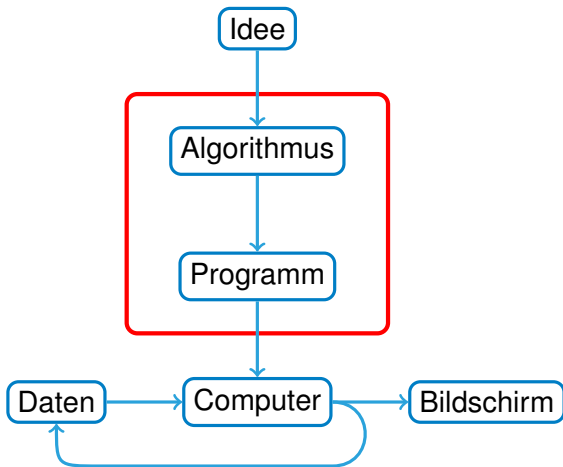
Computegrundlagen Programmieren in Python

F.Uhlig, J.Smiatek, A.Arnold, M.Fyta

Institut für Computerphysik
Universität Stuttgart

Wintersemester 2018/2019

Was ist Programmieren?

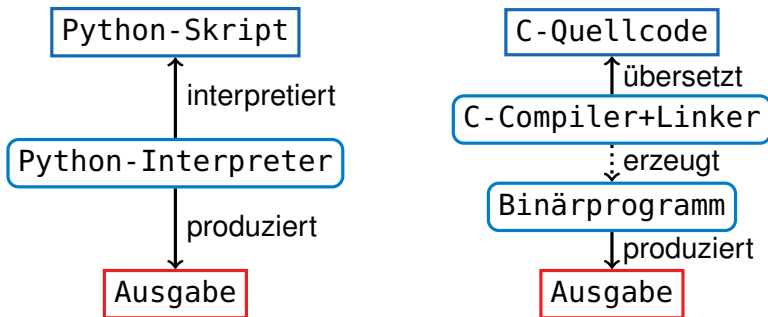


Python



- schnell zu erlernende Programmiersprache
– tut, was man erwartet
- objektorientierte Programmierung ist möglich
- viele Standardfunktionen („all batteries included“)
- breite Auswahl an Bibliotheken
- freie Software mit aktiver Gemeinde
- portabel, gibt es für fast jedes Betriebssystem
- entwickelt von Guido van Rossum, CWI, Amsterdam

Interpretierte- vs. kompilierte Sprachen

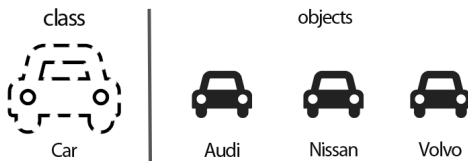


Interpretierte Sprachen

- z.B. Python, Java, C#, Basic, PHP, Shellsprachen
- Das Programm wird vom Interpreter gelesen und ausgeführt
- Es wird kein Maschinencode für den Prozessor übersetzt

Objektorientierte Sprachen

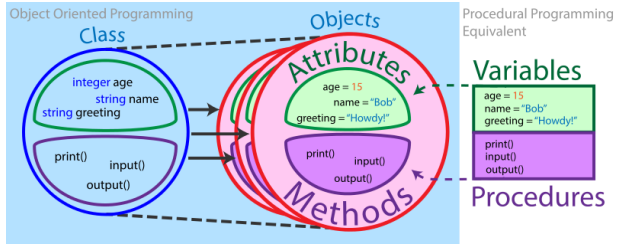
Objekt: Variable, Datenstruktur, etc. (also ein Wert im Speicher der mit einem Kennzeichen identifiziert wird)



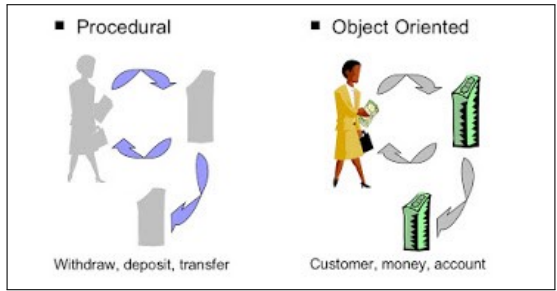
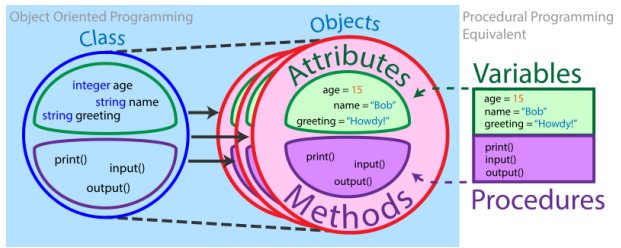
- Objektorientierung (OO) ist ein sprachunabhängiges Programmiermodell
- Ziel ist die bessere Wartbarkeit und Austauschbarkeit von Software durch starke Isolation von Teilproblemen
- Speziell darauf ausgelegt sind z.B. C++, Java, Python,...

Quelle:javatutorial.net

Imperative vs. Objektorientierte Programmierung



Imperative vs. Objektorientierte Programmierung



Informationen zu Python

- aktuelle Versionen 3.7.2
- 2.x ist *noch* weiter verbreitet (z.B. Python 2.7.15rc1 im CIP-Pool)
- diese Vorlesung behandelt daher noch 2.x
- aber längst nicht alles, was Python kann
- Erweiterung über “Python Enhancement Proposals” (PEPs)
<https://www.python.org/dev/peps>

Hilfe zu Python

- offizielle Homepage
<http://www.python.org>
- Einsteigerkurs „A Byte of Python“
<http://swaroopch.com/notes/Python> (englisch)
<http://abop-german.berlios.de> (deutsch)
- mit Programmiererfahrung „Dive into Python“
<http://diveintopython.net>

Wichtige PEPs

PEP 8: Style guide for Python Code

- “[...] code is read much more often than it is written”
- Einheitlichkeit zählt!
 - innerhalb eines Projekts, Moduls, Funktion
 - Einheitlichkeit nie strikt, wenn unsinnig darf gebrochen werden
- leere Zeilen zwischen logisch separierbaren Teilen, für Funktionsdefinitionen
- Zeilenlänge \approx 80 Zeichen
- Blöcke durch Indentierung gegeben (in C: {})
`>>>from __future__ import braces`
- Leerzeichen zur Lesbarkeit verwenden
- <https://www.youtube.com/watch?v=wf-BqAjZb8M>

Wichtige PEPs

PEP 20: The Zen of Python (>>>import this)

- siehe PEP 8 für Schöneheitsempfehlungen
- Vermeidung von Verschachtelungen
- Leerzeichen sind (meist) eine gute Idee
- Fehler niemals ignorieren oder durchgehen lassen
- Raten is schlecht, explizit ist besser als implizit
- immer einen offensichtlich richtigen Weg ein Problem zu lösen
- Problem sollten, wenn möglich immer sofort nach Erkennen behoben werden
- Code simpel halten
 - schwer zu erklärende Implementierung
 - wahrscheinlich eine schlechte Idee
- Namensräume, Namensräume, Namensräume!!!

Namen und Namensräume (in python)

- Namen der Objekte über einer Art Baumstruktur angeordnet und über entsprechenden Pfadnamen eindeutig angesprochen.
- Namensraum (i.e. Klasse): Sammlung von Namen
- Ein Namensraum bildet Namen auf den entsprechenden Objekten
- Unterschiedliche Namensräume können in python existieren, den gleichen Namen haben und komplett isoliert sein.
- In python sind Namensräume über einzelne Module definiert und sind hierarchisch.
- Interpreter erzeugt einen python Namensraum mit allen built-in Namen (weshalb Funktionen wie print() immer verfügbar sind).
- Module erzeugen einen globalen und Funktionen einen lokalen Namensraum.

Alles ist ein Objekt

Python ist eine objektorientierte Programmiersprache

- Objekte sind Pythons Datenabstrahierung
- alle Daten (inklusive Code!) is representiert durch Objekte

Jedes Objekt hat:

- Identität: unveränderlich, denke Adresse im Speicher (**is**, **id**)
- Typ: unveränderlich, bestimmt gültige Operationen für Objekt, mögliche Werte (**type** ← auch ein Objekt)
- Wert: teilweise veränderlich, vom Typ abhängig

Spezielle Methoden:

- *operator overloading, customization*
- definiert das Verhalten im Bezug auf möglichen Sprachoperationen
- Ganzzahldatentyp: 56 spezielle Methoden ??double-check numbers.Integral??

Dynamische Typen

- Objekttyp zur Laufzeit bestimmt (dynamically typed)
 - während Laufzeit möglich anderes Objekt an gleichen Namen zu binden
 - "Duck typing": Wenn es läuft wie eine Ente und quakt wie eine Ente, dann ist es eine Ente. (Objekt-Typ durch Methoden/Attribute und nicht durch seine Klasse beschrieben)
- es wird überprüft, ob eine adequate Methode existiert, die Operator entspricht (double-under (Doppelunterstrich), a.k.a. "dunder", Methoden oder magische Methoden) (sh. Überlastung von Operatoren).

```
>>>jahr=2019  
>>>type(jahr)
```

- Statische Typen: Variable muss vor Start definiert werden

Python starten

Aus der Shell:

```
> python
Python 2.6.5 (r265:79063, Apr 16 2010, 13:57:41)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more...
>>> print "Hello World"
Hello World
>>> help("print")
>>> exit()
```

- >>> markiert Eingaben
- **print**: Ausgabe auf das Terminal
- **help()**: interaktive Hilfe, wird mit "q" beendet
- statt **exit()** reicht auch Control-d
- oder **ipython** mit Tab-Ergänzung, History usw.

Python-Skripte

Als Python-Skript helloworld.py:

```
#!/usr/bin/python
```

```
# unsere erste Python-Anweisung
```

```
print "Hello World"
```

- mit `python helloworld.py` starten
- oder ausführbar machen (`chmod a+x helloworld.py`)
- **Umlaute vermeiden** oder Encoding-Cookie einfügen
- „#!“ funktioniert genauso wie beim Shell-Skript
- Zeilen, die mit „#“ starten, sind Kommentare

**Kommentare sind wichtig,
um ein Programm verständlich machen!**

- und nicht, um es zu verlängern!

Datentypen 1

- ganze Zahlen:

```
>>> print 42
```

```
42
```

```
>>> print -12345
```

```
-12345
```

- Fließkommazahlen:

```
>>> print 12345.000
```

```
12345.0
```

```
>>> print 6.023e23
```

```
6.023e+23
```

```
>>> print 13.8E-24
```

```
1.38e-23
```

- 1.38e-23 steht z. B. für 1.38×10^{-23}
- $12345 \neq 12345.0$ (z. B. bei der Ausgabe)

Datentypen 2

- Zeichenketten (Strings)

```
>>> print "Hello World"
Hello World
>>> print 'Hello World'
Hello World
>>> print """Hello
... World"""
Hello
World
```

- zwischen einfachen (') oder doppelten (") Anführungszeichen
- Über mehrere Zeilen mit dreifachen Anführungszeichen
- Leerzeichen sind normale Zeichen!
`"Hello World" ≠ "Hello World"`
- Zeichenketten sind keine Zahlen! `"1" ≠ 1`

Variablen

```
>>> number1 = 1
>>> number2 = number1 + 5
>>> print number1, number2
1 6
>>> number2 = "Keine Zahl"
>>> print number2
Keine Zahl
```

- Variablennamen bestehen aus Buchstaben, Ziffern oder „_“ (Unterstrich)
- am Anfang keine Ziffer
- Groß-/Kleinschreibung ist relevant: Hase \neq hase
- **Richtig:** i, some_value, SomeValue, v123, _hidden, _1
- **Falsch:** 1_value, some_value, some-value

Arithmetische Ausdrücke

+	Addition, bei Strings aneinanderfügen, z.B. $1 + 2 \rightarrow 3$, " a " + " b " \rightarrow " ab "
-	Subtraktion, z.B. $1 - 2 \rightarrow -1$
*	Multiplikation, Strings vervielfältigen, z.B. $2 * 3 = 6$, " ab " * 2 \rightarrow " abab "
/	Division, bei ganzen Zahlen ganzzahlig, z.B. $3 / 2 \rightarrow 1$, $-3 / 2 \rightarrow -2$, $3.0 / 2 \rightarrow 1.5$
%	Rest bei Division, z.B. $5 \% 2 \rightarrow 1$
**	Exponent, z.B. $3^{**}2 \rightarrow 9$, $.1^{**}3 \rightarrow 0.001$

- mathematische Präzedenz (Exponent vor Punkt vor Strich), z. B. $2^{**}3 * 3 + 5 \rightarrow 2^3 \cdot 3 + 5 = 29$
- Präzedenz kann durch runde Klammern geändert werden:
 $2^{**}(3 * (3 + 5)) \rightarrow 2^{3 \cdot 8} = 16,777,216$

Logische Ausdrücke

<code>==, !=</code>	Test auf (Un-)Gleichheit, z.B. <code>2 == 2</code> \rightarrow True, <code>1 == 1.0</code> \rightarrow True, <code>2 == 1</code> \rightarrow False
<code><, >, <=, >=</code>	Vergleich, z.B. <code>2 > 1</code> \rightarrow True, <code>1 <= -1</code> \rightarrow False
<code>or, and</code>	Logische Verknüpfungen „oder“ bzw. „und“
<code>not</code>	Logische Verneinung, z.B. <code>not False == True</code>

- Vergleiche liefern Wahrheitswerte: **True** oder **False**
- Wahrheitstabelle für die logische Verknüpfungen:

<i>a</i>	<i>b</i>	<i>a und b</i>	<i>a oder b</i>
True	True	True	True
False	True	False	True
True	False	False	True
False	False	False	False

- Präzedenz: Vergleiche vor logischen Verknüpfungen
- Beispiele:
 - `3 > 2 and 5 < 7` \rightarrow True,
 - `3 > (2 and 5) < 7` \rightarrow False,
 - `1 < 1 or 2 >= 3` \rightarrow False

if: bedingte Ausführung

```
>>> a = 1
>>> if a < 5:
...     print "a ist kleiner als 5"
... elif a > 5:
...     print "a ist groesser als 5"
... else:
...     print "a ist 5"
a ist kleiner als 5
>>> if a < 5 and a > 5:
...     print "Das kann nie passieren"
```

- **if-elif-else** führt den **Block** nach der ersten erfüllten Bedingung (logischer Wert True) aus
- Trifft keine Bedingung zu, wird der **else**-Block ausgeführt
- **elif** oder **else** sind optional

Blöcke und Einrückung

```
>>> a=5
>>> if a < 5:
...     # wenn a kleiner als 5 ist
...     b = -1
... else: b = 1
>>> # aber hier geht es immer weiter
... print b
1
```

- Alle *gleich eingerückten* Befehle gehören zum Block
- Nach dem **if**-Befehl geht es auf Einrückungsebene des **if** weiter, egal welcher **if**-Block ausgeführt wurde
- Einzeilige Blöcke können auch direkt hinter den Doppelpunkt
- Einrücken durch Leerzeichen oder Tabulatoren (einfacher)

Blöcke und Einrückung 2

- ein Block kann nicht leer sein, aber der Befehl **pass** tut nichts:

```
if a < 5:  
    pass  
else:  
    print "a ist groesser gleich 5"
```

- **IndentationError** bei ungleichmäßiger Einrückung:

```
>>> print "Hallo"  
Hallo  
>>>     print "Hallo"  
File "<stdin>", line 1  
    print "Hallo"  
    ^
```

IndentationError: unexpected indent

- Falsche Einrückung führt im allgemeinen zu Programmfehlern!

while: Schleifen

```
>>> a = 1
>>> while a < 5:
...     a = a + 1
>>> print a
5
```

- Führt den Block solange aus, wie die Bedingung wahr ist
 - kann auch nicht ausgeführt werden:
-

```
>>> a = 6
>>> while a < 5:
...     a = a + 1
...     print "erhoehe a um eins"
>>> print a
6
```

for: Sequenz-Schleifen

```
>>> for a in range(1, 3): print a
1
2
>>> b = 0
>>> for a in range(1, 100):
...     b = b + a
>>> print b
4950
>>> print 100 * (100 - 1) / 2
4950
```

- **for** führt einen Block für jedes Element einer **Sequenz** aus
- Das aktuelle Element steht in **a**
- **range**(k, l) ist eine Liste der Zahlen a mit $k \leq a < l$
- später lernen wir, Listen zu erstellen und verändern

break und continue: Schleifen beenden

```
>>> for a in range(1, 10):  
...     if a == 2 or a == 4 or a == 6: continue  
...     elif a == 5: break  
...     print a  
1  
3  
>>> a = 1  
>>> while True:  
...     a = a + 1  
...     if a > 5: break  
>>> print a  
6
```

- beide überspringen den Rest des Schleifenkörpers
- **break** bricht die Schleife ganz ab
- **continue** springt zum Anfang