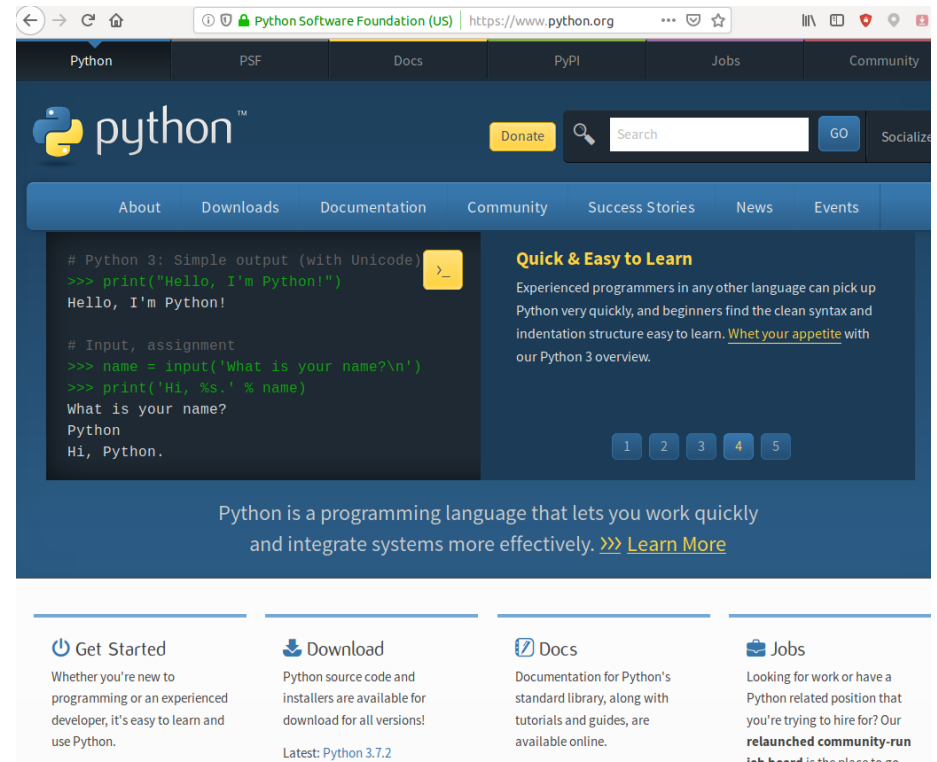


Python 3

Grundlagen der
Programmierung



- Die in diesem Seminar verwendete Werkzeuge und Frameworks sind Open Source
 - LPGL Lizenzmodell
- Dies ist ein Programmier-Seminar
 - Damit werden die Inhalte durch Übungen vertieft und verinnerlicht
 - Musterbeispiele werden zur Verfügung gestellt
 - GitHub
- Dokumentation und Ressourcen stehen auch im Internet zur Verfügung

Copyright und Impressum

© Integrata AG

Integrata AG
Zettachring 4
70567 Stuttgart

Alle Rechte, einschließlich derjenigen des auszugsweisen Abdrucks, der fotomechanischen und elektronischen Wiedergabe vorbehalten.

Einführung	6
Aufsetzen der Umgebung	12
Grundlagen der Programmierung	29
Funktionen	55
Laufzeitmodell	70
Komplexe Datenverarbeitung	82
Module	93
Objektorientierung	108
Weitere Konzepte	127
Die Standardbibliothek	143

1

EINFÜHRUNG

1.1

PYTHON - EINE SKRIPTSPRACHE

- Python ist
 - eine Skriptsprache
 - Objekt-orientiert
 - Funktional
 - Dynamisch typisiert
 - Modular

- Kein aufwändiger Build-Prozess erforderlich
 - Die Skripte sind direkt ausführbar
- Agilität Software-Entwicklung
 - Änderungen des Skript-Codes werden von der Laufzeitumgebung sofort übernommen
- Die Struktur der Programme kann einfach gehalten werden
 - Anweisungen können ohne jeglichen Rahmen "einfach so" geschrieben werden
- Ausführung in einer REPL (Read-eval-print loop)

1.2

EIN ERSTES PROGRAMM

- Das "klassische" erste Programm in Python

```
print("Hello World!")
```

- Es ist tatsächlich so einfach!

2

AUFSETZEN DER UMGEBUNG

2.1

BESTANDTEILE

- Entwickler-Werkzeuge
 - Python Installation
 - Editoren zur komfortablen Erstellung von Programmen
 - Debugger
 - Python Runtime
- Weitere Hilfsmittel
 - Dokumentation
 - Source Code Management

2.2

PYTHON-INSTALLATION

- Python ist frei erhältlich
- Distributionen für alle relevanten Betriebssysteme verfügbar
 - Linux
 - Windows
 - Mac
 - ...
- In der Distribution ist auch eine Dokumentation enthalten
 - Selbstverständlich auch Online verfügbar!
- Weiterhin die Python-Shell
 - Eine interaktive REPL

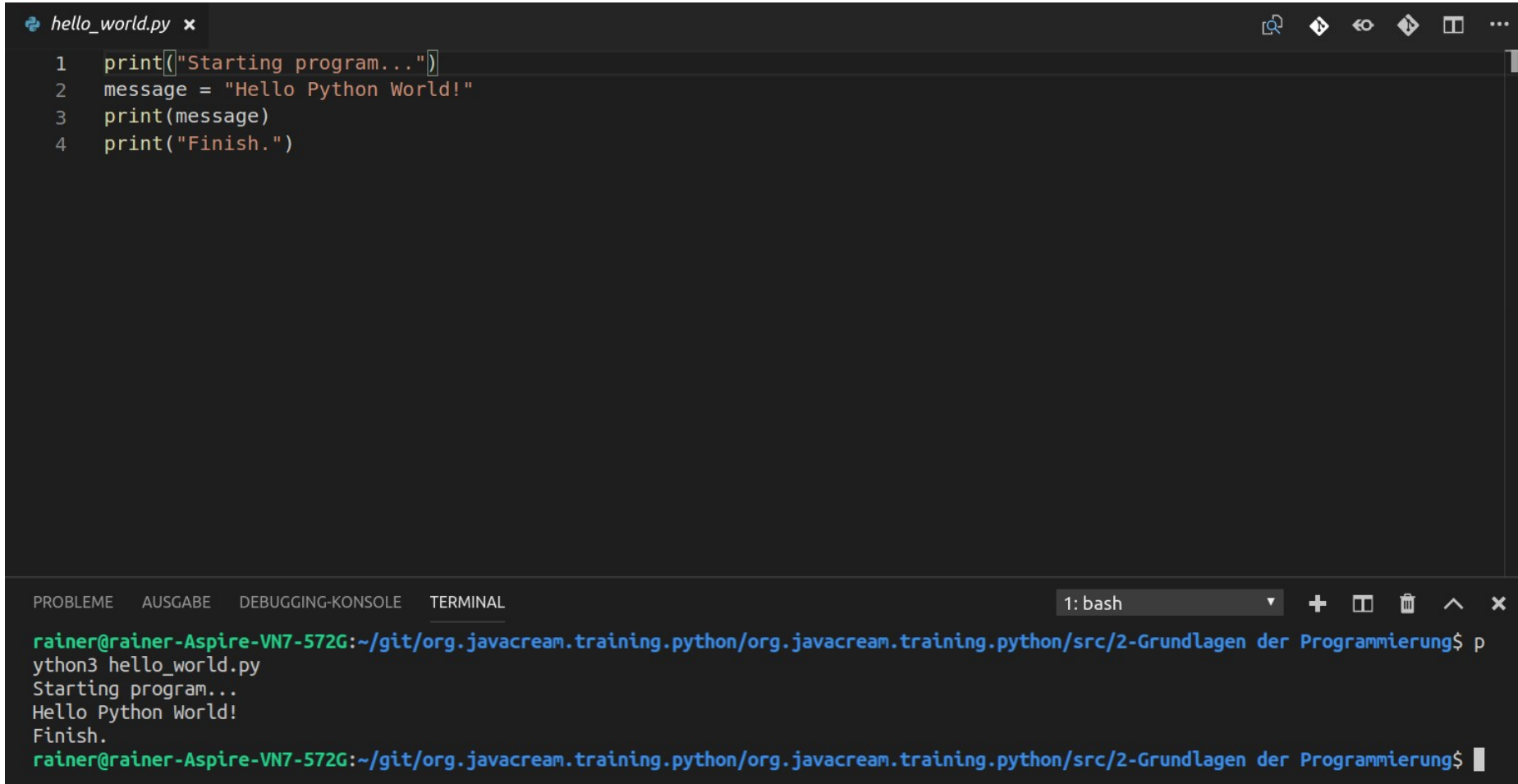

```
Python 3.6.7 (default, Oct 22 2018, 11:32:17)
[GCC 8.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> message = "Hello World!"
>>> print(message)
Hello World!
>>> 
```

2.3

EDITOREN UND ENTWICKLUNGSUMGEBUNG

- Selbst einfache Editoren bieten für Python PlugIns mit elementarer Unterstützung für Syntax-Hervorhebungen etc. an
 - Notepad++
 - UltraEdit
 - ...
- Mächtige Editoren bieten eine noch bessere Unterstützung an
 - Beispiele
 - Atom
 - Visual Studio Code
 - Funktionen
 - Syntax-Prüfungen
 - Integration der Python Shell
 - Damit ist die Abgrenzung zu vollständigen Entwicklungsumgebungen fließend

Beispiel: Python-Skript in Visual Studio Code



The image shows a Visual Studio Code editor window with a file named `hello_world.py`. The code in the editor is as follows:

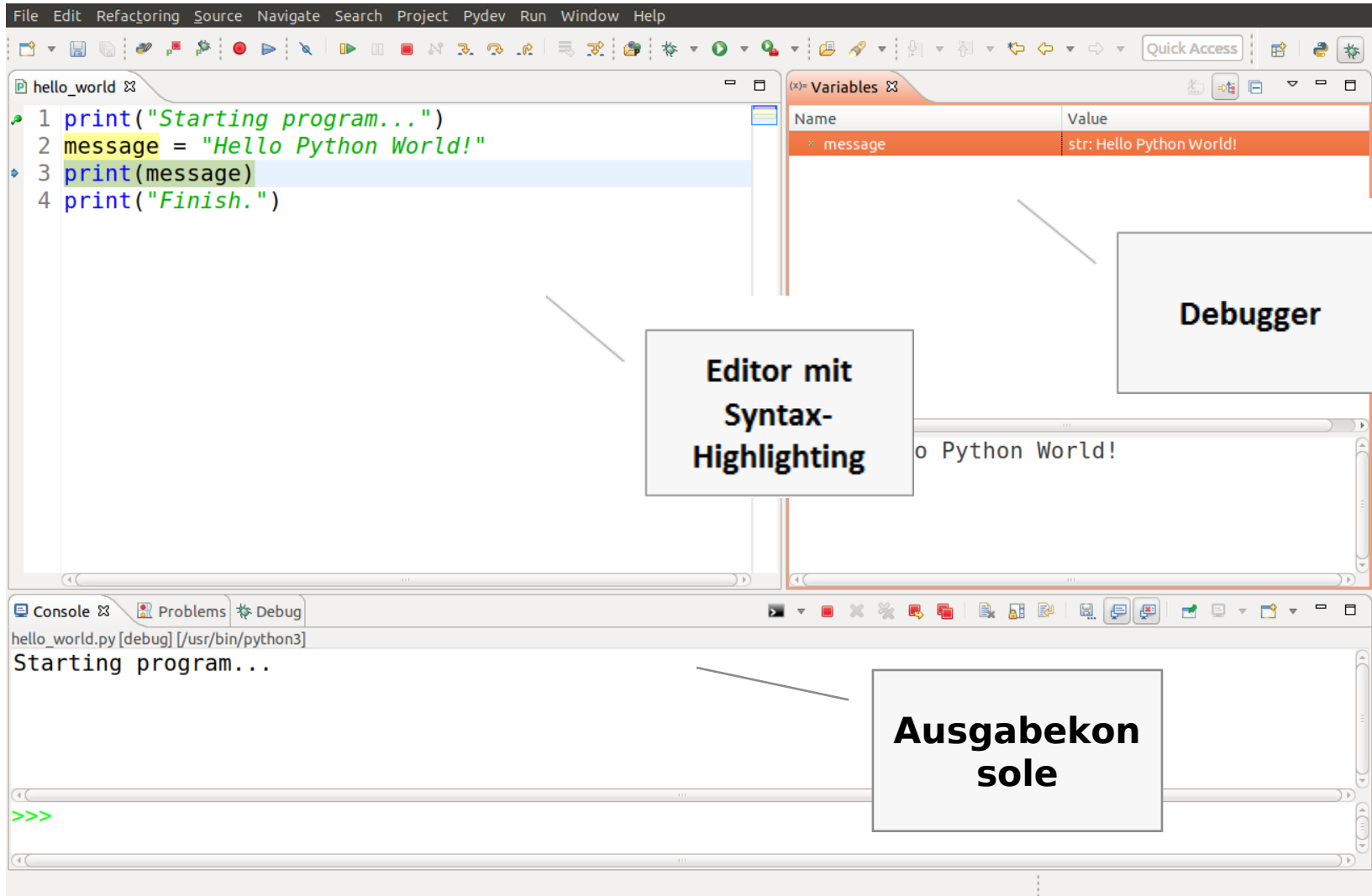
```
1 print("Starting program...")
2 message = "Hello Python World!"
3 print(message)
4 print("Finish.")
```

Below the editor, the **TERMINAL** panel is active, showing the command `python3 hello_world.py` being executed. The output of the script is displayed in the terminal:

```
rainer@rainer-Aspire-VN7-572G:~/git/org.javacream.training.python/org.javacream.training.python/src/2-Grundlagen der Programmierung$ python3 hello_world.py
Starting program...
Hello Python World!
Finish.
rainer@rainer-Aspire-VN7-572G:~/git/org.javacream.training.python/org.javacream.training.python/src/2-Grundlagen der Programmierung$
```

- Im Gegensatz zu den einfachen Editoren bieten Entwicklungsumgebungen noch mehr Werkzeuge an
 - Code Assist
 - Integrierte Hilfefunktion
 - Integrierte Dokumentation
 - Debugger
 - Integrierte Testwerkzeuge
 - Unit-Testing
 - Prüfen der Code-Qualität und der Einhaltung von Richtlinien
 - "Linter"
 - Code-Formatting
 - Anbindung an Versionsverwaltung

Beispiel: Eclipse und PyDev



The screenshot displays the Eclipse IDE interface with the PyDev plugin. The main editor window shows a Python script named `hello_world.py` with the following code:

```
1 print("Starting program...")
2 message = "Hello Python World!"
3 print(message)
4 print("Finish.")
```

The code is syntax-highlighted. A callout box labeled **Editor mit Syntax-Highlighting** points to the editor area.

The **Variables** view on the right shows the current state of the program's variables:

Name	Value
message	str: Hello Python World!

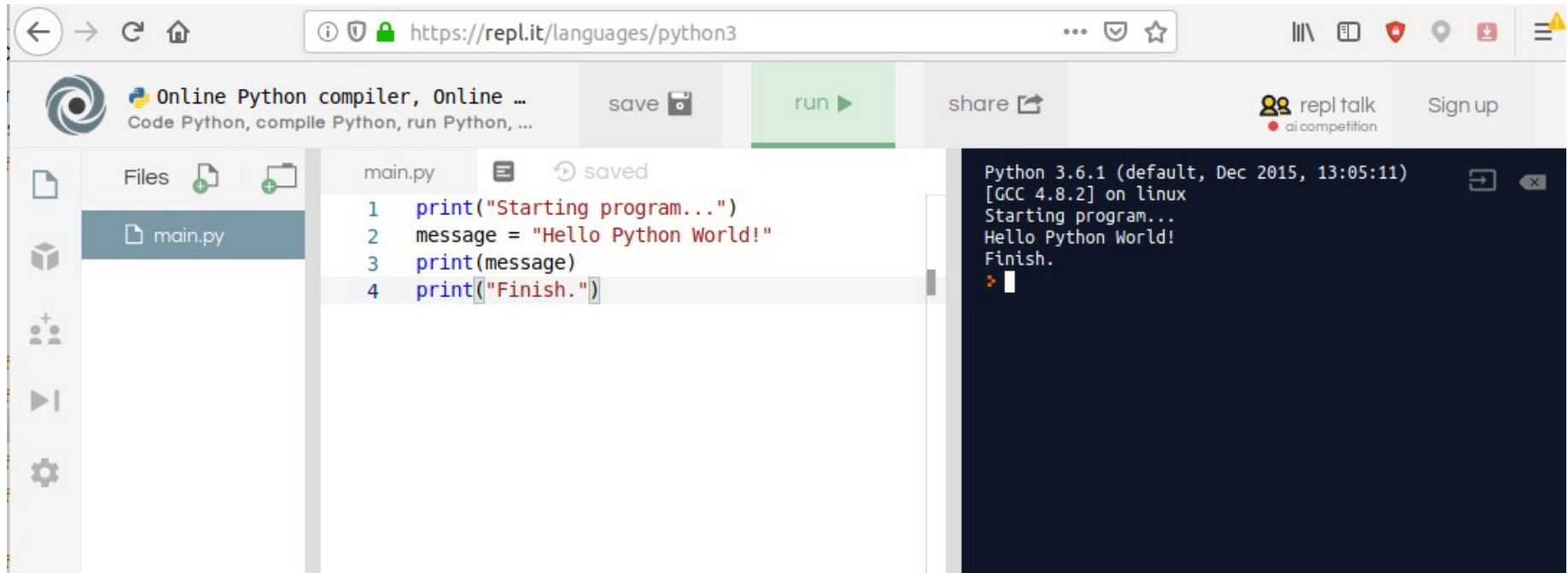
A callout box labeled **Debugger** points to the Variables view.

The **Console** view at the bottom shows the output of the program:

```
hello_world.py [debug] [/usr/bin/python3]
Starting program...
>>>
```





A callout box labeled **Ausgabekonsole** points to the console output.

Beispiel: Repl.It - Eine Online-IDE




2.4

DOKUMENTATION



Python Software Foundation (US) | <https://docs.python.org/3/>

 Python »

English ▼

3.7.2 ▼

Documentation »

Download

Download these documents

Docs by version

- [Python 3.8 \(in development\)](#)
- [Python 3.7 \(stable\)](#)
- [Python 3.6 \(security-fixes\)](#)
- [Python 3.5 \(security-fixes\)](#)
- [Python 2.7 \(stable\)](#)
- [All versions](#)

Other resources

- [PEP Index](#)
- [Beginner's Guide](#)
- [Book List](#)
- [Audio/Visual Talks](#)

Python 3.7.2 documentation

Welcome! This is the documentation for Python 3.7.2.

Parts of the documentation:

What's new in Python 3.7?

or all "What's new" documents since 2.0

Tutorial

start here

Library Reference

keep this under your pillow

Language Reference

describes syntax and language elements

Python Setup and Usage

how to use Python on different platforms

Python HOWTOs

in-depth documents on specific topics

Installing Python Modules

installing from the Python Package Index & other sources

Distributing Python Modules

publishing modules for installation by others

Extending and Embedding

tutorial for C/C++ programmers

Python/C API

reference for C/C++ programmers

FAQs

frequently asked questions (with answers!)

2.1.0219 © Integrata AG

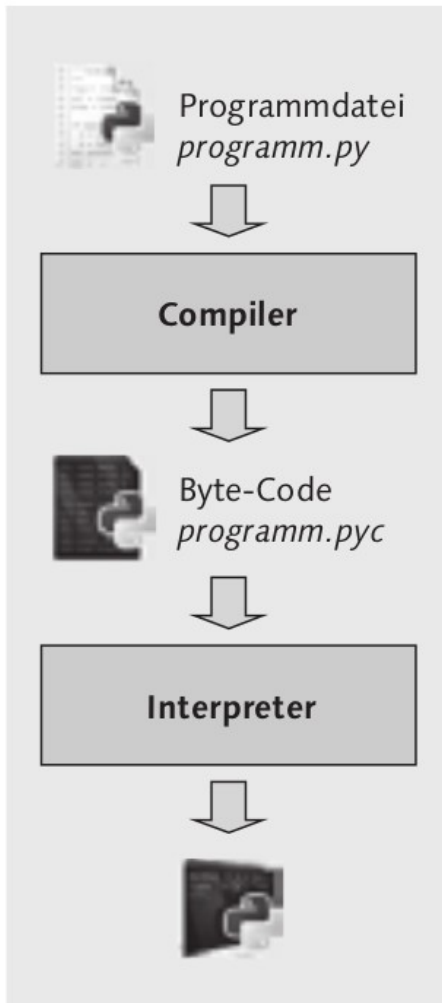
Python 3

25

2.5

DIE PYTHON VIRTUAL MACHINE

- Der "Python-Interpreter" führt das Skript nicht direkt aus
- Der Python Compiler erzeugt aus dem Skript eine ausführbare Binärdatei
 - Damit ist Python eigentlich gar keine Skript-Sprache
 - Allerdings erfolgt diese Compilation automatisch und ist damit für die Programm-Entwicklung transparent
- Damit ist der "Interpreter" eine so genannte "Virtual Machine", die "Bytecode" ausführt



Aus
Python 3
Das umfassende Handbuch
von Johannes Ernesti, Peter Kaiser

3

GRUNDLAGEN DER PROGRAMMIERUNG

3.1

ÜBERSICHT

Variablen

```
even_message = "an even number: "  
odd_message = "an odd number: "  
numbers = range(1, 10)  
finished = False
```

Schleife

```
for i in numbers:  
    print ("processing number " , i, ", finished: ", finished)  
    if i % 2 == 0:  
        print (even_message, i)  
    else:  
        print (odd_message, i)
```

Abfrage

```
finished = True  
print ("all numbers processed, finished: ", finished)
```

Vergleichsoperator

3.2

TYPEN UND OPERATIONEN

Python Tutorial

- [Python HOME](#)
- [Python Intro](#)
- [Python Get Started](#)
- [Python Syntax](#)
- [Python Variables](#)**
- [Python Numbers](#)
- [Python Casting](#)
- [Python Strings](#)
- [Python Operators](#)
- [Python Lists](#)
- [Python Tuples](#)
- [Python Sets](#)
- [Python Dictionaries](#)
- [Python If...Else](#)
- [Python While Loops](#)
- [Python For Loops](#)
- [Python Functions](#)
- [Python Lambda](#)
- [Python Arrays](#)
- [Python Classes/Objects](#)
- [Python Iterators](#)
- [Python Modules](#)
- [Python Dates](#)

Python Variables

[< Previous](#)[Next >](#)

Creating Variables

Unlike other programming languages, Python has no command for declaring a variable.

A variable is created the moment you first assign a value to it.

Example

```
x = 5
y = "John"
print(x)
print(y)
```

[Run example >](#)

Variables do not need to be declared with any particular type and can even change type after they have been set.

- Alles was mit einer Zahl beginnt, ist eine Zahl
- Die Genauigkeit ergibt sich aus dem Literal
 - Fließkommazahl
 - Mit Komma
 - 4.2
 - Integer
 - Ohne Komma
 - 42

- Grundrechenarten

- +
- -
- *
- /

- Modulo und Potenzieren

- %
- **

- Reihenfolge

- "Punkt vor Strich"
- Die Reihenfolge kann durch runde Klammern definiert werden

- Direkte Unterstützung für komplexe Zahlen
 - Berechnung: $(5+3j) * 3$
 - Ergebnis: $(15+9j)$
- Unterstützung großer Zahlen (größer als der Zahlenbereich der CPU):
 - Berechnung: `1234567890987654321**42`
 - Ergebnis:

```
6976362768886603014932234617124898018339922481538698336754469169985189987
1523650874966501895002272033642494060339623454001302863185183843381974796
7099124063806679730000275944197718696863024232437467947893122267950788000
0745677744557515984441935106158912446209654891373984194215683501604736921
6634733993476654189604398954546909373686315783876319241485612864337055724
7861142879386276177100550123949353889253616804532414873048495240940497066
6494706835604248858817625266775120835325140116316458219952192308551489845
4882817249981612435321713387973878080894567582544277473612124864533670237
1488823259609808023431112612341980870199271153431097312917943503305449752
0237023899815346434343392975480141699463397523258748875918223820375395515
250764534847318173082024847841
```

- Markierung durch einfache oder doppelte Anführungsstriche
- Die \ in Zeichenketten werden für spezielle Zeichen verwendet
 - z. B. \n für "Neue Zeile"
- Längere Stringkonstanten lassen sich über Tripelquotes erzeugen

```
""" Dies ist ein  
mehrzeiliger  
String"""
```

- Werden zwei Zeichenketten addiert, werden sie konkateniert
`'Spam' + 'Spam'`
`'SpamSpam'`
- Zeichenketten können "multipliziert" werden (nur mit Zahlen):
`'Spam' * 5`
`'SpamSpamSpamSpamSpam'`
- Zeichenketten können über Indizes und Bereiche angesprochen werden:
`'SpamSpamSpam'[0]`
`'S'`
`'SpamSpamSpam'[5:]`
`'pamSpam'`
`'SpamSpamSpam'[2:10]`
`'amSpamSp'`

- Die Funktion *len* bestimmt die Länge von Strings

```
len("Hallo")
```

5

```
len("SpamSpam")
```

8

```
len("Spam"*5)
```

20

- Listen sind geordnete Sammlungen beliebiger Inhalte
- Listen werden durch `[]` markiert
 - `["Spam", "Ham", 5, 3.0, "Hallo"]`
- Die Funktion `len` funktioniert auch mit Listen
- Die Slices und Indices funktionieren genauso wie bei Strings
- Die Funktion `range` liefert eine Liste:

```
range(4)
```

```
[0, 1, 2, 3]
```

```
range(3, 8)
```

```
[3, 4, 5, 6, 7]
```

```
range(3, 8, 2)
```

```
[3, 5, 7]
```


- In anderen Sprachen auch Hashes oder Hashtabellen genannt
- Markiert durch {}
- Ungeordnete assoziative Listen, "Name/Werte Paare"
- Referenzieren über Schlüssel, nicht Indizes

```
film = {'Director': 'Chapman', 'Actor': 'Gilliam',  
        'Producer': 'Cleese', 'Writer': 'Palin'}
```

```
film['Actor']
```

```
'Gilliam'
```

- Referenzieren genau wie bei Listen über []

- Tupel sind auch Sequenzen
- Werden mit () markiert
 - `tupel = (1, "Cleese", "John")`
- Referenzieren wieder, wie bei Listen, über Indizes in []
 - `tupel[2]`
 - `John`
- Tupel sind unveränderlich
 - d. h. Zuweisungen zu einem Tupel sind illegal
 - `t = (1, 2, 3)`
 - `t[0] = 1` # verboten

- Listen, Tupel und Dictionaries können alle anderen Typen beinhalten
 - damit sind komplexe Datenstrukturen möglich
 - Auch als "Objekt-Graph" bezeichnet

3.3

ANWEISUNGEN

- Mit dem = wird einer Variablen ein Wert zugewiesen und diese gleichzeitig definiert, sofern die Variable nicht existiert

```
print a
```

```
    NameError: name 'a' is not defined
```

```
a="Hallo"
```

```
print a
```

```
    Hallo
```

- Variablen haben keinen festen Typ
 - dieser wird zur Laufzeit bestimmt
- Variablennamen dürfen aus Buchstaben, Zahlen und dem Unterstrich bestehen
 - dürfen jedoch nicht mit einer Zahl beginnen
 - sind CaseSensitive
- Eine Variable kann zu einem Zeitpunkt ein String, später eine Liste sein

```
a="Ich bin eine Zeichenkette"
print a
    Ich bin eine Zeichenkette
a=["Ich", "bin", "eine", "Liste"]
print len(a)
    4
```

- Typfeststellung eines Wertes/Variablen über type()

```
print type("Hallo Welt")
    <class 'str'>
```

- Mit Hilfe der Tupel können mehrere Variablen gleichzeitig belegt werden

```
(a,b) = (5, "Ni")
```

- Damit ist auch eine einfache Vertauschung von Variablen möglich:

```
(a,b) = (b, a)
```

- Soll mehreren Variablen der selbe Wert zugewiesen werden, kann die Mehrfachzuweisung verwendet werden

```
a=b="Hello"
```

- builtins sind Anweisungen, die jedem Python-Programm direkt zur Verfügung gestellt werden
- Beispiele:
 - `abs()`
 - Absolutwert einer Zahl
 - `char(int)`
 - liefert das Zeichen zu dem angegebenen ASCII-Code
 - `len(value)`
 - Länge eines Wertes
 - `exit()`
 - beendet Python
 - `hex(Zahl)`
 - liefert die Hexadezimale Repräsentation der Zahl
 - `min, max`
 - liefert das Minimum oder Maximum einer Liste
 - `range`
 - liefert eine Liste von Zahlen im gewünschten Wertebereich

- Eingebaute Typen `int()`, `float()`, `str()`, `bool()` können für Umwandlungen zu einem der eingebauten Typen verwendet werden

```
float(4)
```

```
4.0
```

```
int(float(4))
```

```
4
```

```
int("42")
```

```
42
```

```
str(int("42"))
```

```
'42'
```

- Vergleich mit
 - `==`
 - Gleichheit der Werte/Inhalte
 - `!=`
 - `<`
 - `<=`
 - `>`
 - `>=`
 - `is`
 - Gleichheit Referenzen
- Ergebnis der Operatoren ist `True` oder `False`

- Ablaufsteuerung im `if-else-elif`

```
if a==3:  
    print "a ist 3"  
elif a==4:  
    print "a ist 4"  
else:  
    print "weder 3 noch 4"
```

- `elif` und `else` sind optional
- Die Einrückung bestimmt den Block
 - Klammern sind nicht nötig

- Zwei Arten von Schleifen:

- for
- while

- for

```
for i in range(1,10)  
    print(i)
```

- while

```
counter = 0  
while (counter < 3)  
    print (counter)  
    counter = counter + 1
```

- `break`
 - beendet die aktuelle Schleife
- `continue`
 - bricht den aktuellen Schleifendurchlauf ab, macht mit dem nächsten weiter

4

FUNKTIONEN

4.1

ÜBERSICHT

- Kleinste "Einheit" von wiederverwendbaren Code
- Definition mit `def`:
- Funktionsaufrufe mit `()`

```
def out():  
    print "called out"  
out()
```



```
def calculate_price(original , discount, shipping):  
    discounted_price = original * (1 - discount/100)  
    price = discounted_price + shipping  
    return price  
  
def calculate_shipping (provider):  
    if provider == "EIM":  
        shipping = 5.99  
    elif provider == "CGK":  
        shipping = 3.99  
    else:  
        shipping = 9.99  
    return shipping  
  
def order():  
    provider = "EIM"  
    shipping = calculate_shipping(provider)  
    original_price = 19.99  
    price = calculate_price(original_price, 10, shipping)  
    print("total price: ", price)
```

**Name der
Funktion**

Parameter

Rückgabe

- Die Parameter der Funktion werden innerhalb der runden Klammern angegeben

```
def f(p):  
    print "param: ",p  
f("Spam")  
Der Wert von A:  Spam
```

- Die Anzahl der übergebenen Parameter muss mit der Parameterliste übereinstimmen

- Bei der Funktionsdefinition können Parameter mit Default-Werten versehen werden. Beim Aufruf müssen für diese dann keine Werte angegeben werden

```
def f(a=5,b=2):  
    print a*b  
f()  
10
```

- Funktionsparameter die bei Definition mit einem * versehen werden, werden als beliebig langes Tupel übergeben

```
def f(*a):  
    print a  
f(1,2,3,4)  
    (1, 2, 3, 4)  
f("bla","bla")  
    ('bla', 'bla')
```

- Parameter die bei Definition mit ** markiert sind, werden im Endeffekt als Dictionary übergeben

```
def f(**a):  
    print (a)
```

```
f(number=42, name="Hugo")  
{ 'number': 2, 'name': 'Hugo' }
```

- Funktionen können Werte zurückliefern
 - Schlüsselwort `return`
- Dieser Rückgabewert kann wiederum in Zuweisungen oder weiteren Funktionsaufrufen verwendet werden

```
def mult(a,b):  
    return a*b  
  
print mult(4,2)  
  
8
```

- In Funktionen definierte Variablen sind lokal, d.h. nur innerhalb der Funktion gültig
- Soll auf Variablen außerhalb der Funktion (schreibend) zugegriffen werden, so müssen diese als `global` markiert werden
- Eine globale Variable, zu der innerhalb einer Funktion ein neuer Wert zugewiesen wird, ist nicht mehr sichtbar
 - Sie wird damit effektiv zu einer lokalen Variablen
- Python3 führt zusätzlich noch `nonlocal` Variablen ein
 - Benutzung in verschachtelten Funktionen

- Komplexere Datentypen (Liste, Tupel, String und Dictionary) werden per Referenz übergeben
- Änderungen an den Parametern wirken sich auf den Aufrufer aus

```
def changeMe(lst):  
    lst[0]='Hallo'  
  
l=range(4)  
change(l)  
print l  
    ['Hallo',1,2,3]
```


4.2

LAMBDAS

- Funktionen können Variablen zugewiesen werden

```
def f: .....  
a=f  
a("Hallo!")
```

- Funktionen können somit auch benutzt werden
 - als Parameter
 - als Rückgabewerte

- Das Schlüsselwort `def` kann nicht an beliebigen Stellen stehen
- `lambda` ist ein "Funktions-Literal"

```
fkt=lambda x:x**2  
fkt(2)  
4
```

- Das Ergebnis der letzten Anweisung ist der implizite Rückgabewert
- Lambda-Ausdrücke können auch innerhalb von Anweisungen stehen

```
map(lambda x:x**x, range(10))
```



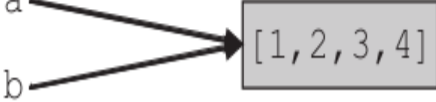
```
if reverse:
    mycmp = lambda x,y: y - x
else:
    mycmp = lambda x,y: x - y
```

5

LAUFZEITMODELL

5.1

ÜBERSICHT

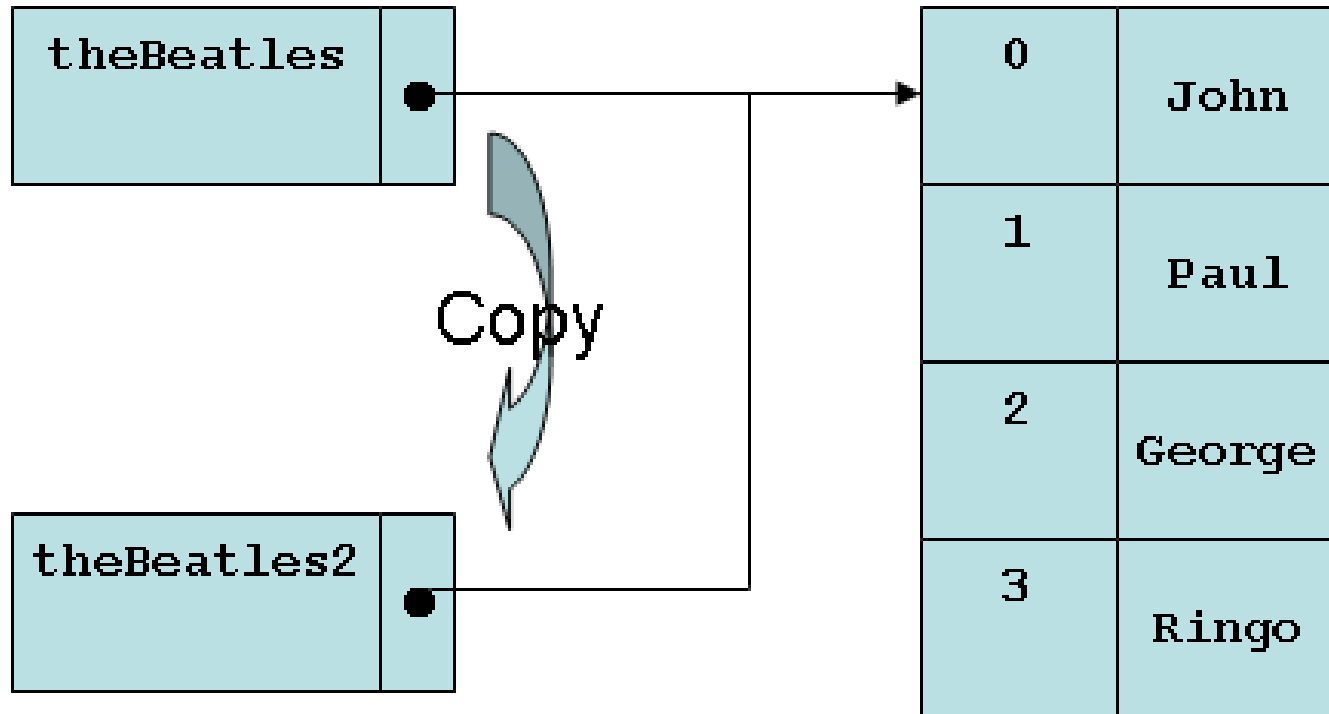
Code	Interner Vorgang
<code>a = [1,2]</code>	
<code>b = a</code>	
<code>a += [3,4]</code>	

aus "Python 3" von Johannes Ernesti und
Peter Kaiser

5.2

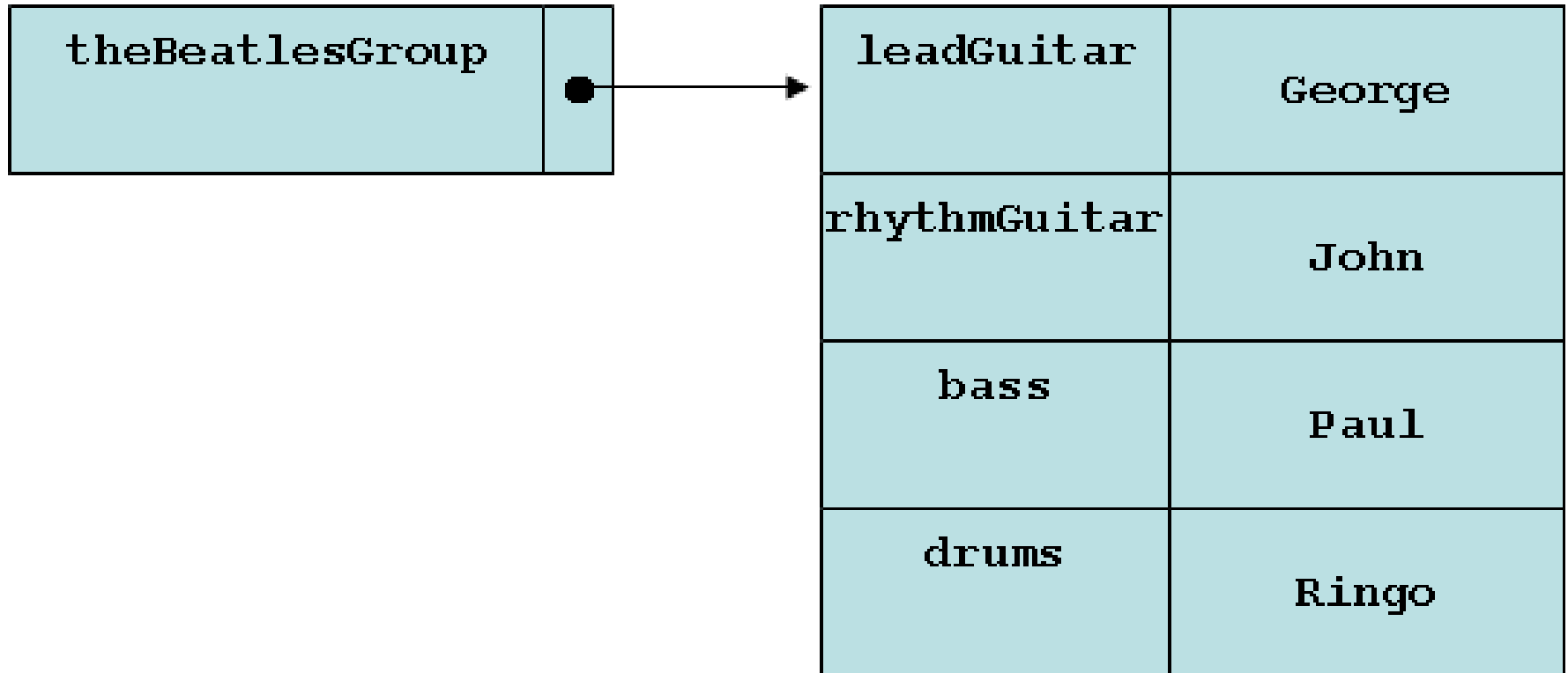
REFERENZEN

- Eine grafische Darstellung arbeitet am Besten mit einem Pfeilsymbol, den sogenannten Referenzen:
 - Technisch gesehen ist eine Referenz eine interne Speicheradresse
- Auch hier wird bei der Zuweisung bzw. bei einer Parameterübergabe an eine Funktion ein Wert kopiert
 - Es ist hier allerdings der Wert der Referenz!
- Wird somit ein Array einer Funktion als Parameter übergeben, so schlagen Änderungen, die innerhalb der aufgerufenen Funktion durchgeführt werden, sehr wohl auf die Variablen der aufrufenden Funktion durch
 - Ein fundamentaler und sehr wichtiger Unterschied im Vergleich zu den bisher benutzten Variablen!



- Auch Dictionary können durch Referenzen dargestellt werden:

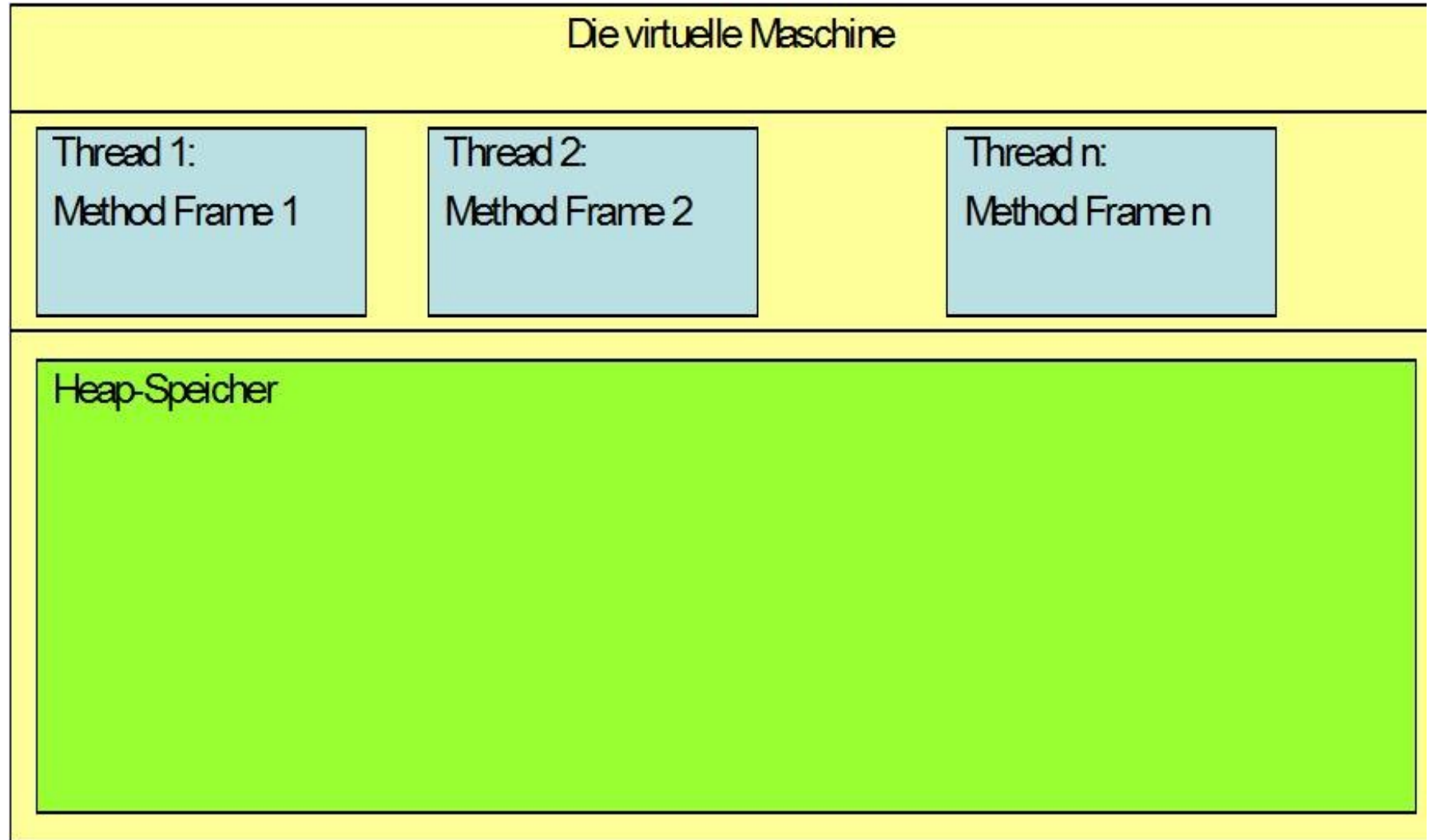
```
theBeatlesGroup = {"leadGuitar": "George",  
"rhythmGuitar": "John", "bass": "Paul", "drums":  
"Ringo"}
```



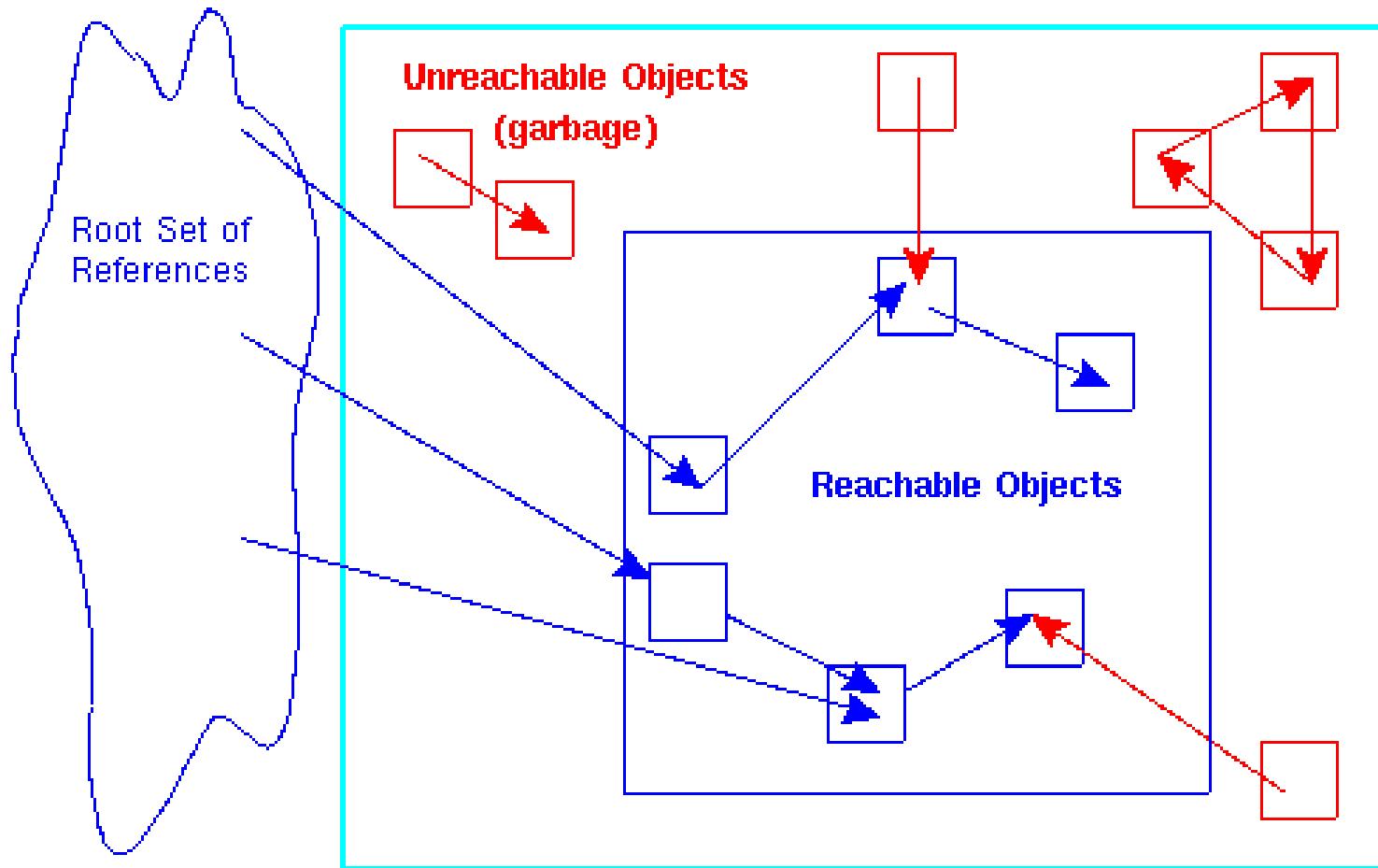
5.3

MEMORY MODEL

- Die Python Virtual Machine organisiert die Datenhaltung in zwei Bereiche
 - Im Stack werden die Variablen des Programms gehalten
 - Jeder nebenläufige Thread bekommt seinen eigenen Bereich
 - Der Heap enthält die Datenstrukturen
- Ein Garbage Collector versucht, den Heap-Bereich automatisch zu bereinigen und zu defragmentieren
 - Python-Programme müssen deshalb unbenutzte Variablen und Referenzen nicht aufräumen



Prinzipielle Arbeitsweise der Garbage Collection



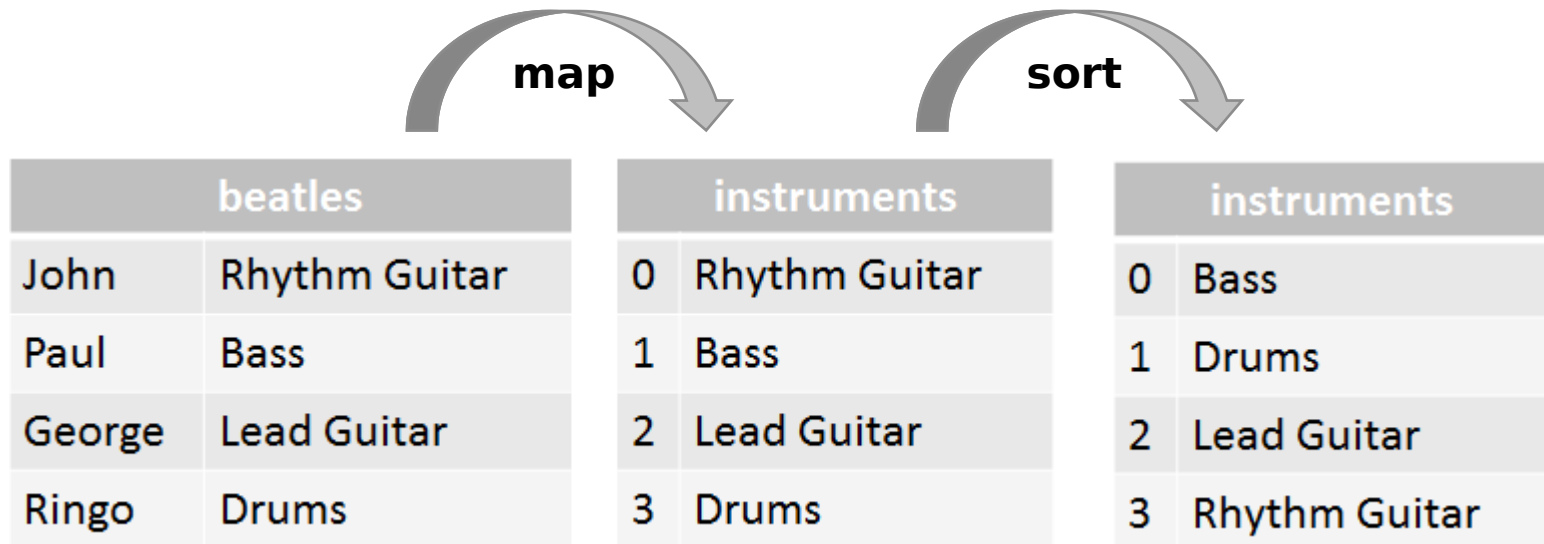
6

KOMPLEXE DATENVERARBEITUNG

6.1

ÜBERSICHT

```
beatles = {"John": "Rhythm Guitar", "Paul": "Bass",  
           "George": "Lead Guitar", "Ringo": "Drums"}  
instruments = list(map(lambda name: beatles.get(name), beatles))  
instruments.sort(reverse=False)  
print(instruments)
```



6.2

COLLECTIONS

- Python unterstützt die üblichen Datencontainer moderner Programmiersprachen
 - Sets
 - Eine ungeordnete Menge
 - Enthält keine Duplikate
 - Listen
 - Reihenfolge wird berücksichtigt
 - Zugriff über Index
 - Dictionaries
 - Key-Value-Paare ohne Reihenfolge
 - Zugriff über den Key
- Darüber hinaus werden benutzt
 - Tuples
 - Enumerations
- Diese Datencontainer werden allgemein auch als Collections bezeichnet

- Die Collections sind in Python elementare Datentypen und deshalb in ihrer Benutzung bereits aus vorherigen Abschnitten bekannt

6.3

STREAMING

- Die klassische Datenverarbeitung von Collections besteht aus
 - Schleifen
 - Abfragen
 - Temporären Collections mit Zwischen-Ergebnissen
- Diese Form der Implementierung ist
 - aufwändig mit vielen Code-Zeilen
 - Speicherintensiv
 - schlecht verallgemeinerbar
 - Auch eine Datei oder eine Datenbankabfrage ist prinzipiell eine Collection!
 - schlecht parallelisierbar
 - Eine Zwischen-Collection muss erst erstellt werden, bevor der nächste Arbeitsschritt erfolgen kann
 - skaliert schlecht
 - Eine "unendlich große" Collection kann nicht verarbeitet werden

Was ist "Streaming" bei Collections?

- Beim Streaming wird die Logik den Collections zugeordnet
 - Iteration
 - Filter
 - Sortierung
 - Transformation
- Die spezifische Logik wird durch Callback-Funktionen übergeben
 - Häufig als Lambda-Ausdrücke
- Die einzelnen Schritte können einfach verknüpft werden
 - `list.filter(...).sort(...)`

- (Noch) nicht konsequent umgesetzt
- Statt dessen eine Mischung aus builtin-Funktionen und Operationen der elementaren Collections-Typen

```
beatles = {"John": "Rhythm Guitar", "Paul": "Bass",  
          "George": "Lead Guitar", "Ringo": "Drums"}  
instruments = list(map(lambda name: beatles.get(name),  
                        beatles))  
instruments.sort(reverse=False)  
print(instruments)
```

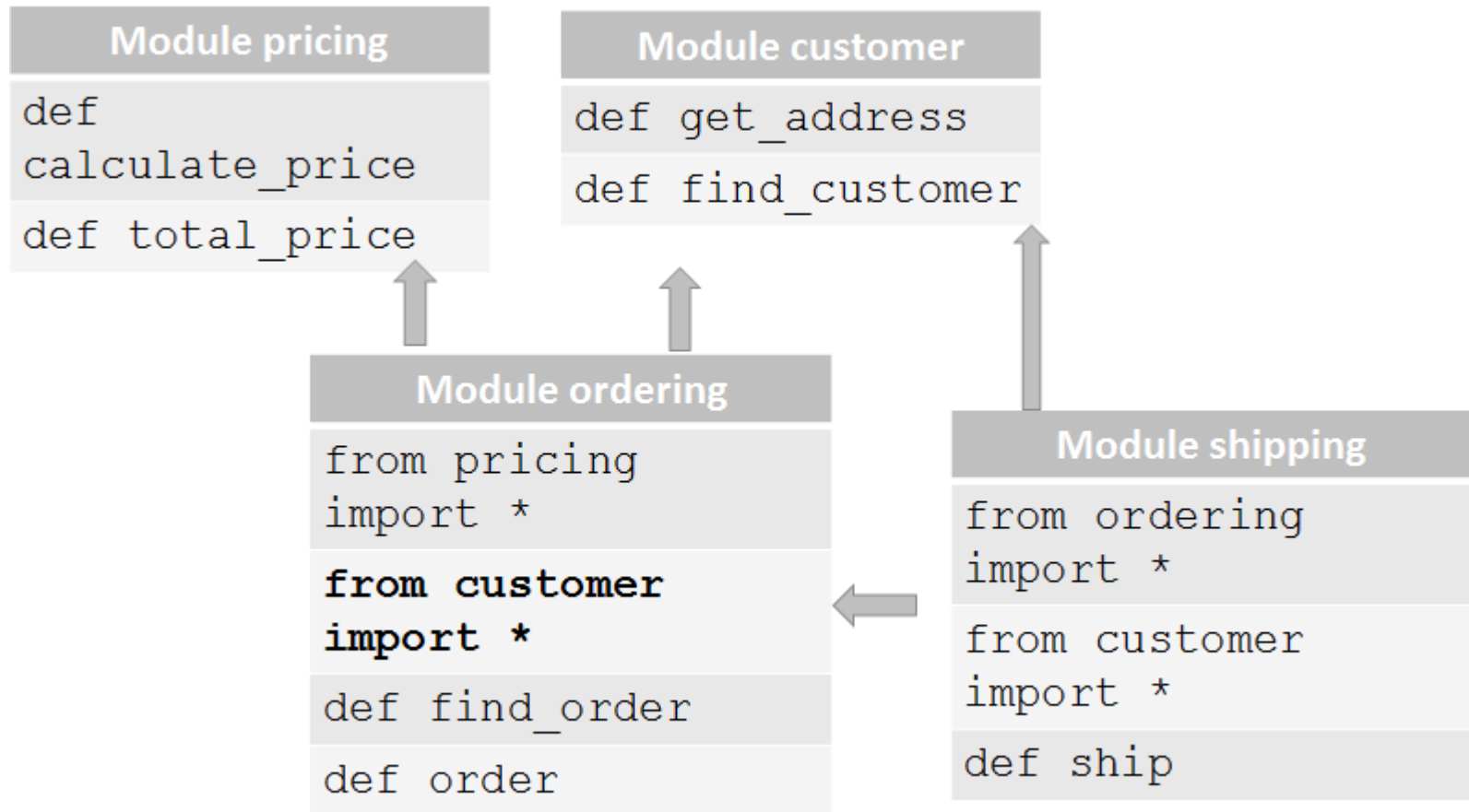
7

MODULE

7.1

ÜBERSICHT

Ein modularisiertes Projekt



- Gekapselte Einheiten
- Wiederverwendung von Code
- Libraries

- Konsequenzen
 - Komplexere Projektorganisation
 - Abhängigkeiten zu externen Projekten mit unbekannten Release-Zyklen
 - Bug-Fixes erfordern erhöhten Kommunikationsaufwand
- Lösungen
 - Einsatz eines Build-Werkzeugs mit
 - Modul-Repository
 - Dependency Management
 - PyPI, Python Package Index

7.2

MODUL-DEFINITION

- Module sind nichts weiter als normale Python-Skripte
- Ablage im `PYTHONPATH`
- Module werden importiert
 - Damit stehen die darin deklarierten Elemente zur Verfügung
 - `from MODUL import element|*`
- Jedes Modul definiert einen Namensraum
 - Damit können potenzielle Namenskonflikte ausgeschlossen werden

- Anzeigen mit `dir()`
 - Ausgabe eines String-Arrays
- `__dict__` ist ein Dictionary, das alle Variablennamen des Moduls mit ihren Inhalten verknüpft

```
def info(module):  
    for i in module.__dict__.keys():  
        print (i, module.__dict__[i])
```

- Durch `import` wird ein Modul unter seinem Namen im aktuellen Namensraum bekannt gemacht

```
import math  
print (math.pow(4,2))
```

- Durch `from import` wird der aktuelle Namensraum erweitert

```
from math import pow  
print (pow(4,2))
```

- `__name__`
 - Enthält den Namen des Moduls
 - `__<name>__` benennt in Python eine interne Variable
- `__main__` bezeichnet das als erstes geladene Modul
 - `__main__` ist eine globale Variable

- **Beispiel:**

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

```
if __name__ == "__main__":  
    print("first modul, starting application...")  
else:  
    print("imported...")
```

- `import` verlangt als Parameter den Modulnamen, keinen String
- Variablen nicht erlaubt
- Programmatische Auswahl des Moduls so nicht möglich
- Lösung:

```
moduleName = "math"  
__import__(moduleName)
```

- Module werden nur einmal geladen
- Die `reload()`-Funktion lädt ein Modul nach
 - Damit werden die darin definierten Elemente aktualisiert
 - Vorsicht:
 - mit `from` importierte Namen werden bei `reload()` nicht aktualisiert

7.3

EIGENE MODULE


```
prices = {"blueray": 5.99, "cd": 9.99, "dvd": 14.99}
discounts = {"blueray": 0, "cd": 30, "dvd": 10}
```

```
def calculate_price(artikel):
    price = prices.get(artikel)
    discount = discounts.get(artikel)
    discounted_price = price * (1 - discount/100)
    return price
```

```
def total_price(price, number):
    return price * number
```

```
from pricing import calculate_price
from pricing import total_price
from customer import find_customer

orders = []

def order(articel, number, customer_id):
    price = calculate_price(articel)
    total = total_price(price, number)
    customer_name = find_customer(customer_id)
    order = {"articel": articel, "total": total, "customer":
customer_name}
    orders.append(order)
    return len(orders) - 1

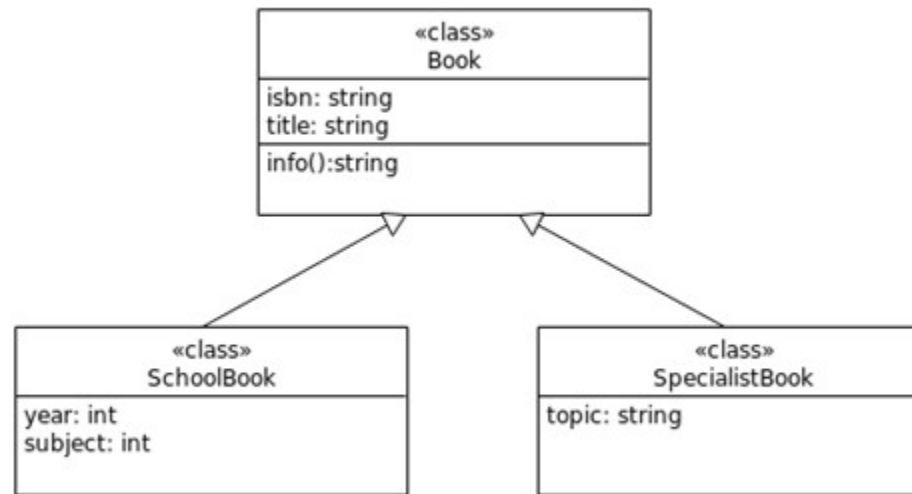
def find_order(id):
    return orders[id]
```

8

OBJEKTORIENTIERUNG

8.1

ÜBERSICHT



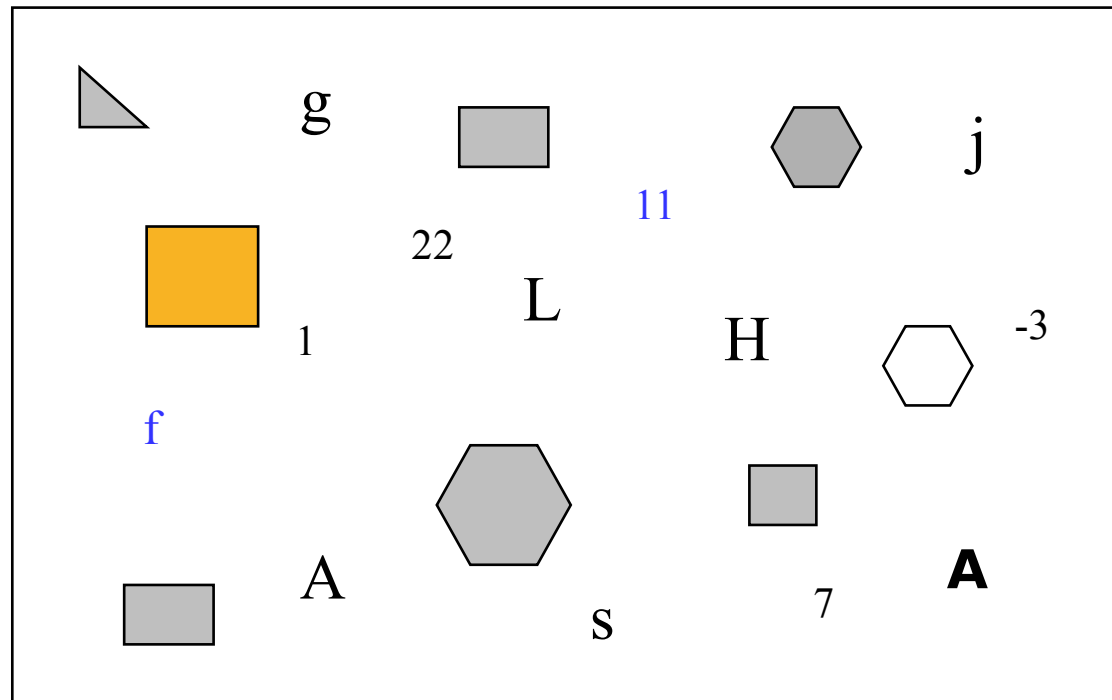
```
class Book(object):
    def __init__(self, isbn, title):
        self.isbn = isbn
        self.title = title

    def info(self):
        return "Book: isbn=%s, title=%s" % (self.isbn, self.title)

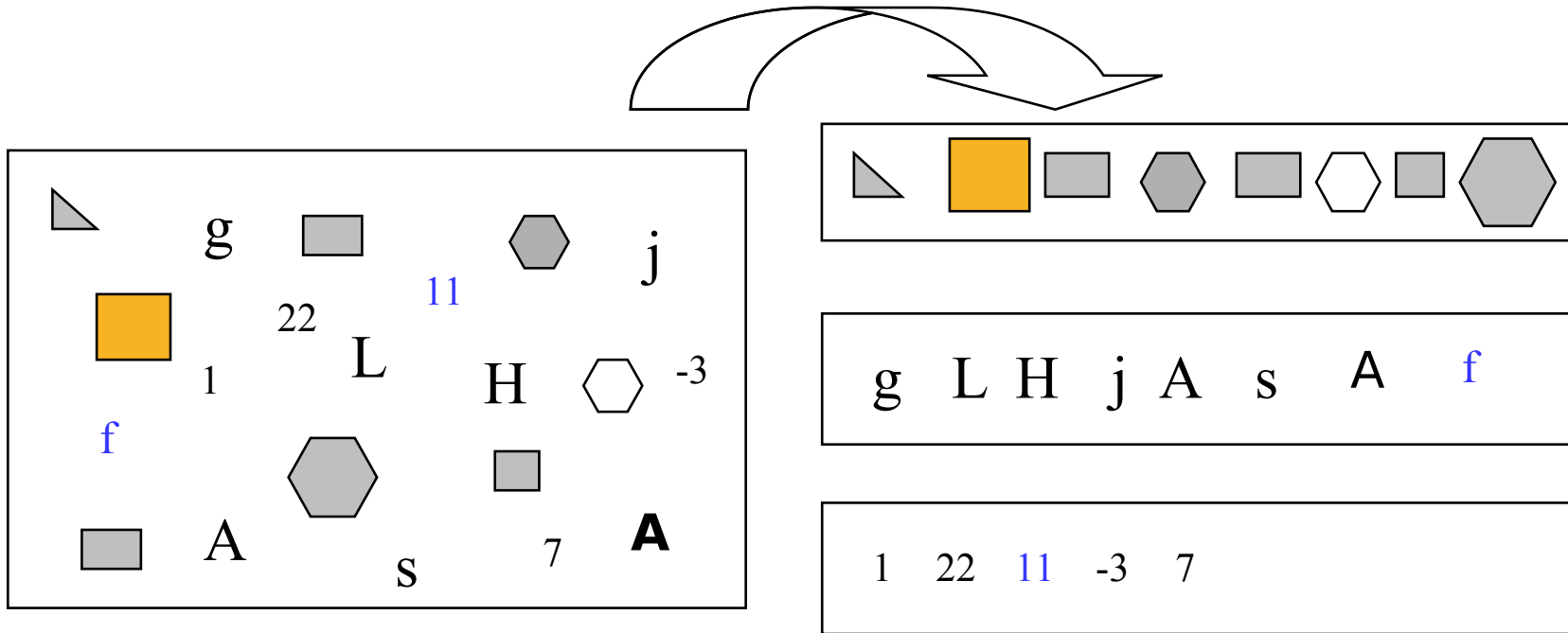
class SchoolBook(Book):
    def __init__(self, isbn, title, year, subject):
        Book.__init__(self, isbn, title)
        self.year = year
        self.subject = subject

class SpecialistBook(Book):
    def __init__(self, isbn, title, topic):
        Book.__init__(self, isbn, title)
        self.topic = topic
```

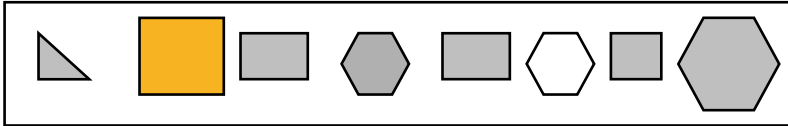
- Komplexes, ungeordnetes System, Zusammenhänge?
 - Was ist wesentlich, was unwesentlich?
 - Existieren Abhängigkeiten?



- Vorgehensweise
 - Klassifizieren
 - Abstrahieren
 - Ordnen, Bilden von Hierarchien
- Ein "menschlicher" Lösungsansatz!



- Ein Zeichnungsobjekt hat eine Farbe, eine Position und eine Größe als Eigenschaften. Das komplexe Zeichnungsobjekt ist eine Komposition einfacherer Elemente.



Zeichnungsobjekte

Farbe
Position
Größe

g L H j A s A f

Buchstaben

Zeichen
Schriftart
Farbe
Position
Größe

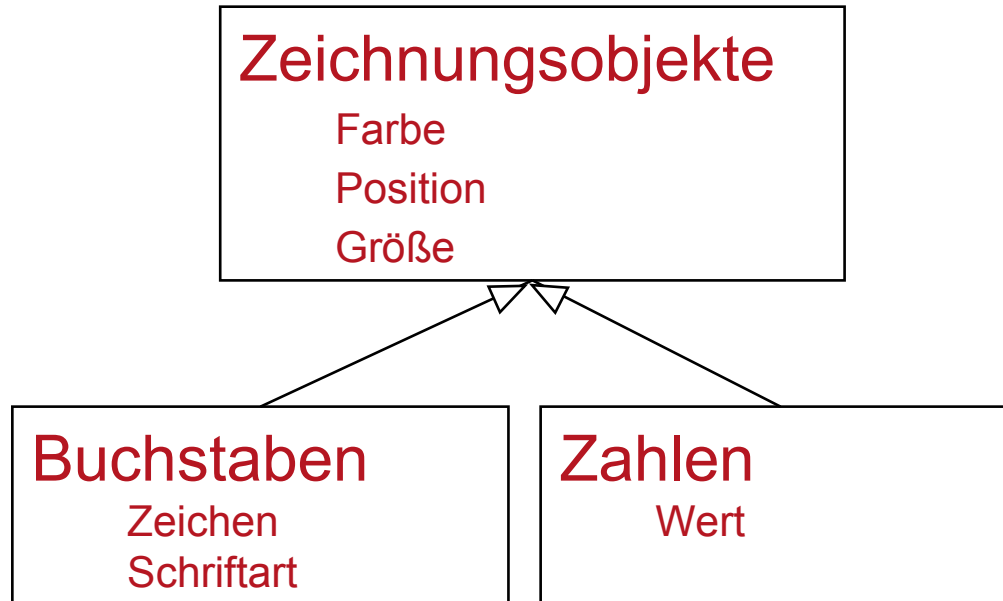
1 22 11 -3 7

Zahlen

Wert
Farbe
Position
Größe

- Ein Buchstabe ist ein Zeichnungsobjekt, das ein Zeichen in einer Schriftart darstellt
- Eine Zahl ist ein Zeichnungsobjekt, das einen Zahlenwert darstellt
- Buchstaben und Zahlen sind Zeichnungsobjekte und erben automatisch auch alle Eigenschaften eines Zeichnungsobjekts

.

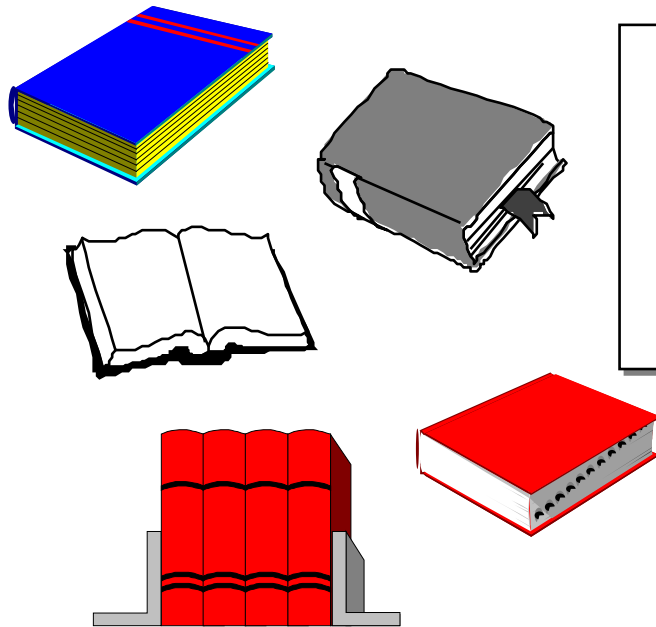


- Ein Objekt besitzt
 - Eigenschaften \Rightarrow **Attribute**
 - Fähigkeiten \Rightarrow **Methoden**
 - Interaktivität \Rightarrow **Botschaften**

- Analogie zu traditionellen Programmen
 - Attribute \Leftrightarrow Variable
 - Methoden \Leftrightarrow Funktionen, Prozeduren
 - Botschaften \Leftrightarrow Ablaufsteuerung, Parameter

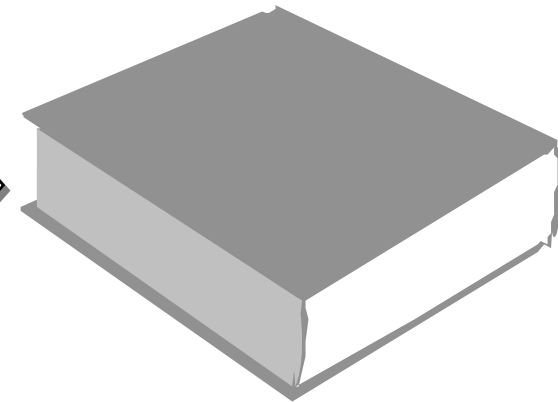
- Gleichartige Objekte werden zu Klassen abstrahiert
 - Eine Klasse dient als Vorlage, Bauanleitung für (mehrere) Objekte

Objekte



**Abstraktion
der
gemeinsamen
Eigenschaften**

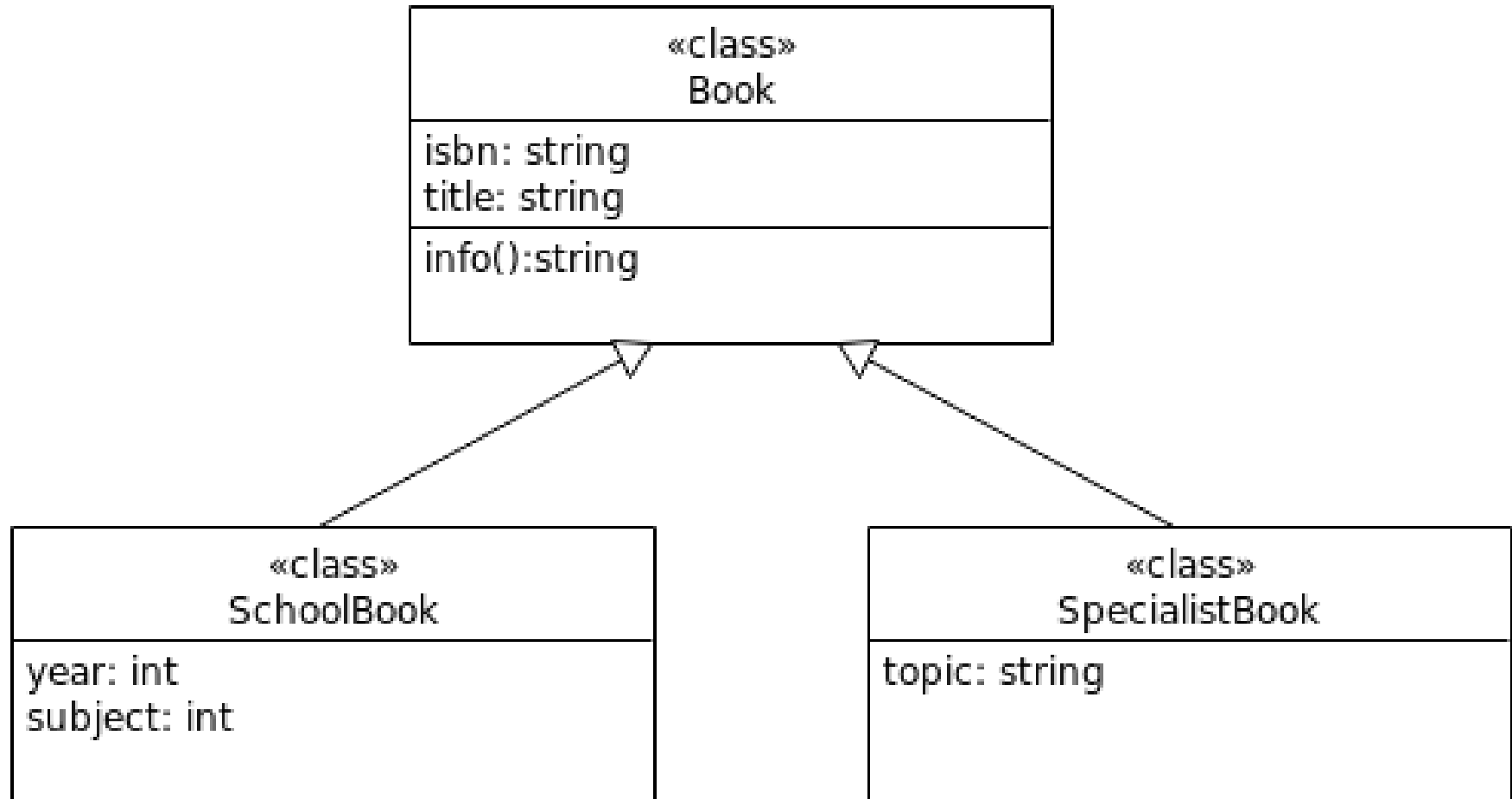
Klasse



Ähnliche Objekte
werden durch eine
Klasse beschrieben

- Bildhaft lässt sich eine Klasse in Form eines Klassendiagramms darstellen
 - Klassendiagramme sind Bestandteil der sogenannten UML-Notation (Unified Modelling Language)
- Dabei werden die Klassen als Rechtecke gezeichnet, die die Attribute und Methoden enthalten
- Die Hierarchie von Klassen wird über gerichtete Pfeile dargestellt

Beispiel: UML - Klassendiagramm



8.2

UMSETZUNG IN PYTHON

- Mit `class` wird in Python eine neue Klasse definiert
- Eine Klasse bildet effektiv einen neuen Namensraum
- Jede Methode bekommt als ersten Parameter `self`
 - Eine Referenz auf das aktuelle Objekt
- Attribute werden über `self` angesprochen
- `__init__` ist der Konstruktor
- Eine Vererbung wird durch die Übergabe der Superklasse an die Klassendefinition der Subklasse realisiert

```
class SubClass(SuperClass)
```

```
class Book(object):
    def __init__(self, isbn, title):
        self.isbn = isbn
        self.title = title

    def info(self):
        return "Book: isbn=%s, title=%s" % (self.isbn, self.title)

class SchoolBook(Book):

    def __init__(self, isbn, title, year, subject):
        Book.__init__(self, isbn, title)
        self.year = year
        self.subject = subject

class SpecialistBook(Book):

    def __init__(self, isbn, title, topic):
        Book.__init__(self, isbn, title)
        self.topic = topic
```

- `__repr__`
 - die Stringrepräsentation erzeugen
- `__getitem__`
 - Array-ZugriffVerwendung von []
- `__setitem__`
 - setter einer Variable bei Zuweisung

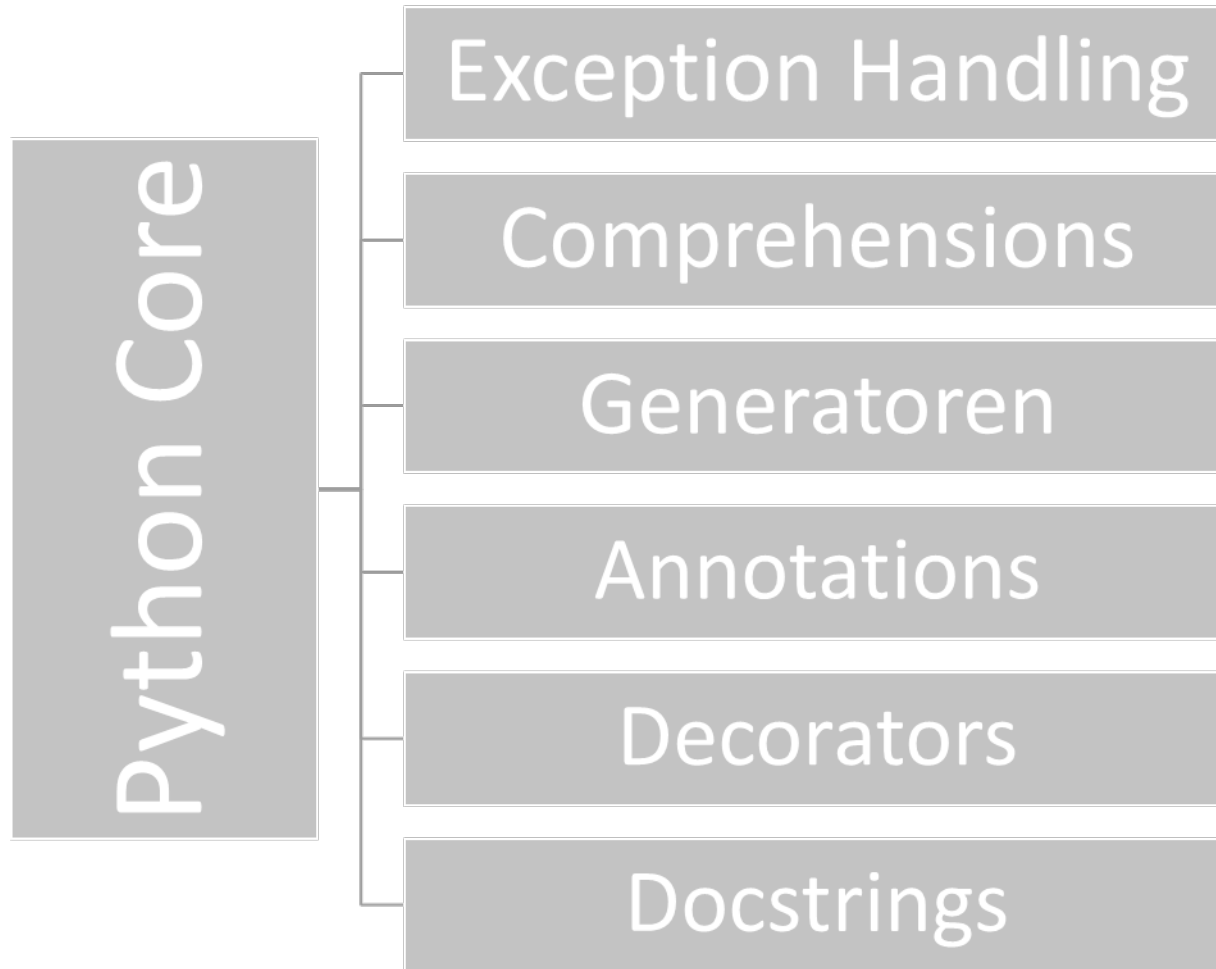
- Eine Implementierung bestimmter Methoden ermöglicht die Verwendung der Operatoren für Objekten
 - `__add__`
 - `+`
 - `__or__`
 - `|`
 - `__mult__`
 - `*`
 - `__lt__`, `__le__`, `__eq__`, `__ne__`, `__ge__`, `__gt__`
 - `<`, `<=`, `==`, `!=`, `>=`, `>`

9

WEITERE KONZEPTE

9.1

ÜBERSICHT



9.2

EXCEPTION HANDLING

- Fehlerbehandlung über Exceptions

```
try:  
    ANWEISUNGEN  
  
except NAME:  
    ANWEISUNGEN  
  
except NAME, DATA:  
    ANWEISUNGEN  
  
except:  
    ANWEISUNGEN  
  
else:  
    ANWEISUNGEN
```

- Es wird immer der zuerst passende Anweisungsblock ausgeführt
 - Exceptions sind hierarchisch organisiert (Objekthierarchie)
- Der `else:` Block wird dann ausgeführt, wenn keine Exception erzeugt wurde
- Ein `finally`-Block kann angegeben werden, um den darin befindlichen Code immer auszuführen
 - egal ob Exception oder nicht

- Eigene Exceptions "werfen" über `raise`
- Parameter können Objekte oder Instanzen der Error-Klassen sein
- Optional können noch Zusatzdaten angegeben werden

```
raise ArithmeticError, "Ni"
```

9.3

COMPREHENSIONS

- Eine Alternative zum bereits angesprochenen Streaming-Konzept der Collections
- Allerdings werden hier neue Syntax-Konstruktionen eingeführt
 - `result = [x**2 for x in range(1, 10) if x%2 == 0]`
- List Comprehensions sind in der Python-Community sehr beliebt und werden in vielen Anwendungen benutzt

9.4

GENERATOREN

- Ein Generator wird erzeugt und einer angegebenen Funktion übergeben
 - Im Wesentlichen das selbe wie eine Callback-Funktion
 - Allerdings auch hier wiederum eine Sprach-Erweiterung
- Dazu benutzt der Generator die `yield`-Anweisung

```
def square_generator(n):  
    i = 1  
    while i <= n:  
        yield i*i  
        i += 1
```

- Benutzung des Generators durch `for - in`

```
for i in square_generator(10):  
    print(i)
```


9.5

ANNOTATIONS

- Annotationen sind "Meta-Informationen", die an bestimmten Stellen des Programms eingesetzt werden können
- Damit können effektiv Erweiterungen der Sprache eingeführt werden, ohne jedesmal die Syntax und den Compiler zu verändern
- Annotation müssen aktiv ausgewertet werden
 - Werkzeuge
 - Innerhalb der eigenen Programmlogik

- In Python werden Annotationen bisher nur für Funktionen genutzt

- Parameter
- Rückgabewert

```
def funktion(p1: Annotation1, p2: Annotation2) ->  
    Annotation3:  
    ...
```

- Damit kann beispielsweise eine Typisierung eingeführt werden

```
def strmult(s: str, n: int) -> str:  
    return s*n
```

- Die Prüfung des Typs erfolgt dann als Bestandteil des Programms oder besser durch eine Vorprüfung, einen "Linter"

9.6

DOCSTRINGS

- Funktionen können am Anfang einen Blockstring `"""` beinhalten, welcher automatisch in der Variablen `__doc__` gespeichert wird

```
def mult(a,b):  
    """multipliziert a und b"""  
    return a*b
```

- Auf diesen "Doc-String" wird über den Namen der Funktion zugegriffen

```
mult.__doc
```

- Damit kann eine technische Dokumentation automatisch erstellt werden
 - z.B. PyDoc

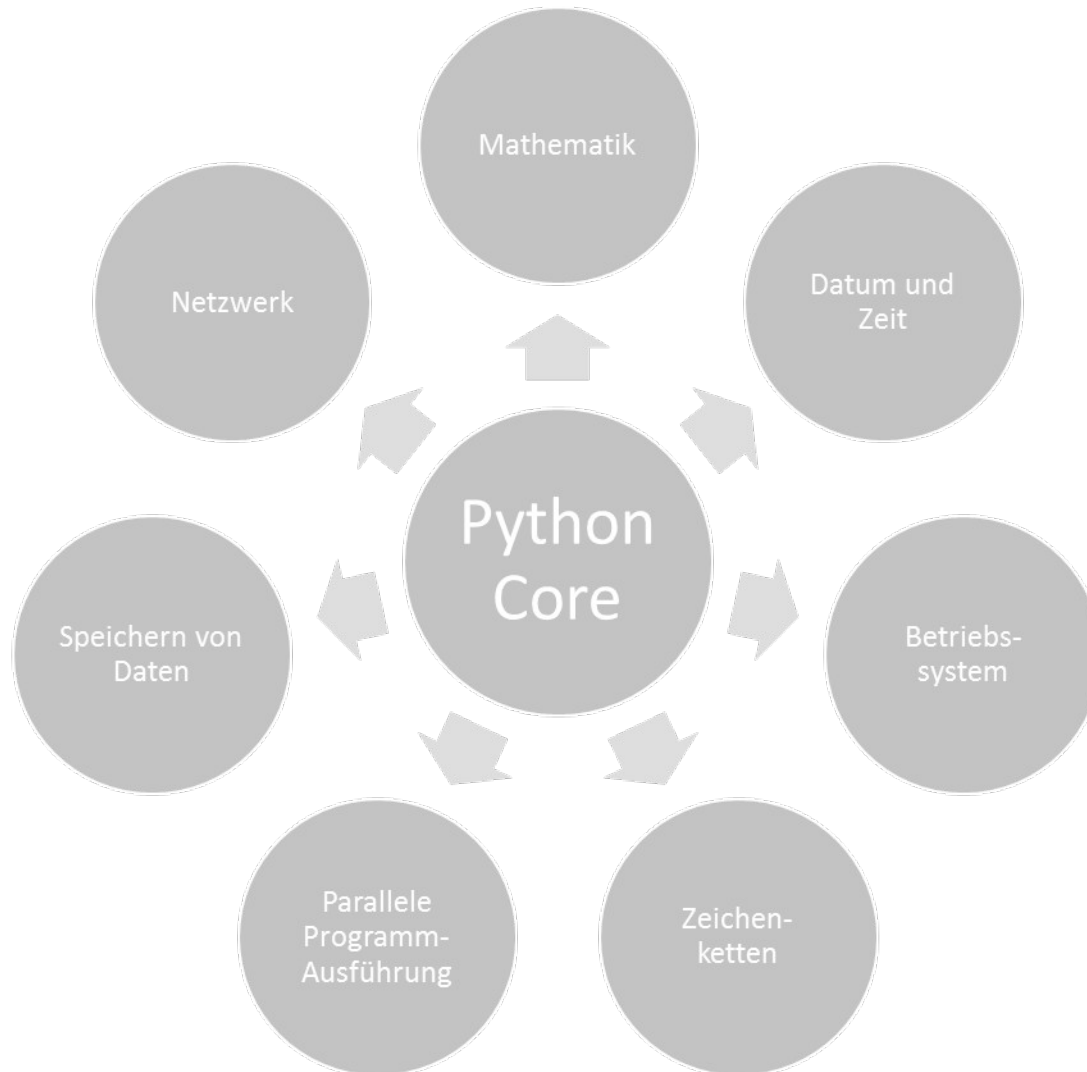
10

DIE STANDARDBIBLIOTHEK

10.1

ÜBERSICHT

Verwendung mächtiger Funktionen aus vorhandenen Bibliotheken



10.2

MATHEMATIK

- `import math`
- `math.sqrt(5)`
- `2.2360679774997898`

- Konstanten
 - Pi, e, ...
- Mathematische Berechnungen
 - Runden, Maximal- und Minimalwerte
 - Trigonometrische Berechnungen
 - ...

- `random()`
 - Zufallswert zwischen 0 und 1
- `randrange()`
- Zufallswert aus gegebenem Bereich
- `randint(min, max)`
- Ähnlich *`randrange()`*, jedoch ist hier die Obergrenze enthalten
- `sample(liste, anzahl)`
 - sucht eine Anzahl von Elementen zufällig aus der gegebenen Liste aus
 - Ergebnis ist eine Liste
- `choice(liste)`
 - sucht sich ein Element zufällig aus der Liste aus

10.3

BETRIEBSSYSTEM

```
import os

os.name # OS basistyp (NT,mac,posix, riscos)
os.platform # Platform (win32,linux,solaris,...)
os.sep # \ für win; / fuer unix
os.linesep # \r\n für win; \n für unix; \r fuer mac
```

```
import os

os.path.isfile      # prüft, ob Parameter Datei ist
os.path.isdir       # prüft, ob Parameter Verzeichnis ist
os.path.getsize     # Dateigröße
os.listdir           # Einträge im Verzeichnis
os.chdir             # wechselt das aktuelle Verzeichnis
os.getcwd            # liefert das aktuelle Verzeichnis
os.realpath          # löst relative Verzeichnisnamen auf
```

- `stdin/stdout/stderr`
 - Die Standardein- und -ausgaben
- `argv:`
 - Die Liste der Skript-Argumente
- `path`
 - Liste aller Verzeichnisse des PythonPath
- `platform`
 - Textrepräsentation über das Betriebssystem
- `modules`
 - Dictionary aller geladenen Module

- Regular Expressions
- Text mit Metazeichen
 - . Beliebiges Zeichen
 - * Beliebig oft es Wiederholen des Zeichens links (auch gar nicht)
 - + Beliebig oft es Wiederholen des Zeichens links (mind. 1-Mal)
 - ? Einmaliges oder gar kein Vorkommen des Zeichens links
 - {n} n-Maliges Vorkommen des Zeichens links
 - {n,m} Bereichsangabe
 - [] Ein Zeichen aus der Liste: [a-z]
 - () Gruppenbildung
 - \w: Alphanumerisches Zeichen
 - \W: Kein alphanumerisches Zeichen
 - \d: Ziffer \D: keine Ziffer
 - \s: Whitespace / \S: Kein Whitespace
 - \\: Der Backslash

■ Verwendung von RE über gleichnamiges Modul

```
■ >>> name="Meyr"
■ >>> import re
■ >>> pattern=re.compile("^M[ae][iy]e?r$")
■ >>> match=pattern.match(name)
■ >>> if match:
■     print "Es ist ein Maier"

■ Es ist ein Maier
■ >>> name="Meier"
■ >>> match=pattern.match(name)
■ >>> if match:
■     print "Es ist ein Maier"

■ Es ist ein Maier
```

10.4

DATUM UND UHRZEIT

- `time()`
 - Zeit in Sekunden seit dem 1.1.1970 0:00
- `clock()`
 - CPU-Zeit für diesen Prozess
- `sleep(n)`
 - Pause von `n` Sekunden
- `gmtime(t)`
 - nimmt Sekunden seit 1.1.1970 als Parameter und generiert daraus ein 9er Tupel mit den Informationen
 - Jahr (4stellig)
 - Monat
 - Tag im Monat
 - Stunde
 - Minute
 - Sekunde
 - Wochentag (Montag ist 0)
 - Tag im Jahr
 - DST (Daylight Saving)

- `strftime()`
 - gibt die Zeit gemäß dem gegebenen Formatstring formatiert aus:
`time.strftime("Uhrzeit: %H:%M:%S %d.%m.%Y")`
- `strptime(String)`
 - Parst einen gegebenen String nach gegebenem Format

10.5

STRINGS

- Zeichenketten werden durch einfache oder doppelte Hochkommas definiert
- Die Länge des Strings ist beliebig
- Unicode wird unterstützt
 - Seit Python 3 Standard für Strings
- Strings können addiert werden
 - auch die "Multiplikation" mit einer Zahl ist möglich
- Strings können als Liste von Einzelzeichen aufgefasst werden

- **Format-Anweisung**

```
format % werte (Formate analog printf())  
a = 99  
print "%d bottles of beer on the wall" % (a)  
print "String: %s, Integer: %d" % ("String", 42)
```

- **Die Anzahl der Platzhalter und der Parameter muss übereinstimmen**

```
print "%d %d %s" % ("Integer erwartet")  
Traceback (most recent call last):  
  File "<stdin>", line 1, in ?  
TypeError: int argument required
```


- `lstrip, rstrip, strip`
 - Whitespaces links, recht oder an beiden Enden des Strings entfernen
- `replace(s, old, new)`
 - im String `s`, `old` durch `new` ersetzen
- `capword(s)`
 - alle Wörter im String `s` groß schreiben, optionales Trennzeichen
- `expandtabs(s)`
 - Alle Tabulatoren durch Leerzeichen ersetzen, Default 8 Leerzeichen
- `rjust, ljust, center`
 - Einen String auf gegebene Breite links-, rechtsbündig oder zentriert darstellen
- `find`
 - im String suchen
- `rindex, rfind`
 - von rechts suchen

- Reguläre Ausdrücke werden benutzt, um komplexe Zeichenkettenverarbeitung zu realisieren
 - Ein unabhängiges Sprachkonzept, das von allen modernen Sprachen unterstützt wird
- Dazu werden eine Reihe von Pattern-Ausdrücken und Sonderzeichen definiert
- String-Methoden arbeiten teilweise direkt mit regulären Ausdrücken
 - `split`
 - `sub`

10.6

PARALLELE PROGRAMMAUSFÜHRUNG

- Der `fork`-Befehl des Moduls `os` erstellt eine genaue Kopie des aktuellen Prozesses
- Das Modul `_thread` ermöglicht den parallelen Ablauf in mehreren Threads
 - `pid=thread.start_new(funktion, (parameterliste,))`
- Thread-Synchronisation über Mutex
 - `mutex=thread.allocate_lock()` #neuen Mutex erstellen
 - `mutex.acquire()` #wartet so lange, bis exklusiver Zugriff möglich
 - `mutex.release()` #anderen Zugriff erlauben
- Das Modul `threading` bietet einen objektorientierten Ansatz

10.7

SPEICHERN VON DATEN

- Dateien werden mit der Klasse `file` geöffnet, bzw. das Dateiojekt erstellt
 - Beim Öffnen muss der Modus angegeben werden, lesen "r", schreiben "w" oder beides
- Methoden
 - `read`
 - `readline`
 - `readlines`
 - `write`
 - `writelines`
 - `close`

- `glob` bietet die Möglichkeit, Dateien aufgrund der Dos/Unix Wildcards zu finden
- `glob("Pattern")`
 - Gibt eine Liste der Dateinamen zurück, auf die dieses Pattern passt
 - Beispiel: `glob.glob("*")`

- Pickle dient zur einfachen Serialisierung von Datenstrukturen
 - Diese können als Byte-Repräsentation beispielsweise in eine Datei geschrieben oder gelesen werden


```
import pickle
data = {"a": "b", "c": C()}
# Daten abspeichern.....
f = file(r"c:\pickle.txt", "w")
pickle.dump(data, f)
f.close()
```

```
f = file(r"c:\pickle.txt", "r")
newdata = pickle.load(f)
f.close()
print str(newdata)
```

10.8

NETWORKING

- Nach dem Import des Moduls socket steht die Networking-Funktionalität von Python zur Verfügung
- Low-Level TCP/IP-Verbindungen
- Höherwertige Protokolle wie http oder Web Services werden durch weitere Module abgebildet
 - Jedoch nicht Bestandteil von Python Core
 - Einbindung beispielsweise über PyPI

10.9

GUI MIT TKINTER

- Modul Tkinter
- Bietet Zugriff auf die wichtigsten Gui-Widget

```
import Tkinter  
  
mainwindow=Tkinter.Tk()  
  
widget=Tkinter.Label(mainwindow, text='King Athur')  
  
widget.pack()  
  
mainwindow.mainloop()
```

- Pack bestimmt die Größe der Widgets
- Nimmt immer den "übrigen" freien Bereich

```
>>> from Tkinter import *  
>>> root=Tk()  
>>> Button(root,text="B1").pack(side=LEFT, fill=BOTH)  
>>> Button(root,text="B2").pack(side=TOP, fill=BOTH)  
>>> Button(root,text="B3").pack(side=RIGHT, fill=BOTH)  
>>> root.mainloop()
```



- Buttons rufen Funktionen auf, wenn sie gedrückt werden
- Geht auch allgemein mit "Events" (z. B. Mausklick, Tastendruck etc.)

```
>>> def say():  
    print "polly wanna cracker"
```

```
>>> Button(root, text="Say: 'Polly wanna  
    Cracker'", command=say).pack(side=LEFT, fill=BOTH)
```

- Textfeldeingaben über das Widget *Entry*
- Eingaben bekommt man mit der Methode `get`

```
>>> root=Tk()

>>> ent=Entry(root)

>>> ent.pack(side=TOP)

>>> btn=Button(root,text="ok")

>>> btn.config(command=lambda event=None:
    sys.stdout.write(ent.get()))

>>> btn.pack(side=BOTTOM)

>>> root.mainloop()
```