Programmieren mit python™

Vorstellung der Multiparadigmen-Sprache

W3L AG
info@W3L.de

2014

Inhalt

- Entstehung
- Besonderheiten
- Programmteile
- Klassen
- Anweisungen
- Generatoren
- Vererbung
- Slicing
- Funktionale Programmierung
- Prototypbasierte Programmierung
- Ausführungsgeschwindigkeit
- Ausblick

Entstehung

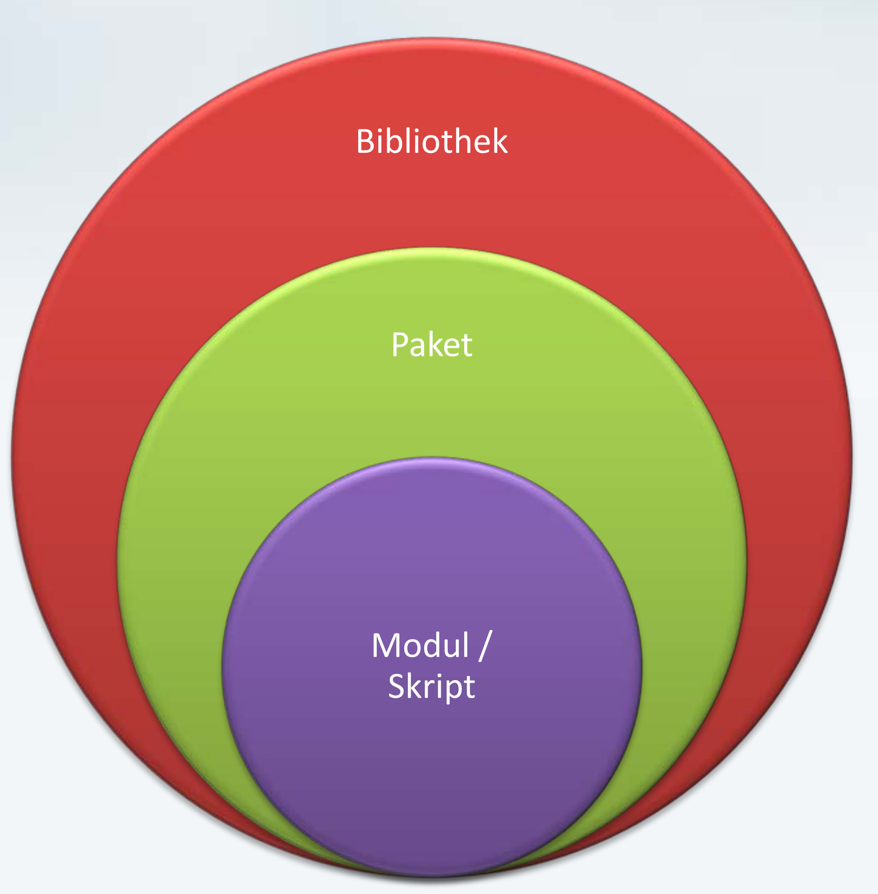
- **Hauptentwickler**
  - Guido van Rossum
  - Geb. 31. Januar 1956
- **Beginn der Entwicklung**
  - Anfang der 90er-Jahre
- **Ziele der Programmiersprache**
  - Einfach erlernbar
  - Gut lesbar
  - Mächtig
- **Aktuelle Versionen**
  - 2.7.6 vom 10. November 2013
  - 3.4.0 vom 17. März 2014

Besonderheiten

- **Blöcke werden durch Einrückung gekennzeichnet** (Leerzeichen oder Tabulatoren)
- **Multiparadigma-Unterstützung**
- **Keine statische Typprüfung**
- **Wenig Schlüsselwörter**
- **Quellcode-Dateien werden in Byte-Code übersetzt**
  - Plattformunabhängig
  - Byte-Code wird interpretiert
  - Kein JIT-Compiler
- **Garbage Collection**
- **Interaktiver Modus**

Programmteile

- **Bibliotheken / Frameworks**
- **Pakete**
  - Sammlungen von Modulen in einem gemeinsamen Verzeichnis
- **Module**
  - Für die Einbindung in anderen Modulen und Skripten vorgesehen
- **Skripte**
  - Für die direkte Ausführung vorgesehen



Programmbeispiel

```
def square(n):
    return n**2

def bubblesort(values, comparer):
    for n in range(len(values) - 1, 0, -1):
        for i in range(n):
            if comparer(values[i], values[i + 1]):
                values[i], values[i + 1] = values[i + 1], values[i]

def main():
    numbers = [5, 3, 6, 2, 9, 8, 1, 7, 4]
    strings = ["Hello", " World", "!"]

    bubblesort(numbers, lambda x, y: x < y)
    bubblesort(strings, lambda x, y: len(x) < len(y))

    for i in (square(n) for n in numbers if n % 2 == 0):
        print(i)

    for string in strings:
        print(string)

if __name__ == '__main__':
    main()
```

Klassen

- **Konstruktor**
  - `__init__`
- **Destruktor**
  - `__del__`
- **Attribute**
- **Eigenschaften**
  - `@property`
- **Klassenattribute**
- **Instanzmethoden**
- **Klassenmethoden**
  - `@classmethod`
- **Statische Methoden**
  - `@staticmethod`
- **„Sichtbarkeiten“**
  - Konventionen

```
class Node:
    __instanceCount = 0

    def __init__(self, content=None):
        self.__content = content
        Node.__instanceCount += 1

    @property
    def content(self):
        return self.__content

    @content.setter
    def content(self, content):
        self.__content = content

    def __str__(self):
        return str(self.content)

    @staticmethod
    def get_instance_count():
        return Node.__instanceCount
```

Anweisungen 1

- **Zuweisungen**
  - Mehrere Zuweisungen in einer Anweisung möglich
  - Vertauschung von Variableninhalten
- **Verzweigungen**
  - Einfache Bereichsprüfung
- **Ausnahmebehandlung**

```
x, y = "Hello", "World"
x, y = y, x
print(x, y)
#> World Hello

x = 3
if 2 <= x <= 5:
    print("x is in [2..5]")
elif x == 6:
    print("x is equal 6")
#> x is in [2..5]

try:
    raise Exception()
except Exception as ex:
    print("Exception caught")
finally:
    print("Finally")
#> Exception caught
#> Finally
```

Anweisungen 2

- For-Schleifen
- While-Schleifen
- List-Comprehensions
- Set-Comprehensions
- Generator-Expressions
- Lambda-Ausdrücke

```
oneToFive = range(1, 6)

# List-Comprehension
lessThanFour = [i for i in oneToFive if i < 4]

# Set-Comprehension
predicate = lambda x: x % 2 == 0
even = {i for i in oneToFive if predicate}

# Generator-Expression
doubled = (i * 2 for i in oneToFive)
```

Funktionen

- **Übergabe von Argumenten nur mit call-by-value**
- **Funktionen und Methoden geben stets einen Wert zurück**
  - Standardwert: `None`
- **Optionale Parameter**
  - `myDefault=42`
  - Standardwert sollte unveränderlich sein

```
def sum(*values):
    result = 0
    for value in values:
        result += value
    return result

print(sum(1, 2, 3))
#> 6

def show_args(**kargs):
    print(kargs)

show_args(One=1, Two=2, Three=3)
#> {'Three': 3, 'Two': 2, 'One': 1}
```

Generatoren

- Funktionen, die Iteratoren generieren
- Werte werden erst bei Bedarf zurückgegeben
- Verkettung mehrerer Generatoren effizient möglich
- Rückgabe einzelner Werte mit yield

```
def repeat(count):
    for i in range(count):
        current = i + 1
        print("  before yield", current)
        yield None
        print("  after yield", current)

x = repeat(2)
print("before loop")
for i, __ in enumerate(x):
    print("    loop body", i + 1)
print("after loop")
#> before loop
#>   before yield 1
#>     loop body 1
#>   after yield 1
#>   before yield 2
#>     loop body 2
#>   after yield 2
#> after loop
```

Typsystem

- **Keine statische Typprüfung**
- **Objekte haben stets einen Typ**
- **Wurzel der Typhierarchie: `object`**
- **Typumwandlungen ggf. notwendig**

```
print(1 + "2")
#> TypeError: unsupported operand type(s) for +: 'int' and 'str'

print(1 + int("2"))
#> 3

print(isinstance(3, object))
#> True

print(isinstance(lambda: True, object))
#> True
```

Vererbung

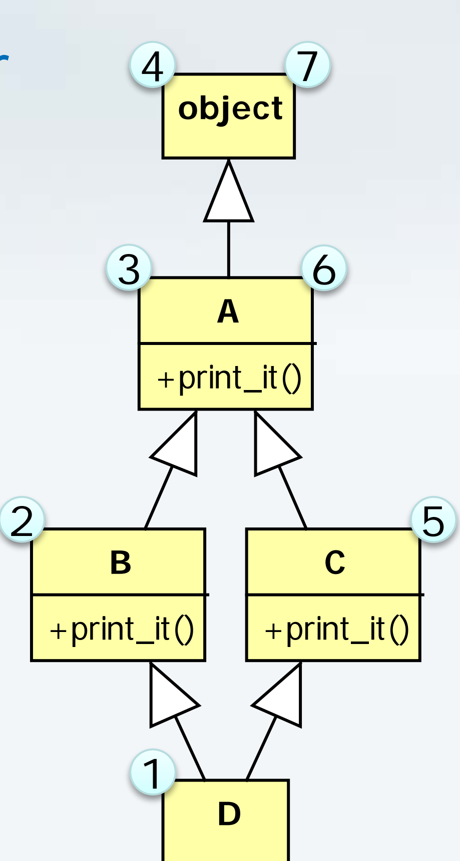
- **Wurzel der Klassenhierarchie: `object`**
- **Konstruktor wird vererbt**
- **Subklassen sind nicht verpflichtet den Konstruktor einer Superklasse aufzurufen**
- **Keine Methodenüberladungen**
  - Methoden mit optionalen Parametern als Alternative
- **Überschreiben von Methoden**
  - Änderung der Methodensignatur möglich
  - Überschreibende Methode kann die Anzahl der Parameter ändern
  - Aufruf einer überschriebenen Methode mit der Methode `super` möglich
- **Mehrfachvererbung wird unterstützt**

Mehrfachvererbung

- **Method Resolution Order bestimmt die zu verwendende Implementierung einer Methode ausgehend von einer Klasse**
- **Python verwendet zur Bestimmung den C3-Algorithmus**
  - Tiefensuche
  - Method Resolution Order gleichbleibend (*monotonic*)
    - „*if C1 precedes C2 in the linearization of C, then C1 precedes C2 in the linearization of any subclass of C.*“ [1]
      (C ist Subklasse von C1 und diese ist Subklasse von C2)
  - Berücksichtigung der lokalen Rangordnung
    (*local precedence ordering*)
- **Method Resolution Order einer Klasse kann mit der Klassenmethode `mro` abgerufen werden**

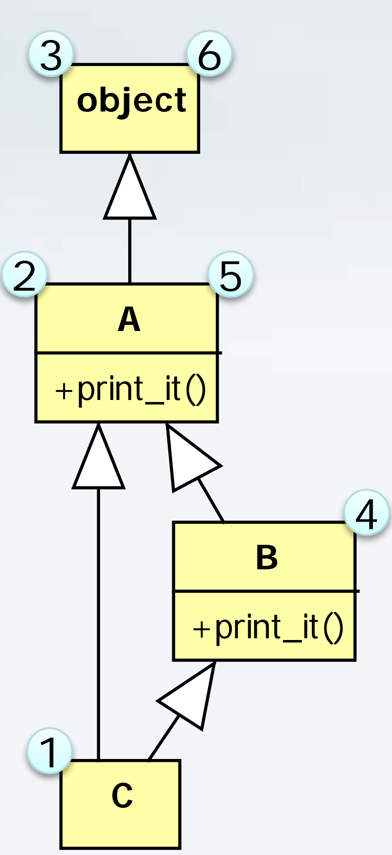
MRO – Beispiel 1

- **Bestimmung der Method Resolution Order der Klasse `D`**
- ***Linearization***
  - Vollständige Liste
    `[D, B, A, object, C, A, object]`
  - Streichen doppelter Klassen
    [D, B, ~~A~~, ~~object~~, C, A, object]
  - Endgültige Liste
    `[D, B, C, A, object]`
- ***Local Precedence Ordering***
  - `B` vor `C`



MRO – Beispiel 2

- **Bestimmung der Method Resolution Order der Klasse `C`**
- ***Linearization***
  - Vollständige Liste
    `[C, A, object, B, A, object]`
  - Streichen doppelter Klassen
    [C, ~~A~~, ~~object~~, B, A, object]
  - Endgültige Liste
    `[C, B, A, object]`
- ***Local Precedence Ordering***
  - A vor B



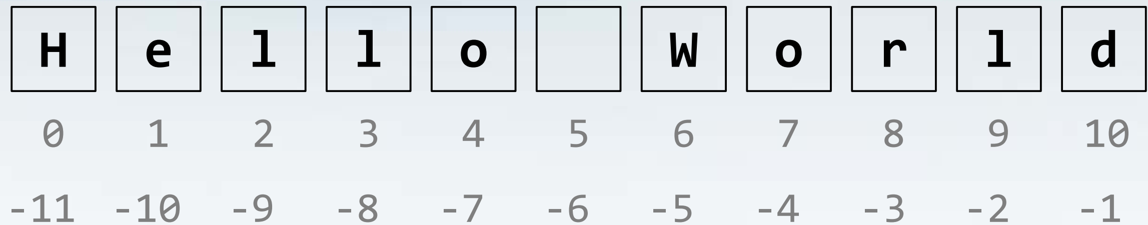
angelehnt an [1]

Slicing

- **Einfache und zugleich mächtige Funktionalität für den Zugriff auf Elemente und Subsequenzen indexbasierter Datenstrukturen**
- **Lesender und schreibender Zugriff möglich**
- **Nutzung in Verbindung mit eigenen Klassen durch Implementierung von `__getitem__` und / oder `__setitem__`**
- **Syntax**
  - x[*<start>*:*<stop>*:*<step>*]
    - *<start>* Standard: 0
    - *<stop>* Standard: (restliche Elemente)
    - *<step>* Standard: 1

Slicing - Lesend

```
x[<start>:<stop>:<step>]
```




```
x = "Hello World"

print(x[-1])
#> d

print(x[:5])
#> Hello

print(x[3:8])
#> lo Wo

print(x[::2])
#> HloWrd
```

Slicing - Schreibend

x[*<start>*:*<stop>*:*<step>*]

H	e	l	l	o		W	o	r	l	d

0 1 2 3 4 5 6 7 8 9 10 11

-11 -10 -9 -8 -7 -6 -5 -4 -3 -2 -1

```
original = list("Hello World")

x = original.copy(); x[:5] = "Greetings"
print(''.join(x))
#> Greetings World

x = original.copy(); x[2:9] = "ro"
print(''.join(x))
#> Herold

x = original.copy(); x[11:11] = "s"
print(''.join(x))
#> Hello Worlds
```

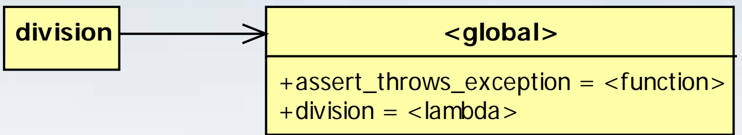
Funktionale Programmierung

- **Funktionen sind First-Class-Objects**
- **Closures**

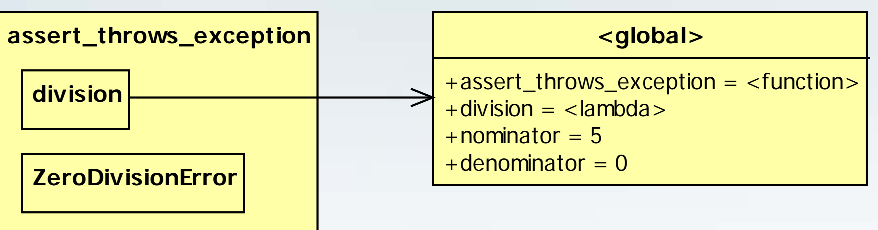
```python
def filter(iterable, predicate):
    return (item for item in iterable if predicate(item))

def map(iterable, selector):
    return (selector(item) for item in iterable)

def reduce(iterable, initial, selector):
    result = initial
    for item in iterable:
        result = selector(result, item)
    return result

list = ["Hello", 42, "World", 9]
temp = filter(list, lambda x: isinstance(x, int))
temp = map(temp, lambda x: x * 2)
temp = reduce(temp, 0, lambda accumulate, x: accumulate + x)
print(temp)
#> 102
```

Closures

Nach Definition von `division`



Innerhalb der Funktion




```
def assert_throws_exception(action, exceptionType):
    try:
        action()
    except exceptionType:
        return
    except:
        raise AssertionError("Expected: " + exceptionType.__name__)
    raise AssertionError("Expected: " + exceptionType.__name__)

division = lambda: nominator / denominator
nominator = 5
denominator = 0
assert_throws_exception(division, ZeroDivisionError)
```

Decorators

- Ergänzen Funktionen und Klassen um Funktionalität
- Verändern aktiv den Programmfluss
- Parametrisierbar
- Mehrfach anwendbar

```
def cache(function):
    storage = {}
    def cache_internal(*args):
        nonlocal storage
        if not args in storage:
            storage[args] = function(*args)
        return storage[args]
    return cache_internal

@cache
def fibonacci(n):
    pass
```

Prototypbasierte Programmierung

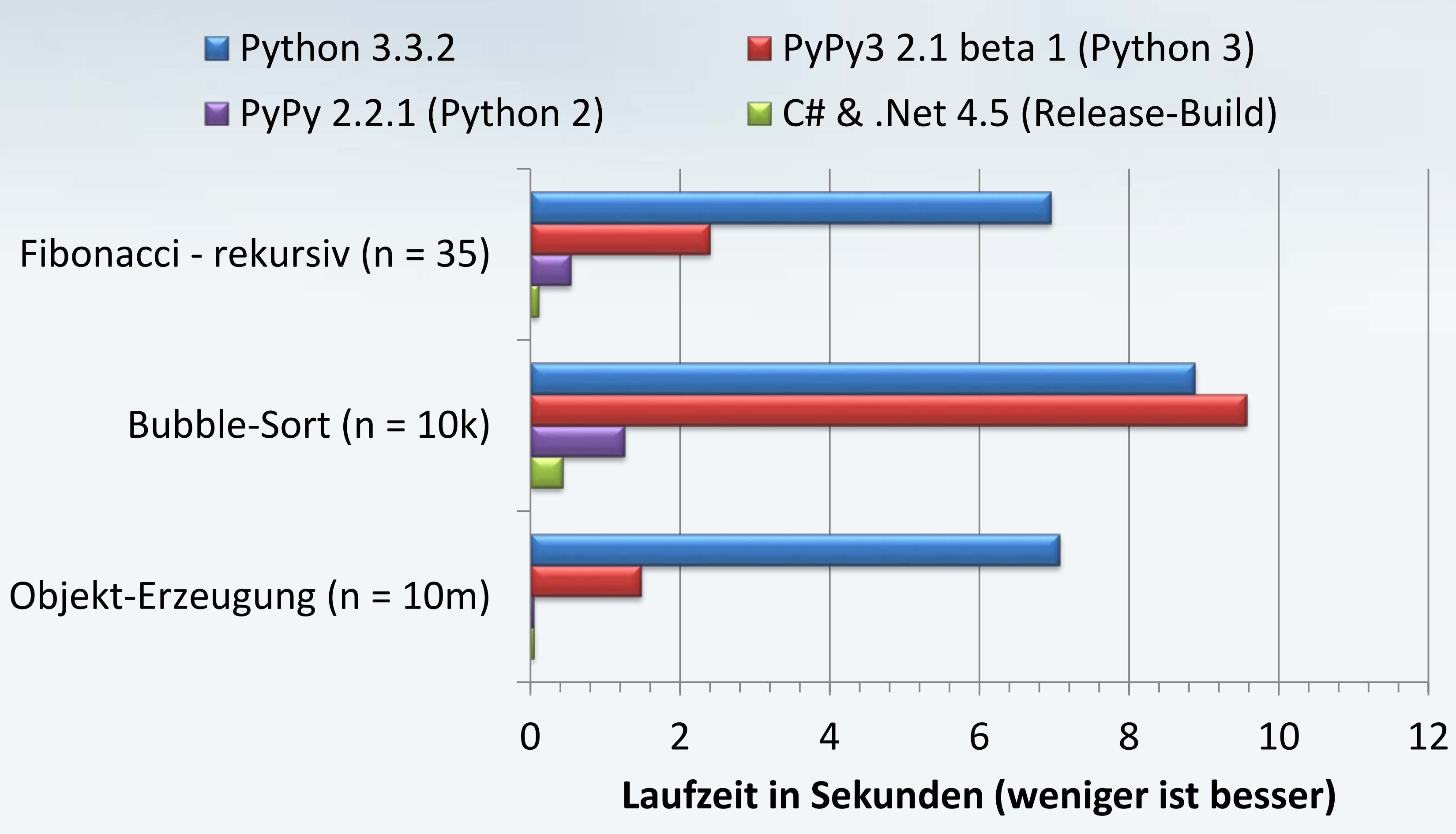
- Flaches / tiefes Kopieren von Objekten
- Klassen können __copy__ und / oder __deepcopy__ implementieren
- Objekte um Attribute ergänzen
- Objekte um Methoden ergänzen

```
import copy
import types

class Sheep:
    pass

copied = copy.copy(Sheep())
copied.cape_color = "red"
copied.fly = types.MethodType(
    lambda self:
        print("Sheep flies around with a cool", self.cape_color, "cape"),
    copied)
copied.fly()
#> Sheep flies around with a cool red cape
```

Ausblick

- **Metaklassen**
  (Klassifizierung von Klassen)
- **Dynamisches Erstellen von Klassen zur Laufzeit**
- **Änderungen an Klassen zur Laufzeit möglich**
  (*Monkey Patching*)
  - Hinzufügen / Entfernen von Methoden
  - Hinzufügen / Entfernen von Attributen
  - Austausch von Methodenimplementierungen
- **Einbindung von C-Bibliotheken**
- **Alternative Implementierungen**
  - IronPython für .NET
  - Jython für Java

Ausführungsgeschwindigkeit



Quellen

[1] Simionato, Michele
The Python 2.3 Method Resolution Order
http://www.python.org/download/releases/2.3/mro/

[2] *Python-Logo*
http://www.python.org/community/logos/python-logo-master-v3-TM.png

Vielen Dank
für Ihre Aufmerksamkeit.

Inhouse-Schulungen

Wir bieten Inhouse-Schulungen und Beratung durch unsere IT-Experten und –Berater.

Schulungsthemen

- Softwarearchitektur (OOD)
- Requirements Engineering (OOA)
- Nebenläufige & verteilte Programmierung

Gerne konzipieren wir auch eine individuelle Schulung zu Ihren Fragestellungen.

Sprechen Sie uns an!
Tel. 0231/61 804-0, info@W3L.de

W3L-Akademie

In Zusammenarbeit mit der Fachhochschule Dortmund bieten wir

zwei Online-Studiengänge

- B.Sc. Web- und Medieninformatik
- B.Sc. Wirtschaftsinformatik

und 7 Weiterbildungen im IT-Bereich an.

Besuchen Sie unsere Akademie!
http://Akademie.W3L.de