Programmierkurs Python I

Stefan Thater & Michaela Regneri
Universität des Saarlandes
FR 4.7 Allgemeine Linguistik (Computerlinguistik)

Übersicht

- mehr zu OOP:
  - Hintergründe
  - Vererbung
- Methoden zur Operatoren-Überladung („Hooks“)

Warum OOP?

- Objektorientierte Programmierung (OOP) ermutigt den Programmierer dazu, Programme in Klassen aufzuteilen.
- Für viele Projekte sind Klassen gute Gliederungsebene, die zu Dingen in der wirklichen Welt passen.
- in einer guten Klassenhierarchie ist die Komplexität einzelner Klassen überschaubar, das macht den Code übersichtlicher

Warum OOP?

- Man kann Implementierungsdetails von Klassen nach außen verbergen
- Andere Programmierer (Benutzer der Klassen) können die Klassen direkt weiterverwenden, oder erweitern, ohne sie zu verändern
- Die Implementierung kann jederzeit verändert werden, ohne das Gesamtprogramm zu stören

Warum OOP?

- Klassen können von anderen Klassen abgeleitet werden.
- Abgeleitete Klassen erben alle Attribute der Basisklasse, können neue dazutun und die geerbten Methoden überschreiben
- Objekte der abgeleiteten Klasse können überall eingesetzt werden, wo Objekte der Basisklasse akzeptiert werden

Wiederholung: Klassen

```
class Rat:

  def __init__(self, num, den):
    self.num = num
    self.den = den

  def mul(self, other):
    num = self.num * other.num
    den = self.den * other.den
    return Rat(num, den)

r1 = Rat(1,2)
r2 = r1.mul(Rat(2,3))
```

- Klassen werden definiert mit Schlüsselwort „class“
- Klassenmethoden mit „*self*“-Parameter
- *init*-Methode als „Konstruktor“
- Instanz-Erzeugung mit Klassen-Name ruft init auf
- Funktionsaufruf mit instanz.methode()

Geltungsbereiche & Namensräume

- Ein Namensraum (Namespace) ist eine Abbildung von Bezeichnern (Namen) auf Objekte
- Namen in verschiedenen Namensräumen können auf verschiedene Objekte referieren
- Man kann sich Namensräume als Dictionaries vorstellen; Schlüssel sind dabei eingeschränkt auf zulässige Namen
- Qualifizierter Zugriff auf Namen (bzw. Objekte) in einem Namensraum: `namespace.attr`

Geltungsbereich (Scope)

- Ein Geltungsbereich ist ein Bereich im Programm, innerhalb dessen man direkt auf Namen zugreifen kann („direkt“ = ohne andere Schküsselwörter)
- Drei (verschachtelte) Namensräume:
  - Eingebaute Namen (z.B. print)
  - Globale Namen
  - Lokale Namen
- Innerhalb von Funktionen referenzieren wir Namen in separaten lokalen Namensräumen
- Ausserhalb von Funktionen: Global = Lokal

Klassenvariablen vs. Instanzvariablen

- Klassenvariablen gehören zur Klasse
  - sie sind unabhängig von einzelnen Objekt-Instanzen
  - man referiert normalerweise auf sie mit `Klasse.variable`
  - man nennt sie auch statisch)

- Instanzvariablen
  - sind in jeder Klassen-Instanz unterschiedlich
  - werden in der Klassendefinition mit `self.variable` aufgerufen
  - können auch von „außen“ mit `instanz.variable` gelesen / geändert werden

Klassen- und Instanzvariablen

- Klassenvariablen befinden sich auch im Namensraum einer Instanz, sie haben für alle Instanzen den gleichen Wert

```
class MyClass:
  ....
```

```
...
instance = MyClass()
```

- Wird mit `instance.i` eine Variable `i` referenziert, wird zuerst nach der entsprechenden Instanz-Variable `i` gesucht
- Gibt es diese Variable nicht, wird gesucht ob es eine Klassen-Variable `MyClass.i` gibt, und ggf. die zurück gegeben

Ein einfaches Beispiel

```
class MyClass:
    i = 123
    def show_i(self):
        print(MyClass.i)
        print(self.i)
```

Klassenvariable

Instanzvariable

```
>>> k = MyClass()
>>> k.show_i()
123
123
>>> print(k.i)
123
```

```
>>> k.i = 321
>>> k.show_i()
123
321
```

```
>>> MyClass.i = 17
>>> k.show_i()
17
321
```


Vererbung

- In objektorientierten Sprachen kann man (normalerweise) Klassen von anderen Klassen ableiten
- Die abgeleitete Klasse *erbt* Variablen und Methoden von der Basisklasse
  - Somit unterstützen die abgeleiteten Klassen die gleichen Methoden/Variablen wie die Basisklassen
  - und können überall dort benutzt werden, wo die Basisklasse benutzt werden kann
- So lange die abgeleiteten Klassen Methoden nicht *überschreiben*, verhalten sie sich in der abgeleiteten Klasse genauso wie in der Basisklasse

Vererbung: Beispiel 1

```
class Face:
  def smile(self):
    print(":-)")
  def kiss(self):
    print(":-*")

class BoyFace(Face):
  def pokeToungue(self):
    print(":-P")

class GirlFace(Face):
  def cry(self):
    print(":‘(")
```

```
>>> boy = BoyFace()
>>> girl = GirlFace()
>>> boy.pokeToungue()
:-P
>>> girl.cry()
:‘(
>>> boy.kiss()
:-*
>>> girl.smile()
:-)
```


Vererbung: Beispiel 2

```
class Person:
  def __init__(self, name):
    self.name = name


class FrenchGuy(Person):
  def sayHello(self):
    print("Bonjour "+ self.name)

class GermanGuy(Person):
  def sayHello(self):
    print("Hallo " + self.name)
```

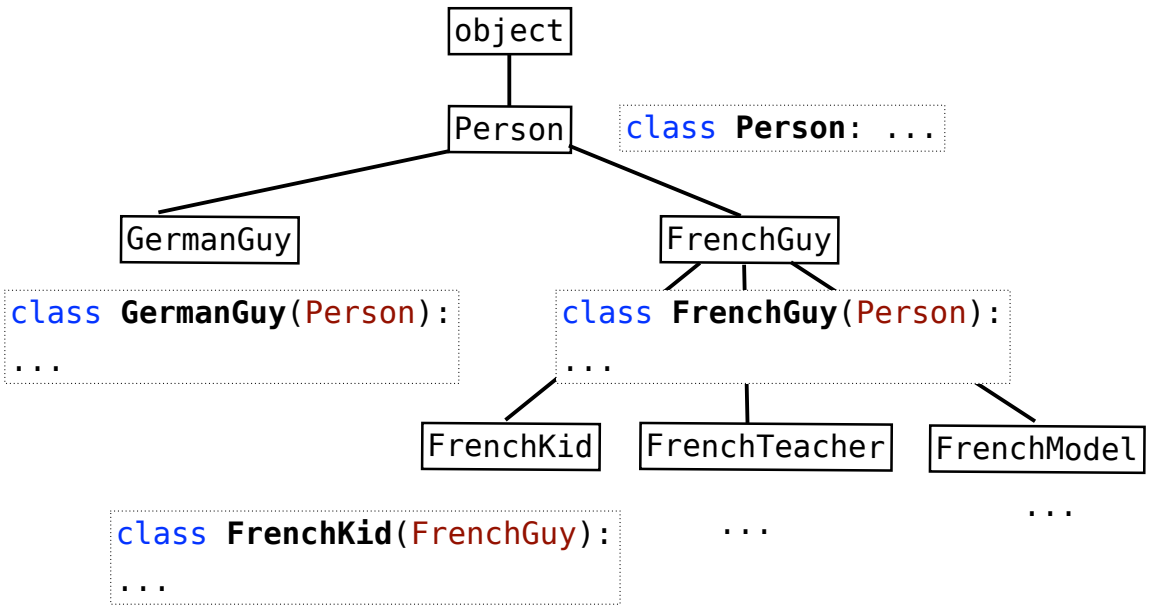
```
>>> g = GermanGuy('Stefan')
>>> g.sayHello()
Hallo Stefan
>>> f = FrenchGuy('Etienne')
>>> f.sayHello()
Bonjour Etienne
```

auch __init__ wird mit vererbt.

Vererbungshierarchien

- Alle Klassen haben automatisch eine Basisklasse (`object`) in Python, die vererbt auch Dinge
  - `object` vererbt eine Methode, die einen *Hash-Code* erzeugt -- das heißt, man darf selbst erzeugte Klassen standardmäßig in Mengen und Dictionaries benutzen
  - was sonst von `object` vererbt wird, sehen wir in späteren Vorlesungen
- Man kann Klassen-Hierarchien durch Vererbung aufbauen; an der Spitze steht immer `object`

Vererbungshierarchien



Vererbung: Methoden überschreiben

- Manchmal möchte man eine Basisklasse nicht nur neue Methoden hinzufügen, sondern existierende Methoden modifizieren (häufig: `__init__`)
- Man kann Methoden einfach überschreiben, indem man sie neu definiert
- Wenn man auf die gleichnamige Methode der Basisklasse zugreifen möchte, kann man die eingebaute Methode **`super`** benutzen:
  `super().methode(...)` tut das gleiche wie `BasisKlasse.methode(self,...)`

Methoden überschreiben: Beispiel 1

```
class Face:
  def smile(self):
    print(":-)")
  def kiss(self):
    print(":-*")

class VerryHappyFace(Face):
  def smile(self):
    print(":-D")
```

Methoden überschreiben: Beispiel 2

```
class Person:
  def __init__(self, name):
    self.name = name
  ...

class Employee(Person):
  def __init__(self, name, salary):
    super().__init__(name)
    self.salary = salary
  ...
```


Abstrakte Klassen

- Ein gängiges Konzept aus der objektorientierten Programmierung sind abstrakte Klassen
- Abstrakte Klassen enthalten nicht implementierte Methoden (ohne Körper) und müssen abgeleitet werden, um sinnvoll verwendet zu werden
- Python kennt keine abstrakten Klassen, man kann sie jedoch einfach simulieren: die Basisklasse definiert eine „Platzhalter“-Methode, die gar nichts tut, oder einen Fehler meldet

Pythons Schlüsselwort für „nichts tun“ ist **pass**

Eine einfache „abstrakte“ Klasse

```
class AnsweringMachine:
  def answerCall(self):
    ...
    self.sayText()
    self.beep()
    ....
  def sayText(self):
    pass # alternativ: raise NotImplementedError

class MyAnsweringMachine(AnsweringMachine):
  def sayText(self):
    return("Ich rufe zurück, hinterlass eine Nachricht!")
```


Mehrfachvererbung

http://www.python.org/download/releases/2.3/mro/

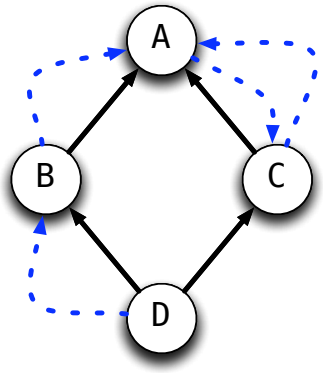
- Python unterstützt Mehrfachvererbung: Eine Klasse kann von mehreren Basisklassen abgeleitet werden:

```
class DerivedClass(Base1, ..., Basen):
  ...
```

- Resolution bei Attribut-Zugriff: depth-first, left-to-right
  - Zuerst wird das Attribut in $Base_1$ gesucht
  - dann rekursiv in den Basisklassen von $Base_1$
  - dann in $Base_2$, usw.
  - zumindest so ungefähr (siehe Python-Doku), in Wirklichkeit ist es viel komplizierter
  - wenn die einzelnen Klassen nicht voneinander erben, stimmt es so

Mehrfachvererbung

```
class A:
  def f(self): return 1
class B(A):
  def f(self): return 2
class C(A):
  pass
class D(B, C):
  pass
d = D()
d.f() ⇒  ?
```



Namenskonflikte & Konvention

- Daten-Attribute überschreiben Methoden-Attribute mit gleichem Namen.
- Gängige Konvention zur Vermeidung von Konflikten: Daten-Attribute beginnen mit einem Unterstrich: `_foo`.

Private Variablen (Name Mangling)

- In Python gibt es keine „echten“ privaten Variablen bzw. Methoden, die nur innerhalb der Klasse zugreifbar sind.
- Um Namenskonflikte zu vermeiden, können Namen „verstümmelt“ werden: Bezeichner der Form `__foo` werden automatisch durch `__klassenname_foo` ersetzt.

Bürger erster Klasse (*first class citizens*)

- In Python sind Klassen „Bürger erster Klasse,“ d.h., sie unterliegen nur den Einschränkungen, die für wirklich alles in Python gelten
- Man kann beispielsweise Klassen auch innerhalb von Funktionen definieren
- oder Klassen selbst (nicht nur ihre Instanzen) als Argument in Funktionsaufrufen verwenden

Hooks

- In den letzten Vorlesungen wurden einige Operatoren vorgestellt: +, -, ...
- Streng genommen gibt es in Python aber gar keine Operatoren, sondern nur Operationen:
  - Der „+“-Operator ruft beispielsweise intern die __add__ Methode des ersten Operanden auf
  - Diese speziellen Methoden („hooks“) kann man selbst definieren (bzw. überschreiben), um damit die Funktionalität zu ändern oder zu erweitern

Rationale Zahlen mit Operatoren

```
class Rat:
  def __init__(self, num, den):
    self.num = num
    self.den = den
  def __mul__(self, other):
    num = self.num * other.num
    den = self.den * other.den
    return Rat(num, den)
  def __repr__(self):
    return "Rat("+ str(self.num)+","+ str(self.den) + ")"
  def __str__(self):
    return str(self.num) + "/" + str(self.den)
```

```
>>> r1 = Rat(1,2)
>>> r2 = Rat(3,4)
>>> r1 * r2
Rat(3, 8)
>>> print(r1 * r2)
3/8
```

Hooks

- Vergleichsoperatoren (Rückgabe: True / False):
  - `__eq__` `==`
  - `__ge__` `>=`
  - `__gt__` `>`
  - `__le__` `<=`
  - `__lt__` `<`
  - `__ne__` `!=`
- `__bool__` : gilt das Objekt als Wahr oder Falsch? (Gibt `True` oder `False` zurück)

Hooks

- Numerische Operationen:
  - `__add__, __iadd__` `+, +=`
  - `__truediv__, __itruediv__` `/, /=`
  - `__mul__, __imul__` `*, *=`
  - `__sub__, __isub__` `-, -=`
  - `__mod__, __imod__` `%, %=`
- Element-Zugriff für Sammeltypen:
  - `x.__getitem__(i)` `x[i]`

`dict` mit Defaultwert

```
class Defaultdict(dict):
  def __init__(self, default):
    self.default = default

  def __getitem__(self, key):
    if key in self:
      return super().__getitem__(key)
    else:
      return self.default
```

```
>>> d = Defaultdict(0)
>>> d[17]
0
>>> d[17] += 1
>>> d[17]
1
```


Zusammenfassung

- Mehr über Klassen & Objekte
- Vererbung
  - Hierarchien
  - Mehrfachvererbung
  - Attribute überschreiben
- Operatoren überladen mit Hooks