Informatik I

16. Objekt-orientierte Programmierung: Aggregierung, Properties, Operator-Überladung und magische Klassen

Bernhard Nebel

Albert-Ludwigs-Universität Freiburg

06. Dezember 2013

Informatik I

06. Dezember 2013 — 16. Objekt-orientierte Programmierung: Aggregierung, Properties, Operator-Überladung und magische Klassen

16.1 Aggregierung

16.2 Properties

16.3 Operator-Überladung

16.4 Der Zoo der magischen Methoden

16.1 Aggregierung

Zusammengesetzte Objekte

- Oft sind Objekte aus anderen Objekten zusammengesetzt.
- Methodenaufrufe an ein Objekt führen dann zu Methoden-Aufrufen der eingebetteten Objekt.
- Beispiel ein zusammengesetztes 2D-Objekt, das andere 2D-Objekte enthält, z.B. einen Kreis und ein Rechteck.

Die CompositeObject-Klasse (1)

- Jede Instanz ist ein 2D-Objekt und hat eine Position.
- Zusätzlich hat jede Instanz als Attribut eine Liste von 2D-Objekten.

newgeoclasses.py (1)

```
class CompositeObject(TwoDObject):

    def __init__(self, x=0, y=0, objs=()):
        super().__init__(x, y)
        self.objects = list(objs)

    def add(self, obj):
        self.objects.append(obj)

    def rem(self, obj):
        self.objects.remove(obj)
  ...
```


Die CompositeObject-Klasse (2)

- Die size_change- und move-Methoden müssen überschrieben werden.
- Wir wälzen dann das Verschieben des zusammengesetzten Objektes auf die Einzelobjekte ab!

newgeoclasses.py (2)

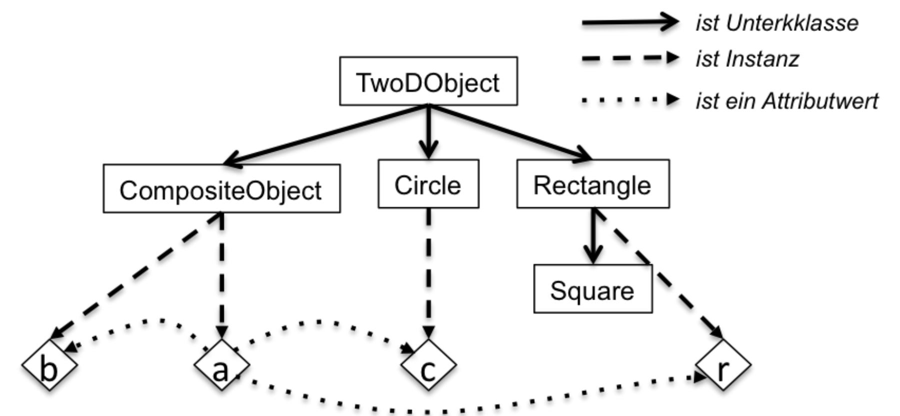
```
    ...
    def size_change(self, percent):
        for obj in self.objects:
            obj.size_change(percent)

    def move(self, xchange, ychange):
        super().move(xchange, ychange)
        for obj in self.objects:
            obj.move(xchange, ychange)
```


Die CompositeObject-Klasse (3)

Python-Interpreter

```
>>> c = Circle(1,2); r = Rectangle(10,10)
>>> a = CompositeObject(0,0,(r,c))
>>> a.size_change(200)
>>> r.area()
4.0
>>> a.move(40,40)
>>> a.position()
(40, 40)
>>> c.position()
(41, 42)
>>> b = CompositeObject(10,10)
>>> a.add(b)
>>> a.move(-10, -10)
>>> b.position()
(0, 0)
```


Vererbung und Komposition



16.2 Properties

Zugriff auf Attribute kontrollieren: Setters und Getters

- Häufig möchte man nach außen sichtbare Attribute „kontrollieren“, d.h. beim Setzen oder Abfragen bestimmte Dinge anstoßen.
- In *Java* deklariert man dazu (alle) Attribute als *private* und schreibt dann Getter- und Setter-Methoden. Damit kann man *nie* direkt auf die Attribute zugreifen.
- In Python sind Attribute im wesentlichen *public*. Wenn man später einmal Attribute „verstecken“ möchte, dann gibt es die Properties.
- Beispiel: Wir wollen die Positionsattribute verstecken, da wir bei einer Neuzuweisung in einem `CompositeObject` alle Objekte entsprechend verschieben wollen.

Die Basis-Klasse wird um Properties erweitert

properties.py (1)

```
class TwoDObject:
   def __init__(self, x=0, y=0):
        self._x = x
        self._y = y
        TwoDObject.counter +=1
    ...
    def setX(self, x):
        print("setX TDO:",x) # Debug-Ausgabe
        self._x = x

    def getX(self):
        return self._x

    x = property(getX, setX)
    ... # und für y
```


Properties – Was passiert?

- `getX` und `setX` sind zwei völlig normale Methoden.
- Die Zuweisung `x = property(getX, setX)` bewirkt, dass `x` ein Attribut wird, wobei bei lesendem Zugriff `getX` und bei schreibendem Zugriff `setX` aufgerufen wird (bei der Angabe von `None` ist der entsprechende Zugriff nicht möglich).

Python-Interpreter

```
>>> t = TwoDObject(10,20)
>>> t.x
10
>>> t.x = 33
setX TDO: 33
>>> t._x
33
```

Property überschreiben

- Jetzt wollen wir die Properties `x` und `y` in `CompositeObject` überschreiben (etwas umständlich).

properties.py

```
class CompositeObject(TwoDObject):
    ...
    def setX(self, newx):
        print("setX CO:", newx)
        for obj in self.objects:
            obj.x += (newx - self._x) # verschiebe
        self._x = newx

    # Ggfs. getX noch einmal definieren
    x=property(TwoDObject.getX,setX) # neue Property
```


Was passiert?

Python-Interpreter

```
>>> c = Circle(1,2); r = Rectangle(10,10)
>>> a = CompositeObject(0,0,(r,c))
    ...  # all das Zeugs vom letzten Beispiel

>>> print(a.position(), b.position(), c.position())
(30, 30) (0, 0) (31, 32)
>>> a.x = 100
setX CO: 100
setX TDO: 110
setX TDO: 101
setX CO: 70
>>> print(a.position(), b.position(), c.position())
(100, 30) (70, 0) (101, 32)
```


16.3 Operator-Überladung

Operator-Überladung

- Man sagt, ein Operator sei überladen (operator overloading), wenn dieser Operator je nach Kontext etwas anderes bedeutet (und macht).
- Die arithmetischen Operatoren sind traditionell in allen Programmiersprachen überladen. Sie funktionieren für alle numerischen Typen.
- In Python ist außerdem „+“ und „*“ für Strings überladen.
- Interessant wird es, wenn der Programmier selbst überladen darf!
- Die magische Methode `__add__` wird immer dann aufgerufen, wenn der „+“-Operator dort steht.

Eine Addition für 2D-Objekte: Rechtecke

overloading.py (1)

```
class Rectangle(TwoDObject):
        ...
  def __add__(self, other):
    return(Rectangle(self.x+other.x, self.y+other.y,
                     self._height+other._height,
                     self._width+other._width))
```

- Was fehlt hier?
- Was passiert, wenn `other` kein `Rectangle` ist?

Eine Addition für 2D-Objekte: Rechtecke

overloading.py

```
class Rectangle(TwoDObject):
           ...
  def __add__(self, other):
    if (isinstance(other,Rectangle)):
        return(Rectangle(self.x + other.x, self.y + other.y,
                         self._height + other._height,
                         self._width + other._width))
    else:
        raise TypeError("Cannot add Rectangle to " +
                   str(other.__class__.__name__))
```

- Entweder nach oben delegieren oder einen Typfehler erzeugen.

Eine Addition für 2D-Objekte: Quadrate

overloading.py

```
class Square(Rectangle):
          ...
  def __add__(self, other):
    if isinstance(other, Square):
      return(Square(self.x+other.x, self.y+other.y,
                    self._height+other._height))
    else:
      return(super().__add__(other))
```

- Hier können wir, falls es keine zwei Quadrate sind, alles nach oben delegieren.

Drucken der 2D-Objekte

- Wenn man Instanzen oder Klassen versucht zu drucken, so sieht das ziemlich hässlich aus:

Python-Interpreter

```
>>> c = Circle(1,2); c
<__main__.Circle object at 0x103dc8d90>
```

- Es gibt zwei magische Methoden `__repr__` und `__str__`, mit denen die Ausgabe gesteuert werden kann.
- `__repr__` soll die maschinenverstehbare Form erzeugen (einen String, der von `eval` verstanden wird und dann ein strukturähnliches Objekt erzeugt).
- `__str__` ist fürs schöne Ausdrucken zuständig. Wenn das nicht definiert ist, wird die `__repr__` eingesetzt.

Drucken von Circle- und CompositeObject-Instanzen

overloading.py (2)

```
class Circle(TwoDObject):
      ...
  def __repr__(self):
    return("Circle(x=%s, y=%s, radius=%s)" %
           (self.x, self.y, self.radius))

class CompositeObject(TwoDObject):
      ...
  def __repr__(self):
    str =("CompositeObject(x=%s, y=%s, objs=(" %
           (self.x, self.y))
    for obj in self.objects:
      str = str + "%s, " % repr(obj)
    return(str + "))")
```


Wie sieht ein Kreis aus?

Python-Interpreter

```
>>> c1 = Circle(1, 1, 1)
>>> r1 = Rectangle( 30, 40, 50)
>>> s1 = Square(2, 2, 10)
>>> a = CompositeObject(55,55,(c1,r1,s1))
>>> print(c1)
Circle(x=1, y=1, radius=1)
>>> print(a)
CompositeObject(x=55, y=55, objs=(Circle(x=1, y=1,
radius=1), Rectangle(x=30, y=40, height=50, width=1),
Square(x=2, y=2, side=10), ))
>>> print(eval(repr(a)))
CompositeObject(x=55, y=55, objs=(Circle(x=1, y=1,
radius=1), Rectangle(x=30, y=40, height=50, width=1),
Square(x=2, y=2, side=10), ))
```


16.4 Der Zoo der magischen Methoden

- Allgemeine magische Methoden
- Numerische magische Methoden
- Magische Container-Methoden

Magische Methoden

- Methoden wie `__init__`, deren Namen mit zwei Unterstrichen beginnen und enden, bezeichnet man als *magisch*.
- Daneben gibt es noch eine Vielzahl an weiteren magischen Methoden, die z.B. verwendet werden, um Operatoren wie + und % für eigene Klassen zu definieren.
- Magische Methoden wie `__add__` sind nicht prinzipiell anders als andere Methoden; der Grund dafür, warum man beispielsweise mit `__add__` das Verhalten der Addition beeinflussen kann, liegt einfach darin, dass Python intern versucht, beim Addieren die Methode `__add__` aufzurufen.

Magische Methoden: Übersicht

Wir können uns nicht alle magischen Methoden im Detail anschauen, aber zumindest sollten wir einen guten Überblick bekommen können.
Es gibt drei Arten von magischen Methoden:

- Allgemeine Methoden: verantwortlich für Objekterzeugung, Ausgabe und ähnliche grundlegende Dinge.
- Numerische Methoden: verantwortlich für Addition, Bitshift und ähnliches
- Container-Methoden: verantwortlich für Indexzugriff, Slicing und ähnliches

Allgemeine magische Methoden

Die allgemeinen magischen Methoden werden weiter unterteilt:

- Konstruktion und Destruktion: `__init__`, `__new__`, `__del__`
- Vergleich und Hashing: `__eq__`, `__ne__`, `__ge__`, `__gt__`, `__le__`, `__lt__`, `__hash__`, `__bool__`
- String-Konversion: `__str__`, `__repr__`, `__format__`
- Verwendung einer Instanz als Funktion: `__call__`
- Attributzugriff: `__getattribute__`, `__getattr__`, `__setattr__`, `__delattr__`
- Magische Attribute: `__dict__` (das dict der Attributnamen) und `__slots__` (um Attribute zu beschränken)

Konstruktion und Destruktion

- `__init__` haben wir bereits behandelt.
- `__new__` ist im Wesentlichen für fortgeschrittene Anwendungen mit Nicht-Python-Klassen interessant und wird von uns übergangen.
- `__del__` wird aufgerufen, wenn das Objekt aus dem Speicher gelöscht wird, weil es über keinen Namen mehr erreichbar ist: Destruktor. Sollte aber nicht benutzt werden, um ein Objekt auf der Programmierungsebene „abzumelden" (z.B. Konto schließen), da nicht direkt vorhersehbar ist, wann `__del__` aufgerufen wird.

Vergleich und Hashing: `__eq__`, `__ne__`

- `obj.__eq__(other)`:
  Wird bei Tests `obj == other` aufgerufen. Damit könnten wir Gleichheit von 2D-Objekten implementieren!'
- `obj.__ne__(other)`:
  Wird bei Tests `obj != other` aufgerufen.

- Definiert man diese Methoden nicht, werden Objekte nur auf Identität verglichen, d.h. `x == y` gdw. `x is y`.
- Aufruf von `!=` gibt automatisch das Gegenteil vom Aufruf von `==` zurück, außer wenn `==` mit `NotImplemented` antwortet. Es reicht also, `obj.__eq__(other)` zu implementieren.

Vergleich und Hashing: `__ge__`, `__gt__`, `__le__`,`__lt__`

- `obj.__ge__(other)`:
  Wird bei Tests `obj >= other` aufgerufen.

  Bei Tests `other <= obj` wird die Methode ebenfalls verwendet, falls `other` über keine `__le__`-Methode verfügt.
- `obj.__gt__(other)`, `obj.__le__(other)`, `obj.__lt__(other)`:
  Wird analog für die Vergleiche `obj > other` bzw. `obj <= other` bzw. `obj < other` aufgerufen.

Vergleich und Hashing: `__hash__`

- `obj.__hash__(self)`:
  Liefert einen Hashwert für `obj` bei Verwendung in einem Dictionary. Wird von der Builtin-Funktion `hash` verwendet.
- Damit Hashing funktioniert, muss immer gelten:

  $$\texttt{x == y} \quad \Longrightarrow \quad \texttt{hash(x) == hash(y)}.$$

  Daher muss man in der Regel auch `__eq__` implementieren, wenn man `__hash__` implementiert.

Vergleich und Hashing: `__bool__`

- `obj.__bool__(self)`:
  Wird von `bool(obj)` und damit auch bei `if obj:` und `while obj:` aufgerufen. Sollte `True` zurückliefern, wenn das Objekt als ‚wahr' einzustufen ist, sonst `False`.
- Ist diese Methode nicht implementiert, dafür aber das später diskutierte `__len__`, dann wird genau dann `True` geliefert, wenn `__len__` einen von 0 verschiedenen Wert liefert.
- Ist weder diese Methode noch `__len__` implementiert, gilt das Objekt immer als wahr.

String-Konversion: `__str__` und `__repr__`

- `obj.__str__(self)`:
  Wird aufgerufen, um eine String-Darstellung von `obj` zu bekommen, z.B. bei `print(obj)`, `str(obj)` und `"%s" % obj`.

  `__str__` sollte eine menschenlesbare Darstellung erzeugen.
- `obj.__repr__(self)`:
  Wird aufgerufen, um eine String-Darstellung von `obj` zu bekommen, z.B. bei Angabe von `obj` im interaktiven Interpreter sowie bei `repr(obj)` und `"%r" % obj`.

  `__repr__` sollte eine möglichst exakte (für Computer geeignete) Darstellung erzeugen, idealerweise eine, die korrekte Python-Syntax wäre, um dieses Objekt zu erzeugen.

Attributzugriff: `__getattr__`, `__getattribut__` und `__setattr__`

- `obj.__getattr__(name)`:
  Wird aufgerufen, wenn für `obj.name` kein Attribut gefunden werden kann. Kann z.B. für die generelle Delegation an eingebettete Objekte genutzt werden. Soll entweder einen Wert liefern oder einen `AttributeError` erzeugen.
- `obj.__getattribute__(name)`:
  Wird bei *jedem* lesenden Zugriff auf `obj.name` aufgerufen. Falls in einer Klasse definiert, wird `__getattr__` ignoriert. Wichtig: Um innerhalb der Methode auf den Wert zuzugreifen, muss man entweder direkt auf `__dict__[name]` zugreifen oder die `__getattribute__`-Methode der Superklasse aufrufen.
- `obj.__setattr__(name, value)`:
  Wird bei *jedem* lesenden Zugriff auf `obj.name` aufgerufen. Das bei `__getattribute__` gesagte gilt entsprechend.

Numerische Methoden

- Bei Operatoren wie +, *, - oder / verhält sich Python wie folgt (am Beispiel +):
- Zunächst wird versucht, die Methode `__add__` des linken Operanden mit dem rechten Operanden als Argument aufzurufen.
- Wenn `__add__` mit dem Typ des rechten Operanden nichts anfangen kann, kann sie die spezielle Konstante `NotImplemented` zurückliefern. Dann wird versucht, die Methode `__radd__` des rechten Operanden mit dem linken Operanden als Argument aufzurufen.
- Wenn dies auch nicht funktioniert, schlägt die Operation fehl.

Magische Methoden für Grundrechenarten

Hier sehen wir die Zuordnung zwischen den Grundrechenarten und den Namen der zugehörigen magischen Methoden:

- +: `__add__` und `__radd__`
- -: `__sub__` und `__rsub__`
- *: `__mul__` und `__rmul__`
- /: `__truediv__` und `__rtruediv__`
- //: `__floordiv__` und `__rfloordiv__`
- %: `__mod__` und `__rmod__`
- unäres -: `__neg__` (`-obj` entspricht `obj.__neg__(self)`).

Magische Methoden für Boolesche Operatoren

Hier das gleiche für die Booleschen Operatoren:

- &: `__and__` und `__rand__`
- |: `__or__` und `__ror__`
- ^: `__xor__` und `__rxor__`
- «: `__lshift__` und `__rlshift__`
- »: `__rshift__` und `__rrshift__`
- ∼ (unär): `__invert__`

Magische Methoden für In-Place-Operationen

- Bei Klassen, deren Instanzen veränderlich sein sollen, wird man in der Regel zusätzlich zu Operatoren wie + auch Operatoren wie += unterstützen wollen.
- Dazu gibt es zu jeder magischen Methode für binäre Operatoren wie `__add__` auch eine magische Methode wie `__iadd__`, die das Objekt selbst modifizieren und `self` zurückliefern sollte. (Der Rückgabewert ist wichtig; die Gründe dafür sind etwas technisch.)
- Implementiert man `__add__`, aber nicht `__iadd__`, dann ist `x += y` äquivalent zu `x = x + y`.

Container-Methoden

Mit den Container-Methoden kann man Klassen implementieren, die sich wie `list` oder `dict` verhalten.
Die Container-Methoden im Einzelnen:

- `obj.__len__(self)`:
  Wird von `len(obj)` aufgerufen.
- `obj.__contains__(item)`:
  Wird von `item in obj` aufgerufen.
- `obj.__iter__(self)`:
  Wird von `for x in obj` aufgerufen

Ferner für Zugriffe mit der eckigen Klammer `obj[key]`:

- `obj.__getitem__(key)`
- `obj.__setitem__(key, value)`
- `obj.__delitem__(key)`