Teil VI

Objektorientierte Programmierung

Was ist ein Objekt?

Was ist ein Objekt?

- Ein Auto
- Eine Katze
- Ein Stuhl
- ...

Was ist ein Objekt?

- Ein Auto
- Eine Katze
- Ein Stuhl
- ...

- Ein Molekül
- Ein Stern
- Eine Galaxie
- ...

Was ist ein Objekt?

- Ein Auto
- Eine Katze
- Ein Stuhl
- ...

- Ein Molekül
- Ein Stern
- Eine Galaxie
- ...

- Alles, was ein reales Objekt repräsentiert
- Alles, was ein abstraktes Objekt repräsentiert
- Alles, was irgendwelche Eigenschaften besitzt

Wie programmiert man objektorientiert?

- Wir müssen unsere Denkweise ändern!
- Wir müssen unseren Blickpunkt ändern!
- Es ist keine rein technische Frage.

Wie programmiert man objektorientiert?

- Wir müssen unsere Denkweise ändern!
- Wir müssen unseren Blickpunkt ändern!
- Es ist keine rein technische Frage.

Besonderheiten in python?

- Wir verwenden schon die ganze Zeit objekorientierte Konzepte.
- Wir merken es nur nicht.
- In Python ist alles ein Objekt (sogar int).
- Woher weiß z.B. print, wie ein int als Text aussieht?

```
>>> a=4
>>> a.__str__()
'4'
```

Theorie: Klassen und Objekte

Klassen:
”Idee“ aller Objekte einer Art;
Beispiel: Auto

Theorie: Klassen und Objekte

Klassen:
”Idee“ aller Objekte einer Art;
Beispiel: Auto

Instanzen / Objekte:
Konkretes Ding;
Beispiel: ”Das rote Auto dort“

Theorie: Klassen und Objekte

Klasse: ”Idee“ aller Objekte einer Art; Bsp.: Auto

- Ähnliche wie Definition eines Typs
- Enthält spezifische Eigenschaften der zu erzeugenden Objekte
- Daten, die einmalig für die Gesamtheit an Objekten gespeichert werden (Klassenvariable / statische Variable)
- Daten, die für jedes (bzw. in jedem) Objekt gespeichert werden (Objektvariable)
- Operationen, die auf Objekten ausgeführt werden (Methoden)
- Repräsentiert eine Klasse von Dingen/Objekten

Theorie: Klassen und Objekte

Klasse: ”Idee“ aller Objekte einer Art; Bsp.: Auto

- Ähnliche wie Definition eines Typs
- Enthält spezifische Eigenschaften der zu erzeugenden Objekte
- Daten, die einmalig für die Gesamtheit an Objekten gespeichert werden (Klassenvariable / statische Variable)
- Daten, die für jedes (bzw. in jedem) Objekt gespeichert werden (Objektvariable)
- Operationen, die auf Objekten ausgeführt werden (Methoden)
- Repräsentiert eine Klasse von Dingen/Objekten

Instanz / Objekt: Konkretes Ding; Bsp.: ”Das rote Auto dort“

- Konkretes Objekt, das zu einer Klasse gehört
- Eine Klasse spezifiziert Eigenschaften, ein Objekt hat konkrete Werte für diese Eigenschaften.
- Zu einer Klasse können beliebig viele Objekte gehören.
- Jedes Objekt gehört genau zu einer Klasse (Ausnahme: Polymorphismus)

Grundstruktur einer Klasse

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

Typische Statements

- Funktionsdefinitionen $\Rightarrow$ methods
- Variablendefinitionen $\Rightarrow$ data attributes (data members, fields, ...)

Theorie: Konstruktor, Destruktor

Beim Erzeugen einer Instanz wird...

- Speicher allokiert
- eine Variable zum Referenzieren des Objekts erzeugt
- vielleicht manches initialisiert
- die Initialisierung mit einem Konstruktor (`__init__(self,...)`) durchgeführt

Theorie: Konstruktor, Destruktor

Beim Erzeugen einer Instanz wird...

- Speicher allokiert
- eine Variable zum Referenzieren des Objekts erzeugt
- vielleicht manches initialisiert
- die Initialisierung mit einem Konstruktor (`__init__(self,...)`) durchgeführt

Wenn ein Objekt nicht mehr gebraucht wird, ...

- wird Speicher freigegeben
- müssen einige Dinge aufgeräumt werden
- wird der Destruktor (`__del__(self,...)`) aufgerufen
- ABER: Es ist nicht garantiert, wann er aufgerufen wird

```
class Example:
  def __init__(self, ...):
    do_something
  def __del__(self, ...):
    do_something
```

Theorie: Verwendung eines Objektes

Verwenden eines Objekts

- Zugriff auf Methoden: `objekt.methode(...)`
- Beispiel: `liste.append('x')`
- Zugriff auf Variablen: `objekt.variable`
- Beispiel: `complex.real`

Theorie: Verwendung eines Objektes

Verwenden eines Objekts

- Zugriff auf Methoden: `objekt.methode(...)`
- Beispiel: `liste.append('x')`
- Zugriff auf Variablen: `objekt.variable`
- Beispiel: `complex.real`

Zugriff innerhalb der Objektmethoden

- Konkretes Objekt wird außerhalb der Klassendefinition erzeugt

$\Rightarrow$ Den Klassenmethoden ist der Name nicht bekannt

$\Rightarrow$ Alle Methoden bekommen zusätzlichen Formalparameter `self`

- `self` ist immer der erste Formalparameter, dieser wird beim Aufruf (Aktualparameter) weggelassen
- Innerhalb der Methoden Zugriff auf andere Methoden und Variablen mit `self.variable` bzw. `self.methode()`

Erstes Klassenbeispiel

```
class Student:
  "Einfache␣Beschreibung␣eines␣Studierenden"
  def __init__(self, name): # Konstruktor
    self.name = name

  def getName(self):          # Methode
    return self.name

person1 = Student("Alex")
person2 = Student("Michaela")
print "Hier␣kommt␣" + person1.getName()
```

Theorie: Methoden

- . . . implementieren das Verhalten von Objekten.
- . . . repräsentieren alle Arten von Veränderungen eines Objekts nach der Initialisierung.
- **Accessor methods**: geben Info über Zustand des Objekts
- **Mutator methods**: verändern Zustand des Objekts
  $\Rightarrow$ verändern (mind.) eine Objektvariable

Theorie: Objektübergreifende (statische) Daten

Statische Variablen (Klassenvariablen)

- Variablen pro Objekt speicherbar: Objektvariable
  Beispiel: Name in `Student`
  $\Rightarrow$ in Konstruktor definieren mit self
- Variablen pro Klasse speicherbar: Klassenvariable
  $\Rightarrow$ nur 1x pro Klasse existent
  Klassisches Beispiel: Counter

Theorie: Objektübergreifende (statische) Daten

Statische Variablen (Klassenvariablen)

- Variablen pro Objekt speicherbar: Objektvariable
  Beispiel: Name in `Student`
  $\Rightarrow$ in Konstruktor definieren mit self
- Variablen pro Klasse speicherbar: Klassenvariable
  $\Rightarrow$ nur 1x pro Klasse existent
  Klassisches Beispiel: Counter

Statische Methoden

- Methoden, die lediglich statische Variablen (Klassenvariablen) benutzen und unabhängig von konkreten Objekten verwendbar sein sollen
- Definition via Schlüsselwort `@staticmethod`

Beispiel: Student reloaded

```
class Student:
  "Einfache Beschreibung eines Studierenden"
  anzahl = 0                    # Klassenvariable
  def __init__(self, name):     # Konstruktor
    self.name = name            # Objektvariable
    Student.anzahl += 1
  def getName(self):            # Methode
    return self.name
  @staticmethod
  def getAnzahl():              # statische Methode
    return Student.anzahl

person1 = Student("Alex")
person2 = Student("Michaela")
print "Hier kommt " + person1.getName()
print "Anzahl Studis=", Student.getAnzahl()
```

Hands-On: Legen Sie eine Klasse für Autos an!

- Jedes Auto soll eine eigene Seriennummer und eine Farbe (z.B. “rot”) besitzen.
- Jedes Auto soll fahren können. Legen Sie hierzu eine Methode `drive()` an, die die Farbe und Seriennummer des Autos ausgibt.

Hands-On: Legen Sie eine Klasse für Autos an!

- Jedes Auto soll eine eigene Seriennummer und eine Farbe (z.B. “rot”) besitzen.
- Jedes Auto soll fahren können. Legen Sie hierzu eine Methode `drive()` an, die die Farbe und Seriennummer des Autos ausgibt.

```
class Auto:
  def __init__(self,farbe,nummer):
    self.nummer=nummer
    self.farbe=farbe
  def drive(self):
    print "Die Farbe meines Autos (Nummer %i) ist %s" \
    % (self.nummer, self.farbe)

...
c = Auto("rot", 23)
c.drive()
```

Prozeduraler Ansatz: Ofen

```
#!/usr/bin/python
def backe(temperature, mode):
    Anweisungsblock

ofentemperatur = 180
ofenmodus = 2
backe(ofentemperatur, ofenmodus)
```

Prozeduraler Ansatz: Ofen

```
#!/usr/bin/python
def backe(temperature, mode):
    Anweisungsblock

ofentemperatur = 180
ofenmodus = 2
backe(ofentemperatur, ofenmodus)
```

Bei einer ganzen Bäckerei?

```
ofentemperaturen = []
ofenmodi = []
ofentemperaturen.append(180)
ofenmodi.append(2)
...
for i in range(len(ofentemperaturen)):
  backe(ofentemperaturen[i], ofenmodi[i])
```

Objektorientierter Ansatz: Ofen

```
#!/usr/bin/python
class Ofen:
  def __init__(self, temperatur, modus):
    self.temperatur=temperatur
    self.modus=modus
  def backe(self):
    Anweisungsblock
meinOfen = Ofen(180,2)
meinOfen.backe()
```

Objektorientierter Ansatz: Ofen

```
#!/usr/bin/python
class Ofen:
  def __init__(self, temperatur, modus):
    self.temperatur=temperatur
    self.modus=modus
  def backe(self):
    Anweisungsblock
meinOfen = Ofen(180,2)
meinOfen.backe()
```

Und wieder die ganze Bäckerei

```
ofenliste = []
ofenliste.append(Ofen(180,2))
...
for ofen in ofenliste:
  ofen.backe()
```

Was sind die Unterschiede?

Prozeduraler Ansatz

- Prozeduren/Funktionen legen fest, wie etwas „produziert“wird
- Bei der Implementierung denkt man an durchzuführende Aktionen
- Um die Speicherung der Daten muss man sich selbst kümmern

Was sind die Unterschiede?

Prozeduraler Ansatz

- Prozeduren/Funktionen legen fest, wie etwas „produziert“wird
- Bei der Implementierung denkt man an durchzuführende Aktionen
- Um die Speicherung der Daten muss man sich selbst kümmern

Objektorientierter Ansatz

- Objekte spezifizieren „Dinge“(real oder abstrakt)
- Bei der Implementierung muss man sich das Ding und seine Eigenschaften vorhalten
- Sämtliche Daten sind im Objekt selbst gespeichert (und damit gekapselt!)

$\Rightarrow$ Wissen über das Was, aber nicht über das Wie

$\Rightarrow$ Modularisierung! (Größere) Unabhängigkeit versch. Code-Teile

- Wenn viele Daten/Variablen mit einer Aktion verbunden sind, lohnt sich der objektorientierte Ansatz besonders

Informationskapselung / Information Hiding

public

- per default erstmal alles (alle Variablen/Methoden)
- komfortabel (Zugriff von außen)
- oft gefährlich / unerwünscht

Informationskapselung / Information Hiding

public

- per default erstmal alles (alle Variablen/Methoden)
- komfortabel (Zugriff von außen)
- oft gefährlich / unerwünscht

private

- Zugriff von außen (d.h. außerhalb der Klasse) verboten

$\Rightarrow$ Variablen/Methoden außen nicht sichtbar

$\Rightarrow$ Variablen/Methoden außen nicht nutzbar

- Workaround in python: name mangling

$\Rightarrow$ Syntax: beginnend mit `__`, aber nicht endend mit `__` (`__privateVar`, `__privateMet()`)

- Achtung: indirekt zugreifbar (`._Klasse__privateVar`)

Information Hiding: Beispiel

Hands-On: Modifizieren Sie Ihr Auto!

- Jedes Auto soll eine eigene Seriennummer und eine Farbe (z.B. “rot”) besitzen.
- Nur das Auto selbst soll seine Seriennummer wissen.
- Die Farbe des Autos ist von außerhalb sichtbar.
- Jedes Auto soll fahren können. Legen Sie hierzu eine Methode `drive()` an, die die Farbe und Seriennummer des Autos ausgibt.

```
class Auto:
  def __init__(self,farbe,nummer):
    self.nummer=nummer
    self.farbe=farbe
  def drive(self):
    print "Die␣Farbe␣meines␣Autos␣(Nummer␣%i)␣ist␣%s" \
    % (self.nummer, self.farbe)
```

Information Hiding: Beispiel

Hands-On: Modifizieren Sie Ihr Auto!

- Jedes Auto soll eine eigene Seriennummer und eine Farbe (z.B. “rot”) besitzen.
- Nur das Auto selbst soll seine Seriennummer wissen.
- Die Farbe des Autos ist von außerhalb sichtbar.
- Jedes Auto soll fahren können. Legen Sie hierzu eine Methode `drive()` an, die die Farbe und Seriennummer des Autos ausgibt.

```
class Auto:
  def __init__(self,farbe,nummer):
    self.__nummer=nummer
    self.farbe=farbe
  def drive(self):
    print "Die␣Farbe␣meines␣Autos␣(Nummer␣%i)␣ist␣%s" \
    % (self.__nummer, self.farbe)
```

Public und Private: Wozu?

Verwende private, um

- ... Eigenschaften/Funktionen zu verstecken
  $\rightarrow$ Implementierdetails nicht wichtig für reine Objektnutzung
- ... Objekt nur zweckmäßig nutzen zu können
  $\rightarrow$ Grundsatz: Nur so viele Zugriffs-/Änderungsrechte an Objekten wie nötig!
  $\rightarrow$ bestmögliche Kontrolle über Objektzustand

Verwende public, um

- ... Zugriff auf das Objekt und seine Eigenschaften/Funktionen von außen zu erlauben

Public und Private: Beispiel

Variante 1:

```
class Auto:
  def __init__(self, \
  farbe,nummer):
    self.__nummer=nummer
    self.__farbe=farbe
  def drive(self):
    print self.__farbe, \
    self.__nummer
```

Variante 2:

```
class Auto:
  def __init__(self, \
  farbe,nummer):
    self.nummer=nummer
    self.farbe=farbe
  def drive(self):
    print self.farbe, \
    self.nummer
```

- Angenommen: `Auto` in großem Code-Framework benutzt (100 Klassen)
- Änderung: `farbe`/`__farbe` in `color`/`__color`.

Hands-On: Wie viele Programmstücke können bei Variante 1 bzw. 2 durch diese Modifikation beeinträchtigt werden?

Public und Private: Beispiel

Variante 1:

```
class Auto:
  def __init__(self, \
  farbe,nummer):
    self.__nummer=nummer
    self.__farbe=farbe
  def drive(self):
    print self.__farbe, \
    self.__nummer
```

Variante 2:

```
class Auto:
  def __init__(self, \
  farbe,nummer):
    self.nummer=nummer
    self.farbe=farbe
  def drive(self):
    print self.farbe, \
    self.nummer
```

- Angenommen: `Auto` in großem Code-Framework benutzt (100 Klassen)
- Änderung: `farbe`/`__farbe` in `color`/`__color`.

Hands-On: Wie viele Programmstücke können bei Variante 1 bzw. 2 durch diese Modifikation beeinträchtigt werden?

Variante 1: kein Programmstück (da `__color` private)

Variante 2: potentiell alle Programmstücke, die bisher auf `farbe` zugreifen

Spezielle Variablen und Methoden

- `__get__(value)`: Erlaubt Lesezugriff mit `[]`
- `__set__(i, value)`: Erlaubt Schreibzugriff mit `[]`
- `__add__(value)`: Erlaubt addition mit `+`
- `__str__()`: Wird von `print` verwendet
- `__call__()`: Erlaubt Aufruf mit `objekt()`
- `__gt__()`, `__lt__()`: Vergleich mit `>`,`<`
- ... und viele, viele mehr!

Spezielle Variablen und Methoden - Beispiel

```
class MyString:
  "Own implementation of a string with comparison"
  def __init__(self,string):
    self.__string = string
  def getLen(self):
    return len(self.__string)

  def __gt__(self,other):
    return self.getLen() > other.getLen()
  def __lt__(self,other):
    return self.getLen() < other.getLen()
  def __str__(self):
    return self.__string
...
  s1 = MyString("Here I am")
  s2 = MyString("This is me")
  if s1 < s2:
    print s1
```