Übersicht

Eclipse mit PyDev

Objektorientierung

Beispielaufgaben

ECLIPSE MIT PYDEV

Eclipse

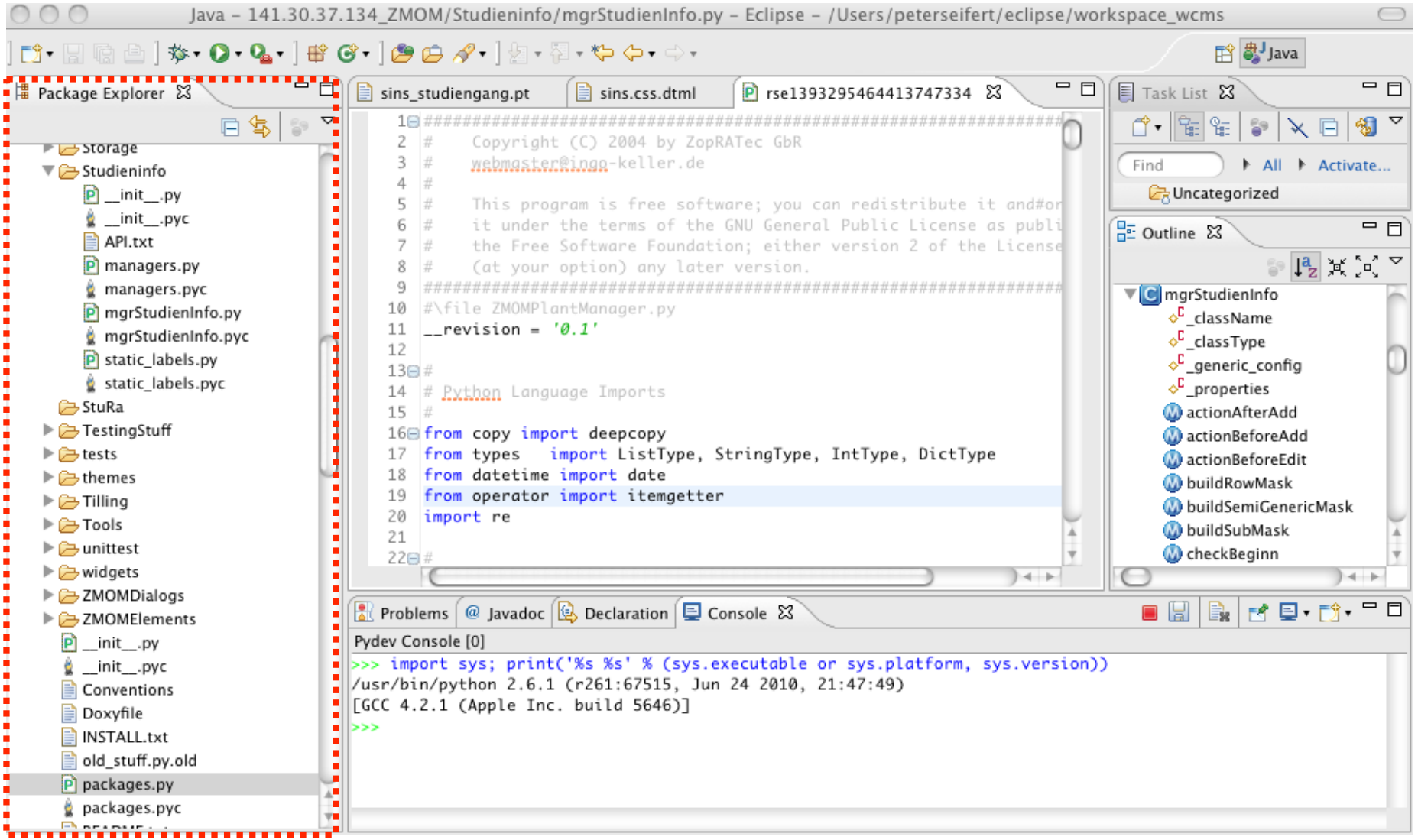
- Open Source Community
- Open Development Plattform bestehend aus erweiterbaren Frameworks, Werkzeugen und Umgebungen für Erstellung, Deployment und Management von Software über den gesamten Lebenszyklus hinweg
- Eclipse Foundation
  - not-for-profit mitgliedsgestützte Korporation
  - Hostet und koordiniert die Eclipse Projekte
  - Hilft bei der Entwicklung der open source community
  - Hilft bei der Erstellung eines Ecosystems verschiedenster Produkte und Services

PyDev

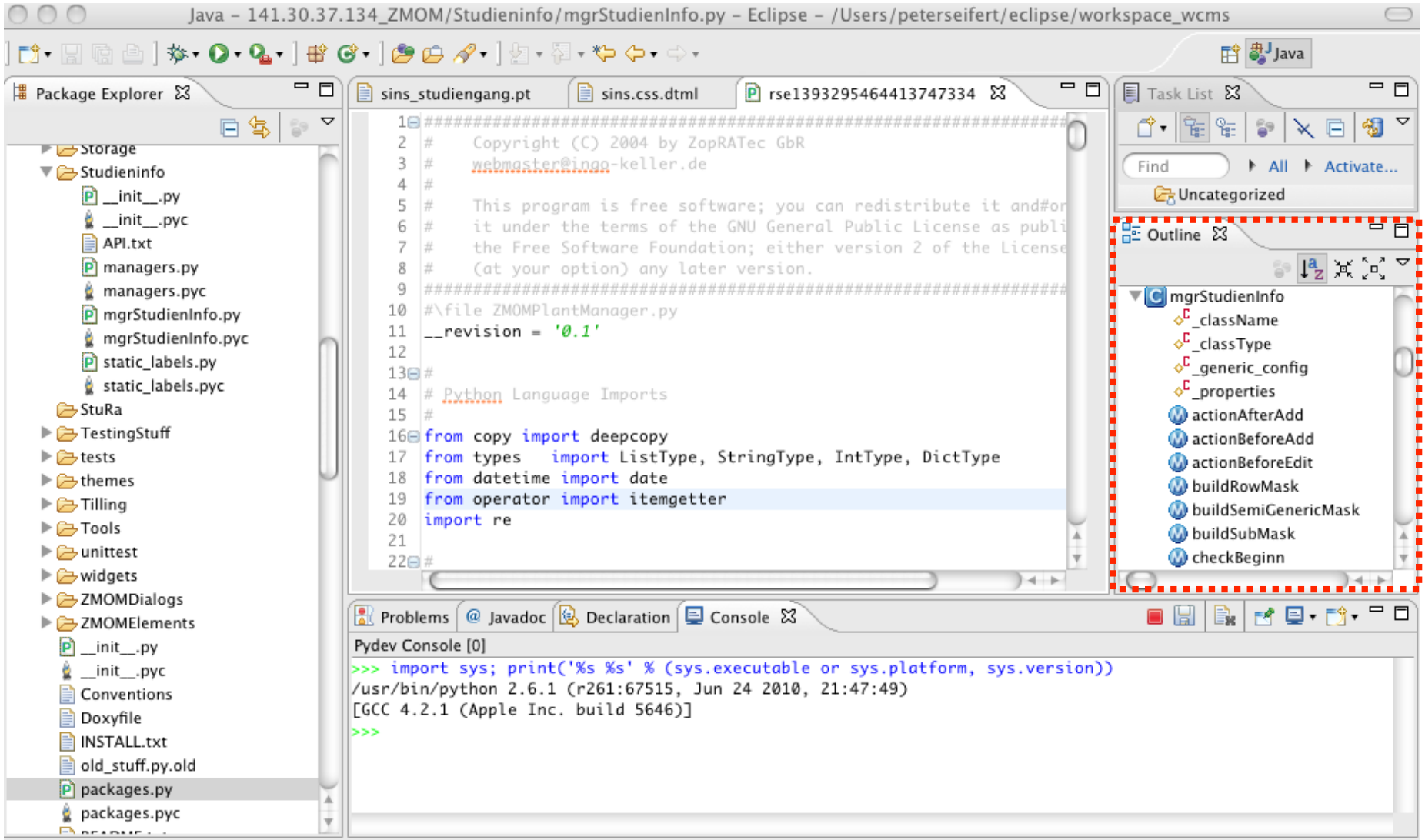
- PyDev is a **Python IDE** for **Eclipse**, which may be used in **Python**, **Jython** and **IronPython** development

- Features
  - Django integration
  - Code completion with auto import
  - Syntax highlighting
  - Code analysis
  - Go to definition
  - Refactoring
  - Mark occurrences
  - (Remote) Debugger
  - Tokens browser
  - Interactive console
  - Unittest integration

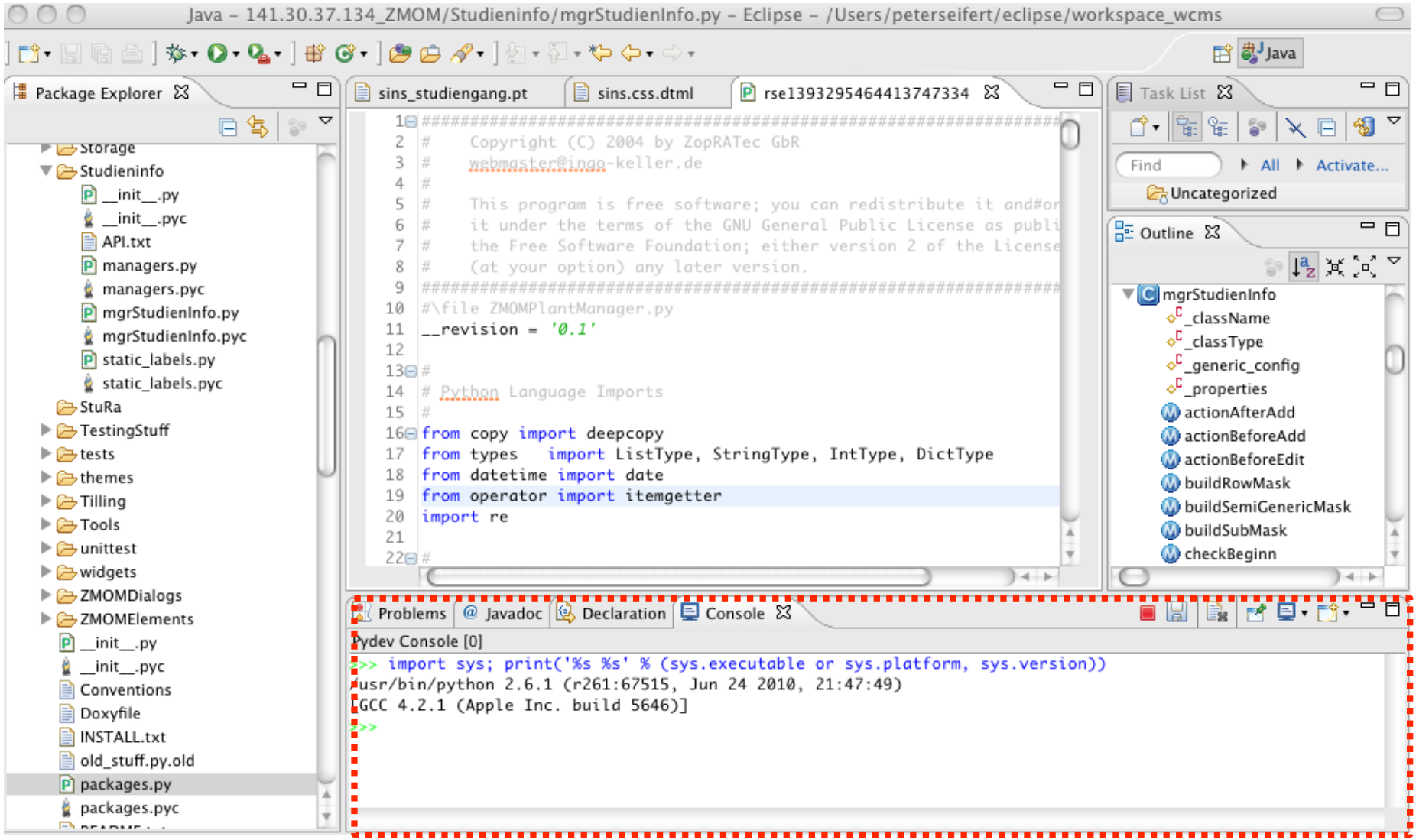
Oberfläche - Navigation



Oberfläche - Outline



Oberfläche - Konsole



Coding Conventions

- Use 4-space indentation, and no tabs
- Wrap lines so that they don’t exceed 79 characters
- Use blank lines to separate functions and classes, and larger blocks of code inside functions
- Put comments on a line of their own
- Use docstrings
- Use spaces around operators and after commas, but not directly inside bracketing constructs
  - `a = f(1, 2) + g(3, 4)`
- Name your classes and functions consistently
  - Klasse Test
  - Methode test_something
- Always use self as the name for the first method argument

OBJEKTORIENTIERUNG

Begriffsklärung

- Kapselung von Daten und Funktionen auf diesen Daten in einem Objekt
- Klasse als Schablone für Objekte
- Objekte kommunizieren miteinander (über Methoden)
- Objekte (und damit Klassen) haben:
  - Attribute (mit eigenem Datentyp)
  - Methoden (Parameter und Rückgabewert)
- Vererbung zur Übernahme von Methoden und Attributen aus bereits bestehenden Klassen
- Erster Schritt: Objektorientierte Analyse

- Spezialfall Python: keine strikte Kapselung

Klassen

- Keyword „class“
- Einrückung wie gewohnt
- Zugriff auf klasseneigene Variablen und Methoden innerhalb von Methoden mit „self“ (Konvention)
- Beispiel mit Code Konventionen

```
class Testclass:
    """This is an example"""
    a = 1
    b = 2

    def sum(self):
        """Example sum method adding a and b"""
        # add a and b and return it
        return self.a + self.b
```

Objekte

- Instanzen von Klassen
- Erzeugung zur Programmlaufzeit
- Mehrfache Erzeugung möglich
- Objektidentität

```
>>> ob1 = Testclass()
>>> ob2 = Testclass()
>>> ob1 == ob2
False
>>> ob1.sum()
3
```

Methoden

- Parameter zur Übergabe von Daten
- Erster Parameter ist das Objekt
- Default-Werte (hintere Parameter)
- Funktionsparameter sind Referenzen, das nützt aber nur bei mutablen Datentypen (List, Dict, eigene Klassen, etc.)

```
def something(self, a, b):
    pass
def something(self, a, b = 2):
    pass

>>> ob.something(1)
>>> ob.something(1,3)
>>> ob.something(a = 7, b = 8)
>>> ob.something(7, b = 8)
```

Alles ist ein Objekt

- Klassen sind Type-Objekte (von der Klasse Type)
  - die wiederum die Klasse Type haben
  - ''.__class__.__class__
- 1 ist ein Objekt vom Typ int
- ''.upper ist ein Objekt vom Typ function
- dir ist ein Objekt vom Typ built-in function

Alles ist ein Objekt

- Function-Typ-Objekte haben eine Funktion __call__ (vom Typ Method-Wrapper), die den Aufruf der Funktion repräsentiert
  - ''.upper() löst also eigentlich ''.upper().__call__() aus
  - auch ''.upper.__call__ hat wieder eine Funktion __call__ hat wieder eine Funktion __call__ ...
- built-in Funktionen sind immer verfügbar
  - z.B. callable(''.upper) -> True, weil upper eine Funktion ist
- Funktionen sind Objekte und können daher gepickled werden
  - Laden des pickle und ausführen der Funktion möglich

Klassen untersuchen

- Inspection in der Kommandozeile mittels dir, help und type
  - dir([]) Liste aller Attribute und Methoden
  - type([]) Klassenobjekt des Objekts (__class__)
  - help([]) Listing aller doc-Strings aller Attribute/Methoden

```
>>> help([])
Help on list object:
class list(object)
 |  list() -> new list
 |  list(sequence) -> new list initialized from sequence's items
 |  Methods defined here:
 |  __add__(...)
 |      x.__add__(y) <==> x+y
 |  __contains__(...)
 |      x.__contains__(y) <==> y in x
...
>>> [].__doc__
```

Konstruktor

- Funktion die nach der Erstellung des Objekts aufgerufen wird
- Wertübergabe zum Zeitpunkt der Erzeugung möglich

```
Class Testclass:
    """This is an example class with constructor"""
    def __init__(self, value1, value2):
        """set a and b values"""
        self.a = value1
        self.b = value2

    def sum(self):
        """example sum method adding a and b"""
        return self.a + self.b

>>> ob = Testclass(4,5)
>>> ob.sum()
```

Private Methoden

- Können nicht direkt aufgerufen werden
- Gekennzeichnet durch __xyz__
- Werden durch Python-Operatoren ausgeführt

```
>>> a = 1
>>> b = 2
>>> dir(a)

>>> a + b
3
```

- int Type
  - __add__ a + b
  - __cmp__ a == b
- List Type
  - __add__ l + r
  - __getitem__ l[2]

Hinweise zur Kapselung

- Attribute und Methoden sind immer von aussen erreichbar
- Konvention: _xyz kennzeichnet nicht-öffentliche Attribute
  - Zugriff möglich, aber nicht sicher (Invarianz)
  - Klassen sollten setter/getter zur Verfügung stellen
- Implementierung: __xyz erzeugt private Attribute (Name Mangling)
  - Zugriff trotzdem möglich
  - Dient der Eindeutigkeit bei Vererbung

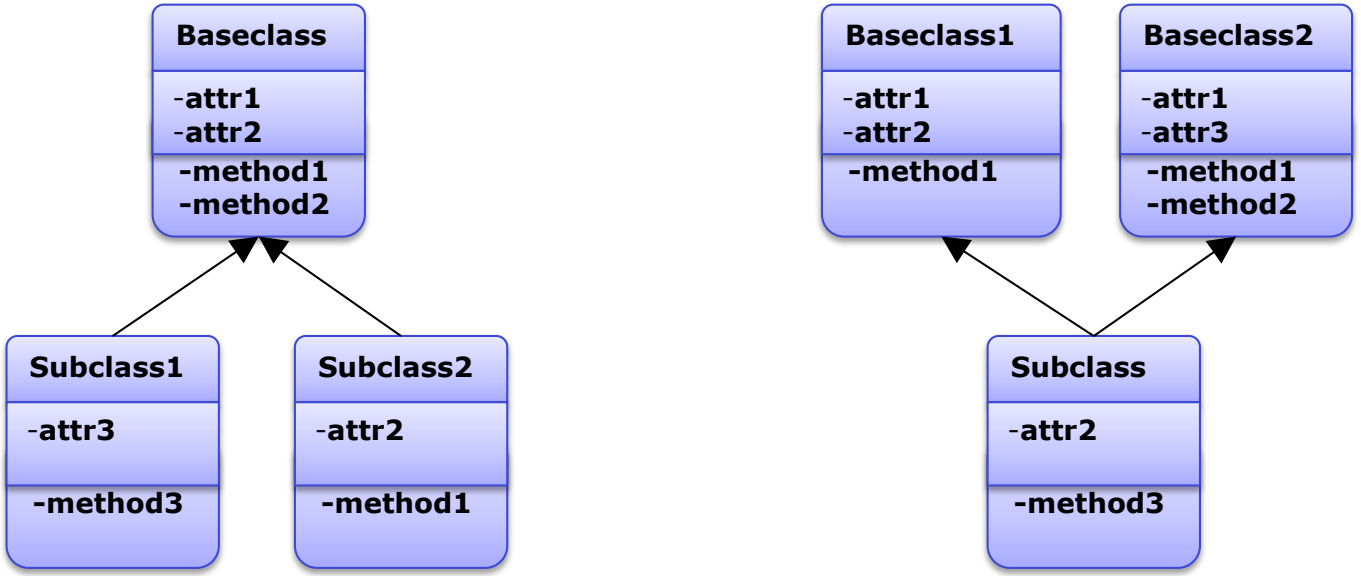
Referenzen und Garbage Collection

- Objekte sind solange vorhanden, wie eine Referenz auf sie existiert
- Ohne Referenz existieren sie noch, sind aber nicht zugreifbar
- Garbage Collector entfernt sie zu beliebigem Zeitpunkt

```
>>> l = []
>>> m = l
>>> l.append(1)
>>> l
[1]
>>> m
[1]
>>> func = l.append
>>> func
<built-in method append of list object at 0x1004d1758>
>>> func(3)
>>> m
[1, 3]
```

Vererbung

- Übernahme aller Methoden / Attribute der Basisklasse
- Überlagerung (Overriding) möglich
- Mehrfachvererbung in Python erlaubt
- Builtin Methoden isinstance und issubclass zur Prüfung



TECHNISCHE UNIVERSITÄT DRESDEN

Beispiel Vererbung I

```
class Baseclass:
    attr1 = 'Base_attr1'
    attr2 = 'Base_attr2'
    def method1(self):
        return 'call base_method1'
    def method2(self):
        return 'call base_method2' + self.attr2

Class Subclass2(Baseclass):
    attr2 = 'Sub_attr2'
    def method1(self):
        return 'call sub_method1'
```

Beispiel Vererbung II

```
from Baseclass import Baseclass
from Subclass2 import Subclass2
ob1 = Baseclass()
ob2 = Subclass2()
ob1.method1()
ob1.method2()

ob2.method1()
ob2.method2()

isinstance(ob2, Baseclass)
issubclass(type(ob2), Baseclass)
```

Beispiel Mehrfachvererbung I

```
class Baseclass1:
    attr1 = 'Base1_attr1'
    attr2 = 'Base1_attr2'
    def method1(self):
        return 'call base1_method1‘

class Baseclass2:
    attr1 = 'Base2_attr1'
    attr3 = 'Base2_attr3'
    def method1(self):
        return 'call base2_method1'
    def method2(self):
        return 'call base_method2' + self.attr2

Class Subclass(Baseclass1, Baseclass2):
    attr2 = 'Sub_attr2'
    def method3(self):
        return 'call sub_method3‘ + self.method1() + self.method2()
```

Beispiel Mehrfachvererbung II

```
from Subclass import Subclass
from Baseclass1 import Baseclass1
ob = Subclass()
ob.method1()
ob.method2()
ob.method3()
Baseclass2.method1(ob)
ob2.method2()

isinstance(ob2, Baseclass)
issubclass(type(ob2), Baseclass)
```

Überladen Privater Methoden

- Beispiel: Eigener Integer-Typ mit neuer Addition
- a + b -> a.__add__(b)

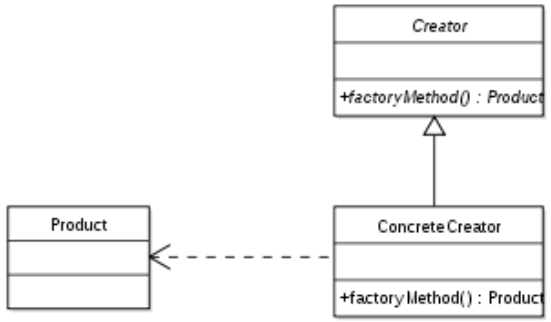
```
>>> class MyInt(int):
...     def __add__(self, b):
...         return b + self + 2
...
>>> a = 1
>>> b = MyInt(1)
>>> a + b
2
>>> b + a
4
```

Herangehensweise

- Objektorientierte Analyse
  - Identifikation von Klassen
  - Identifikation von Attributen
  - Identifikation von Beziehungen zwischen Klassen
  - Identifikation von Methoden
- Objektorientiertes Design
  - Erstellung von Vererbungshierarchien
  - Anwendung von Mustern (Pattern)
  - Anpassung an technische Erfordernisse
- Objektorientierte Programmierung
  - Umsetzung in einer OO-Sprache

Einfaches Muster: Factory method

- Software Design Patterns zur Strukturierung und Standardisierung der Herangehensweise an Problemklassen
- Erzeugung von komplexen Objekten mit standardisierter Erzeugerfunktion



- Literatur: GoF-Book (*Design Patterns: Elements of Reusable Object-Oriented Software*)

Beispiel Weide

Auf einer Weide stehen Kühe mit schwarzen Flecken herum. Die Weide zählt die Kühe selbständig. Die Kühe wissen, auf welcher Weide sie stehen. Kühe können grasen. Wenn eine Kuh zum 4. Mal grast, macht sie „Muh!“. Es gibt einzelne Kühe mit braunen Flecken, diese machen schon nach 3 Mal grasen „Muh!“. Wird eine braune Kuh gepiekt, macht sie „Muh!“.

Beispiel Weide OOA

- Weide
  - Anzahl Kühe
  - (Kuh hinzufügen)

- Kuh
  - Farbe (schwarz)
  - macht Muh
  - grast
  - kennt Weide

- Braune Kuh
  - Farbe (braun)
  - macht Muh
  - grast anders
  - wird gepiekt