



Zwei Arten von Attributen



- Die Daten, die von einem Objekt gespeichert werden und keine Methoden sind, heißen Attribute. Es gibt zwei Arten:
 - **Instanzzattribute (= Instanzvariablen):**
Variable, die einer bestimmten Instanz einer Klasse gehört. Jede Instanz kann ihren eigenen Wert für diese Variable haben. Dies ist die gebräuchlichste Art von Attributen.
 - **Class attributes (=Klassenvariablen):**
Gehört einer Klasse.
Für alle Instanzen dieser Klasse hat dieses Attribut den gleichen Wert. In manchen Programmiersprachen als "*static*" bezeichnet.
Nützlich für Konstanten oder als Counter für die Anzahl der Instanzen, die bereits erstellt wurden.



Klassenvariablen



- Bisher: Variablen waren Instanzvariablen, d.h., jede Instanz hat eigene "Kopie"
- **Klassenvariable** := Variablen, die innerhalb der Klasse genau 1x existieren
 - Alle Instanzen einer Klasse haben eine Referenz auf das gemeinsame Klassenattribut,
 - wenn eine Instanz es verändert, so wird der Wert für alle Instanzen verändert.
- In Python: definiere Klassenvariable außerhalb einer Methode
- Notation zum Zugriff: `self.__class__.name`

```
class sample:
    x = 23
    def increment(self):
        self.__class__.x += 1
```

```
>>> a = sample()
>>> a.increment()
>>> a.__class__.x
24
```



Introspektion



- Was tun, wenn Sie den Namen des Attributs oder der Methode einer Klasse nicht kennen, aber trotzdem auf das Element zur Laufzeit zugreifen wollen...
- Lösung: **Introspektion**
 - Methoden, um alle Attribute (Inst.vars und Methoden) einer Klasse aufzulisten
 - Zu einer Zeichenkette, die den Namen des Attributs oder der Methode beinhaltet, eine Referenz zu bekommen (die man verwenden kann)
 - Theoretisch: man könnte sogar zur Laufzeit Methoden hinzufügen
- Im folgenden nur 2 (von vielen!) Möglichkeiten der Introspektion



getattr(object_instance, string)



```
>>> f = student("Bob Smith", 23)
>>> getattr(f, "full_name")
"Bob Smith"

>>> getattr(f, "get_age")
<method get_age of class studentClass at 010B3C2>

>>> getattr(f, "get_age")() #Das können wir aufrufen.
23

>>> getattr(f, "get_birthday")
# Verursacht AttributeError - No method exists.
```

```
class student(object):
    def __init__( self,
                  name = "",
                  age = 0 ):
        self.name = name
        self.age = age
```

hasattr(object_instance, string)

```
>>> f = student("Bob Smith", 23)

>>> hasattr(f, "full_name")
True

>>> hasattr(f, "get_age")
True

>>> hasattr(f, "get_birthday")
False
```

Prof. Dr. G. Zachmann Informatik 1 - WS 05/06 Einführung in Python, Teil 2 71

Identitätsfindung

- Ein Objekt hat mehrere "Identitäten"
- Gleichheit mit anderen Objekten: `x == y`
 - Liefert True oder False (kann in der Klasse umdefiniert werden!)
 - Überprüft **Inhalt** (= Instanzvariable) auf Gleichheit
- Eindeutige ID: `id(x)`
 - Liefert eine eindeutige Zahl für jedes Objekt zu einem best. Zeitpunkt
- "is a"-Beziehung ("Instanz von")
`isinstance(x, (ClassName1, ClassName2, ...))`
 - Liefert True oder False (liefert True auch, falls x Instanz einer Unterklasse)

Prof. Dr. G. Zachmann Informatik 1 - WS 05/06 Einführung in Python, Teil 2 72



Anwendungsbeispiel

- z.B. um verschiedene Aktionen innerhalb einer Methode durchzuführen, abhängig vom Typ des tatsächlichen Parameters

```
class MyClass( object ):  
    def __init__( self, other ):  
        if isinstance( other, MyClass ):  
            ... # make a copy  
        elif isinstance( other, (int, float) ):  
            inst_var = other # create inst.var  
        else:  
            inst_var = 0 # default
```



Kein Bedarf für Freigaben

- Objekte braucht man nicht zu löschen oder freizugeben
- Python hat eine automatische *Garbage Collection* (Speicherbereinigung)
- Funktioniert mit *Reference Counting* (im 2. Semester mehr)
- Python ermittelt automatisch, wann alle Referenzen auf ein Objekt verschwunden sind und gibt dann diesen Speicherbereich frei
- Funktioniert im Allgemeinen gut, *wenige memory leaks*



Dokumentation



- Häufige Haltung: Source-Code sei die beste Dokumentation
 - "UTSL" ("use the source, luke")
- Aber:
 - ist viel zu **umfangreich** für einen schnellen Überblick
 - unterstützt das **Navigieren** zu gesuchten Informationen nur wenig
 - gibt keine Auskunft, welche **Annahmen / Voraussetzungen** einzelne Systemteile über das Verhalten anderer Teile machen (Schnittstellen)
 - gibt keine Auskunft, warum gerade diese **Lösung** gewählt wurde (warnt also nicht vor subtilen Fallen)
 - ...
- Deswegen: Korrekte (d.h. auch aktuelle) und hilfreiche **Dokumentation** ist extrem wichtig für die Entwicklung und Wartung eines Programms!



In Python integrierte Dokumentation



- In Python ist Doku fester Bestandteil der Sprache!
 - Geht noch weiter als in Java

```
def f():  
    """  
    blub  
    bla  
    """  
    ...  
help(f)
```

Ausgabe

```
Help on function f in module __main__:  
  
f()  
    blub  
    bla
```

- Diese Kommentare heißen **Docstrings** in Python



Vollständige Boilerplate für Docstrings



```
"""
Documentation for this module.

More details.
"""

def func():
    """ Documentation for a function.

        More details.
    """
    . . .

class MyClass:
    """ Documentation for a class.

        More details.
    """

    def __init__(self):
        """ Docu for the constructor. """
        . . .

    def method(self):
        """ Documentation for a method. """
        . . .
```



Automatische Generierung aus Source



- In allen Sprachen kann die Dokumentation zum großen Teil als Kommentar in den Source eingebettet werden
- Tools (wie z.B. Doxygen www.doxygen.org) erzeugen daraus automatisch sehr ansprechende und effiziente HTML-Seiten (oder LaTeX, oder ...)
 - Einzige Bedingung: Markup der Doku durch sog. *Tags*
- Vorteile:
 - Hoher Automatisierungsgrad (also Arbeitersparnis)
 - durch enge Kopplung an Implementierung leichter aktuell zu halten als separate Dokumente
 - Sowohl interne (Developer-) als auch externe (API-) Doku kann aus demselben Source generiert werden



Boilerplate für Dokumentation einer Funktion



```
def func( a ):  
    """! @brief Einzeilige Beschreibung  
  
    @param param1      Beschreibung von param1  
    @param param2      Beschreibung von param2  
  
    @return  
        Beschreibung des Return-Wertes.  
  
    Detaillierte Beschreibung ...  
  
    @pre  
        Annahmen, die die Funktion macht...  
        Dinge, Aufrufer unbedingt beachten muss...  
  
    @todo  
        Was noch getan werden muss  
  
    @bug  
        Bekannte Bugs dieser Funktion  
  
    @see  
        andere Methoden / Klassen  
    """
```



Dokumentation von Klassen, Moduln, ...



```
"""! @brief Documentation for a module.  
  
    More details.  
    """  
  
def func():  
    """! @brief Blub """  
    . . .  
  
class MyClass:  
    """! @brief Class Blub ...  
    """  
  
    def __init__(self):  
        """! @brief the constructor """  
        . . .  
  
    def method( self, p1 ):  
        """! @brief Documentation for a method  
            @param p1 blubber  
        """  
        . . .  
  
    ## A class variable.  
    classVar = 0;  
  
    ## @var _memVar  
    # a member variable
```

Doxygen-Dokumentation als HTML

The screenshot shows a web browser window displaying the HTML output of Doxygen documentation for a Python class named `pyexample.PyClass`. The page includes a navigation menu with links for "Main Page", "Classes", "Class List", and "Class Members". The main content area is titled "pyexample.PyClass Class Reference" and contains the following sections:

- Documentation for a class:** Includes a "More..." link and a "List of all members" link.
- Public Member Functions:** Lists the `__init__` method (The constructor) and the `pyMethod` method (Documentation for a method).
- Static Public Attributes:** Lists the `classVar = 0` attribute (A class variable).
- Detailed Description:** A section for documentation for a class, with a "More details" link.
- Member Function Documentation:** Provides details for the `pyexample.PyClass.pyMethod (self)` method, including its parameters and a note that the documentation was generated from a file named `pyexample.py`.

At the bottom of the page, it states: "Generated on Tue Oct 4 15:59:30 2005 for Python by **doxygen** v 1.5".

Interaktives Beispiel: Rationale Zahlen

- Ziel: Neue Klasse (= Typ) **Rational** mit allen Operatoren

```
# The class Rational
class Rational( object ):
    """ Brief Implements rational numbers
    Numbers are internally represented by two integers.
    The numerator and the denominator are kept in normal form
    (gcd or factored out), so as to avoid accumulation of large numbers.
    Note
    Allow Rational/Int mix like Rational/Int is allowed (see __add__). Also for + and -
    Implement Int + Rational ( __add__ ) et al
    """
    def __init__( self, num, denom = 1 ):
        """ Brief constructor = Rational(n,d)
        *param num: Double; (can be Rational, otherwise it will be converted to int, if possible)
        *param denom: (int), must > 0 warn
        """
        if isinstance( num, Rational ):
            # we make a copy
            self.num = num.getNumerator()
            self.denom = num.getDenominator()
            return
        num = convert_to_int( num )
        denom = convert_to_int( denom )
        if denom == 0:
            num = 0
            denom = 1
        elif denom < 0:
            # THIS check that all op's work with 0/0
            # better might be throw exception
            num = -num
            denom = -denom
        self.num = num
        self.denom = denom

    def __repr__( self ):
        """ Brief overloads printing ...
        str = '%d/%d' % (self.num, self.denom)
        return str
    """
    def getNumerator( self ):
        return self.num
    def getDenominator( self ):
        return self.denom

    def __add__( self, rhs ):
        """ Brief overloads binary +, ...
        *param rhs: Can be Rational, int, float, or str (Behavior undefined for other types)
        """
        if isinstance( rhs, Rational ):
            n = self.num + rhs.getNumerator()
            d = self.denom * rhs.getDenominator()
            return Rational( n, d )
        else:
            return Rational( self + Rational( rhs ) ) # recursion

    def __sub__( self ):
        """ Brief overloads unary -, ...
        return Rational( -self.num, self.denom)
    """
```



Unbehandelte Features



- Vererbung, Unterklassen,
- Exceptions
- Lambda-Funktionen
- Compilieren von Code zur Laufzeit durch das Programm selbst
- Embedding
- ...