Vorlesung

10

Klassen

Inhalt

Vorlesung

8

Lernziele: In der Objektorientierten Programmierung ist das Konzept der Klasse grundlegend für die Realisierung verschiedener Paradigmen: Abstraktion, Kapselung, Polymorphie, Vererbung, Introspektion. Diese sind zu verstehen und deren Realisierung in Python kennen zu lernen.

1 Objekte – Klassen – Instanzen

Schon in der Kapitel V02 hatten wir den Begriff „Objekt" im Kontext der Datentypen eingeführt und diskutiert. Kurz gesagt werden aus der Blickrichtung der Datentypen in der Objektorientierten Programmierung (OOP) zusammengehörige Daten und die darauf arbeitenden Programmteile zu Einheiten zusammengefasst, den so genannten **Objekten**. Die OOP schließt aber auch ein Verfahren zur Strukturierung von Computerprogrammen mit ein: Zumindest konzeptionell arbeitet ein Programm dann nicht mehr wie bei der imperativen/ prozeduralen Programmierung so, dass sequenziell einzelne Schritte eines Programms durchlaufen und Daten verändert werden, sondern die Daten stehen im Mittelpunkt des Denkens. Die Programmlogik entfaltet sich in der Kommunikation und den internen Zustandsveränderungen der Daten-Objekte, aus denen das Programm aufgebaut ist.

Folgende Paradigmen lassen sich durch ihre Schlagworte kennzeichnen:

- **Klassen**
- **Methode**
- **Abstraktion**

- **Kapselung**
- **Polymorphie**
- **Vererbung**
- **Introspektion**

Die allgemeine Idee hinter dem Klassen- und Objektkonzept ist relativ einfach und ist besser unabhängig von den vielen Details und Sonderfällen einer Implementierung zu verstehen. Im Unterschied zu den vorherigen Kapiteln soll deshalb zuerst die allgemeine Idee und dann ihre Umsetzung in Python beschrieben werden.

1.1 Klassen und Instanzen

Bei der Einführung von Datentypen definierten wir uns auch gleich die dazu gehörenden Methoden und fassten beides zu einem Objekt zusammen. Zur Definition und Verwaltung gleichartiger Objekte bedienen sich die meisten Programmiersprachen des Konzeptes der Klasse:

> **Klassen sind Vorlagen, aus denen die konkreten Objekte zur Laufzeit erzeugt werden.**

Im Programm wird dann für die vielen einzelnen Objekte gleicher Art nur eine einzige Klasse definiert; Klassen sind die „Konstruktionspläne" (Schablonen) für Objekte. Wir dürfen also eine Klasse als Konkretisierung eines abstrakten Datentyps ansehen: Sie legt nicht nur den **Datentypen oder Datenstrukturen** fest, aus denen die mithilfe der Klassen erzeugten Objekte bestehen, sie definiert auch die Algorithmen (**Methoden**), die auf diesen Daten operieren. Während also zur Laufzeit eines Programms einzelne Objekte miteinander interagieren, wird das Grundmuster dieser Interaktion durch die Definition der einzelnen Klassen festgelegt. Im Kontext der Datenstrukturen (Datentypen) realisieren Klassen also einen konstruktiven Weg, neue benutzerspezifische Datenstrukturen und die darauf arbeitenden Methoden zu beschreiben und zu programmieren.

In rein objektorientierten Sprachen wie Smalltalk oder Python werden dem Prinzip „**alles ist ein Objekt"** folgend, auch elementare Typen wie Ganzzahlen (Integer) durch Objekte repräsentiert. Auch Klassen selbst sind hier Objekte, die wiederum Ausprägungen von Metaklassen sind. Viele Sprachen, unter anderem C++ und Java, folgen allerdings aus Effizienzgründen nicht der „reinen Lehre" der Objektorientierung; daher sind dort elementare Typen keine vollwertigen Objekte, sondern müssen auf Methoden und Struktur verzichten.

Innerhalb einer Klasse bilden die **Attribute**, auch Eigenschaften genannt, die Datenstruktur. Attribute selbst können auch wieder komplexe Datentypen (z.B. Listen, Mengen, Dictionaries) oder auch wiederum Klassen sein.

Beispiel: Ein Klasse *Bankkonto* könnte beispielsweise folgendermaßen definiert werden:

- hat eine Kontonummer (Attribut *name*)
- hat einen Kontoinhaber (Referenz auf einen Kunden)
- hat eine Liste von Buchungen (Liste von Referenzen auf Buchungen)
- kann eine Einzahlung durchführen (Methode *deposit*)
- kann eine Auszahlung durchführen (Methode *withdraw*)
- kann den Saldostand mitteilen (Methode *inquiry*)

Jede Kontobewegung würde dann einem Methodenaufruf entsprechen und der Liste der Buchungen ein Element hinzufügen sowie den Saldo modifizieren.

Aus Klassen erzeugte Objekte werden **Instanzen** (Exemplare) der Klasse genannt.

In manchen Programmiersprachen gibt es zu jeder Klasse ein bestimmtes Objekt (Klassenobjekt), das dazu da ist, die Klasse zur Laufzeit zu repräsentieren und ansprechbar zu machen. Dieses Klassenobjekt ist dann z.B. zuständig für die Erzeugung von Objekten der Klasse durch einen **Konstruktor**.

Klassen werden in der Regel in Form von Klassenbibliotheken zusammengefasst, die häufig thematisch organisiert sind.

1.2 Abstraktion und Kapselung

Jedes Objekt im System kann als ein **abstraktes Modell** eines *Akteurs* betrachtet werden, der Aufträge erledigen, seinen Zustand berichten, ändern und mit den anderen Objekten im System kommunizieren kann, ohne offen legen zu müssen, wie diese Fähigkeiten implementiert sind. Dabei ist nur seine Schnittstelle bekannt. Dies entspricht den Forderungen eines abstrakten Datentyps (ADT).

Dies gilt auch für Objekte. Sie können den internen Zustand anderer Objekte nicht in unerwarteter Weise lesen oder ändern. Ein Objekt hat eine Schnittstelle, die darüber bestimmt, auf welche Weise mit dem Objekt interagiert werden kann. Als **Kapselung** (englisch: *encapsulation*) bezeichnet man den kontrollierten Zugriff auf Objekte.[1] Vom Innenleben eines Objektes soll der Benutzer (Programmierer des aufrufenden Objektes) möglichst wenig wissen müssen (**Geheimnisprinzip**, *information hiding*). Durch die Kapselung werden nur Informationen über das "Was" eines Objektes (was es leistet) nach außen sichtbar, nicht aber das "Wie" (die interne Repräsentation und die benutzten Algorithmen). Dadurch wird eine Schnittstelle nach außen definiert und zugleich dokumentiert.

Vorteile der Kapselung

- Dadurch, dass die Implementierung eines Objektes anderen Objekten nicht bekannt ist, kann die Implementierung geändert werden, ohne die Zusammenarbeit mit anderen Objekten zu beeinträchtigen. Es wird verhindert, dass innere Zusammenhänge, die möglicherweise später einmal verändert werden, Änderungen in anderen Programmteilen erfordern.

- Erhöhte Übersichtlichkeit, da nur die öffentliche Schnittstelle eines Objektes betrachtet werden muss.

[1] Anmerkung: Das Kapselungsprinzip gibt es auch unabhängig von objektorientierten Konzepten, z.B. als Modularisierungsprinzip.

- Beim Zugriff über eine Zugriffsfunktion spielt es von außen keine Rolle, ob diese Funktion 1:1 im Inneren des Objekts existiert, das Ergebnis einer Berechnung ist, oder möglicherweise aus anderen Quellen (z.B. einer Datei oder Datenbank) stammt.

  Eine Zugriffsmethode, die eine Eigenschaft eines Objekts abfragt, heißt auch **Abfragemethode** oder **Getter** (von englisch *to get* - etwas holen). Die Eigenschaft kann entweder direkt einer Instanzvariablen des Objekts entnommen oder im Moment des Aufrufs berechnet werden. Für den Aufrufer ist das nicht erkennbar. In einigen Programmiersprachen ist es üblich, die Namen aller Abfragemethoden mit **`get`** beginnen zu lassen. So könnte z. B. eine Methode, die den Namen einer Person abfragt, `getName` heißen. In anderen Programmiersprachen ist dies verpönt und die Methode hieße einfach `name`.

  Eine Zugriffsmethode, die eine Eigenschaft eines Objekts ändert, heißt auch **Änderungsmethode** oder **Setter** (von englisch *to set* - etwas einstellen). Ein Vorteil der Änderungsmethode besteht darin, dass sie vor der Änderung den Wert auf Gültigkeit prüfen kann. In einigen Programmiersprachen ist es üblich, die Namen aller Änderungsmethoden mit **`set`** beginnen zu lassen. So könnte z. B. eine Methode, die den Namen einer Person ändert, `setName` heißen. In anderen Programmiersprachen ist dieser Präfix verpönt und die Methode hieße einfach `name`.

- Deutlich verbesserte Testbarkeit, Stabilität und Änderbarkeit der Software.

Die **Unified Modeling Language** (UML) als De-facto-Standardnotation erlaubt die Modellierung folgender Zugriffsarten (in Klammern die Kurznotation der UML):

public (+): Zugreifbar für alle Ausprägungen (auch die anderer Klassen),

private (-): Nur für Ausprägungen der eigenen Klasse zugreifbar,

protected (#): Nur für Ausprägungen der eigenen Klasse und von Spezialisierungen derselben zugreifbar,

package (~): Erlaubt den Zugriff für alle Elemente innerhalb des eigenen Pakets.

Die Möglichkeiten zur Spezifizierung der Zugreifbarkeit sind je nach Programmiersprache unterschiedlich.

1.3 Vererbung

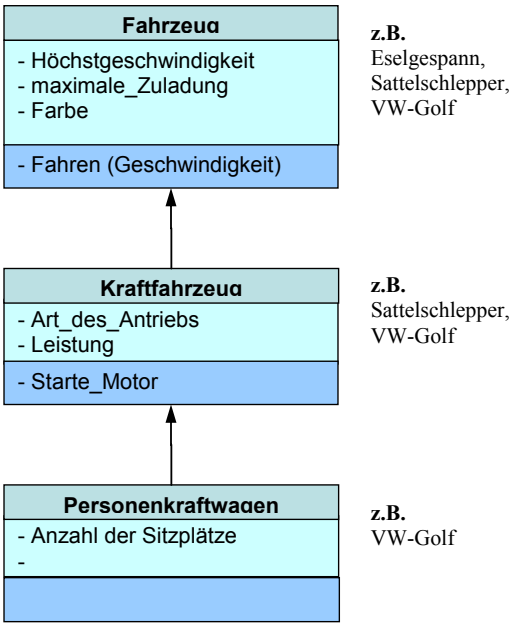
Klassen können auch als Vorlage für ähnliche, andere Klassen dienen. Man sagt dann, dass die neue Klasse von der alten Klasse „abgeleitet“ wurde und ihre Methoden und Attribute „ererbt“ habe (**Vererbung).** Dabei erbt die neue Klasse (**abgeleitete Klasse** bzw. **Subklasse**.) die Datenstruktur (Attribute) und die Operationen (Methoden) auf der Datenstruktur von der vererbenden Klasse (**Basisklasse** oder **Superklasse**). Die Nutzung der Vererbung bietet sich an, wenn es Objekte gibt, die konzeptionell aufeinander aufbauen. Gegebenenfalls lassen sich Objektdefinitionen von vorneherein so aufteilen, dass identische Merkmale in der Definition eines "vererbenden" Objektes zusammengefasst werden. Wird keine Vererbung zugelassen, so spricht man zur Unterscheidung oft auch von *objektbasierter Programmierung*

Eine besondere Bedeutung haben dabei **abstrakte** Klassen. Eine abstrakte Klasse definiert Schnittstellen für Methoden und Eigenschaften ohne die zugehörige Implementierung zur Verfügung zu stellen, so dass es nicht möglich ist, Objekte daraus zu generieren. Erst von daraus abgeleiteten Klassen können, sofern entsprechende Implementierungen der Methoden bzw. Eigenschaften zur Verfügung gestellt werden, Objekte generiert werden.

„Das" klassische **Beispiel** zu Klassen betrachtet die Einteilung von Fahrzeugen:

- Ein **Fahrzeug** besitzt bestimmte Attribute. Diese können z.B. Höchstgeschwindigkeit, maximale Zuladung und Farbe sein, die für alle Arten von Fahrzeugen existieren. Die Klasse **Kraftfahrzeug** erbt all diese Attribute, kann aber noch zusätzliche Attribute besitzen, die nur Kraftfahrzeuge haben, z.B. Art des Motors und Leistung. Des Weiteren kann ein Kraftfahrzeug auch zusätzliche Methoden wie *Motor starten* besitzen, welche die Basisklasse Fahrzeug nicht kennt.

- Die Klasse **Personenkraftwagen** kann dann wiederum von **Kraftfahrzeug** abgeleitet werden und weitere zusätzliche Attribute wie Anzahl der Sitze oder Farbe besitzen. Durch die Ableitung von Kraftfahrzeug erbt der Personenkraftwagen automatisch alle Attribute von Fahrzeug.



Ob eine Klasse in einer Vererbungsbeziehung zu einer anderen Klasse steht, lässt sich durch eine einfache **"ist-ein"-Regel** feststellen, also

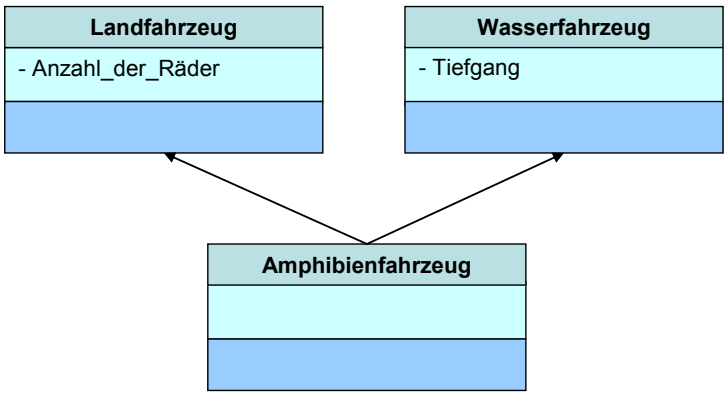
- Ein Personenkraftwagen **ist ein** Kraftfahrzeug,
- ein Personenkraftwagen **ist ein** Fahrzeug, aber
- ein Fahrzeug **ist kein** Personenkraftwagen.
- Ein Personenkraftwagen ist auch **kein** Sitz, sondern
- ein Personenkraftwagen **besitzt einen** oder **hat einen** Sitz.

Die Beziehung **„hat einen"** kennzeichnet die Attribute einer Klasse.

Obwohl die Ideen (die „Philosophie") hinter Klassen klar ist, ist in verschiedenen Programmiersprachen die Terminologie leider nicht einheitlich. Folgende Bezeichnungen werden synonym verwendet:

Superklasse	= Basisklasse	= Oberklasse
Subklasse	= abgeleitete Klasse	= Unterklasse
Methode	= Elementfunktion	= Memberfunktion
Attribut	= Datenelement	= Member
(aus einer Klasse erzeugtes) Objekt	= Exemplar	= Instanz

Von **Mehrfachvererbung** spricht man, wenn eine Klasse mehrere unmittelbare Basisklassen hat. Ein Anwendungsbeispiel hierfür ist die Modellierung eines Amphibienfahrzeugs, siehe unten. Es erbt sowohl die Attribute von Landfahrzeug als auch die von Wasserfahrzeug. Damit besitzt Amphibienfahrzeug sowohl eine *Räderzahl* als auch einen *Tiefgang*.



Nur wenige Programmiersprachen bieten die Möglichkeit der Mehrfachvererbung. Programmiersprachen mit Mehrfachvererbung sind z.B. C++, Eiffel und **Python**. Dagegen unterstützen z.B. Smalltalk und Ada Mehrfachvererbung nicht. Als Einwand gegen Mehrfachvererbung wird häufig genannt, dass es das Design **unnötig verkompliziere** und undurchsichtig machen könne.

Java, Delphi und C# bieten mit so genannten „Schnittstellen" eine eingeschränkte Form der Mehrfachvererbung. Hierbei kann eine Klasse maximal von einer Basisklasse abgeleitet werden, jedoch kann sie beliebig viele Schnittstellen erben. Damit verpflichtet sich diese Klasse, die Methoden der Schnittstelle zu erfüllen. Mit einfacher Vererbung und eingeschränkter Mehrfach-

vererbung in Form von Schnittstellen sind die meisten Anforderungen an ein Software-Design realisierbar, ohne die „Nachteile“ der uneingeschränkten Mehrfachvererbung in Kauf nehmen zu müssen.

1.4 Polymorphie

„Überdeckt“ ein neues Merkmal ein bei der Vererbung übernommenes Merkmal, dann spricht man von **Überschreiben**. In den meisten Programmiersprachen können in der abgeleiteten Klasse (**Subklasse**) die alten Methoden der Basisklasse überschrieben werden. Man kann also einzelne Methoden neu implementieren und außerdem eigene Methoden und Daten (Attribute) hinzufügen.

Ein Objekt der abgeleiteten Klasse kann überall verwendet werden, wo ein Objekt der Basisklasse erwartet wird; die überschriebenen Methoden werden dann aus der neuen, abgeleiteten Klasse ausgeführt. Der Wechsel zwischen den gleich bezeichneten, aber unterschiedlich implementierten Methoden wird **Polymorphie** („Vielgestaltigkeit“) genannt. Es ist also erlaubt, einem Wert oder einem Namen (z. B. einer Variablen) mehreren Typen zuzuordnen. Den Gegensatz dazu bildet Monomorphie. Dort ist jeder Name und jeder Wert von genau einem Typ.

Dies bedeutet: Verschiedene Objekte können auf die gleiche Nachricht unterschiedlich reagieren. Wird die Zuordnung einer Nachricht zur Reaktion auf die Nachricht erst zur Laufzeit aufgelöst, dann wird dies auch späte Bindung (oder *dynamische Bindung*) genannt.

1.5 Introspektion

Eine **Introspektion** (engl. *introspection*)**,** auch **Reflexion** (engl. *reflection*) genannt, bedeutet, dass ein Programm Erkenntnisse über seine eigene Struktur gewinnen kann. Introspektion ermöglicht es, z.B. zur Laufzeit Informationen über Klassen oder deren Instanzen abfragen zu können. Bei einer Methode sind das u.a. deren Sichtbarkeit, die Art des Rückgabewertes oder die Art der Übergabeparameter. Die Umsetzung ist dabei sprachspezifisch realisiert.

Eine wichtige Rolle spielt Introspektion im Zusammenhang mit typsicherer Programmierung, aber auch in Fragen der Persistenz (dauerhafte Datenhaltung von Objekten) und deren Beziehungen.

2 Klassen in Python

In der Programmiersprache Python sind Klassen der wesentliche Mechanismus, um neue Datenstrukturen und neue Datentypen zu definieren.

2.1 Die `class`- Anweisung

Eine *Klasse* definiert eine Anzahl von *Attributen*, die in allen Objekten, den *Instanzen* der Klasse, existieren. Diese Attribute beinhalten normalerweise Variablen, genannt **Klassenvariablen**, und Funktionen, die auch *Methoden* genannt werden. Klassen werden mit der `class`-Anweisung definiert.

Der Rumpf einer Klasse enthält mit `__init__(.)` eine Folge von Anweisungen, die ausgeführt werden, wenn die Klasse zum ersten Mal definiert wird.

Beispiel *Bankkonto*

```
class Account(object):
  """Eine einfache Klasse"""
  account_type = "Basic"
  def __init__(self, name, balance):
    """Initialisiere eine neue Account-Instanz."""
    self.name = name
    self.balance = balance
  def deposit(self, amount):
    """Addiere zur Bilanz hinzu."""
    self.balance = self.balance + amount
  def withdraw(self, amount):
    """Subtrahiere von der Bilanz."""
    self.balance = self.balance - amount
  def inquiry(self):
    """Gib aktuelle Bilanz zurueck."""
    return self.balance
```

Objekte, die während der Ausführung eines Klassenrumpfes erzeugt werden, werden in ein Klassenobjekt platziert, das als Namensraum dient. Auf die Attribute der Klasse `Account` kann man wie folgt zugreifen:

```
Account.account_type
Account.__init__
Account.deposit
Account.withdraw
Account.inquiry
```

Es ist wichtig, zu bemerken, dass eine `class`-Anweisung **keine** Instanzen einer Klasse erzeugt (d.h. im obigen Beispiel werden keine Konten erzeugt). Stattdessen definiert eine Klasse nur die Menge von Attributen und Methoden, über die alle Instanzen verfügen, sobald sie erzeugt werden. Funktionen, die innerhalb einer Klasse definiert werden (d.h. Methoden) operieren immer auf einer Klasseninstanz, die als erstes Argument übergeben wird. Gemäß einer **Konvention** wird dieses Argument **self** genannt, obwohl jeder erlaubte Bezeichner verwendet werden könnte. Klassenvariablen wie `account_type` sind solche, die allen Instanzen einer Klasse identisch zur Verfügung stehen, d.h. sie gehören nicht einzelnen Instanzen (für Java- und C++-Programmierer: Klassenvariablen verhalten sich wie statische Variablen).

Das obige Beispiel zeigt auch eine Vorgehensweise zur Dokumentation, den sogenannten Dokumentations-String: Wenn eine Funktion, Moduldatei oder eine Klasse mit einer String-Konstante beginnt (*Achtung*: keine Kommentare!), wird dieser String im Attribut `__doc__` des Objekts gespeichert. Die Funktion `help()` ist eine eingebaute Funktion, die das Python-Hilfesystem aufruft und speziell für den interaktiven Einsatz gedacht ist. Zum Beispiel liefert `help(Account)`

```
>>> help (Account)
Help on class Account in module __main__:

class Account
 | Eine einfache Klasse
 |
 | Methods defined here:
 |
 | __init__(self, name, balance)
 |   Initialisiere eine neue Account-Instanz.
 |
 | deposit(self, amt)
 |   Addiere zur Bilanz hinzu.
 |
 | inquiry(self)
 |   Gib aktuelle Bilanz zurueck.
 |
 | withdraw(self, amt)
 |   Subtrahiere von der Bilanz.
 |
 | ----------------------------------------------------------------------
 | Data and other attributes defined here:
 |
 | account_type = 'Basic'

>>>
```

Werkzeuge zur Navigation im Quellcode bzw. zur Erstellung von Dokumentation benutzen manchmal solche Dokumentations-Strings. Diese Strings sind als Attribute namens `__doc__` eines Objektes verfügbar. (Die Einrückung des Dokumentations-Strings muss konsistent sein mit der aller anderen Anweisungen der Definition.). Zusätzlich wird der Dokumentations-String auch zur Anzeige von „Tool-Tips" im interaktiven Interpreter genutzt, siehe unten. Also eine sehr hilfreiche kleine Dokumentationshilfe, die man sich unbedingt angewöhnen sollte. Weitere Anwendungen des Doc-Strings siehe: http://docs.python.org/lib/module-pydoc.html

Obwohl eine Klasse einen Namensraum definiert, stellt dieser Namensraum keinen Sichtbarkeitsbereich für den Code innerhalb des Klassenrumpfes dar. Daher müssen Referenzen auf andere Klassenattribute einen vollständig qualifizierten Namen verwenden.

Zusammengefasst:

- Die `class`-Anweisung erzeugt ein **Klassen-Objekt** und weist diesem einen Namen zu.
- Zuweisungen in `class`-Anweisungen erzeugen **Klassen-Attribute**, die Zustand und Verhalten von Objekten exportieren.
- **Klassenmethoden** sind `def` (ggf. verschachtelt) mit einem speziellen ersten Argument `self`, das die Instanzvariable aufnimmt.
- Zuweisungen an Attribute des ersten Arguments (zum Beispiel `self.X = obj`) in Methoden erzeugen instanzspezifische Attribute.

2.2 Klasseninstanzen

Instanzen einer Klasse werden erzeugt, indem man ein Klassenobjekt wie eine Funktion aufruft. Dieser Aufruf erzeugt eine neue Instanz und ruft dann die Methode `__init__()` innerhalb der Instanz auf, falls definiert.

Beispiel *Erzeugen von Konten*

```
a = Account("Rainer", 1000.00)
b = Account("Maria", 100000000000)
```

Diese Anweisungen rufen folgendes auf:
`Account.__init__(a, "Rainer", 1000.00)`

Nachdem die Instanz erzeugt worden ist, kann mit dem Punkt-Operator wie folgt auf ihre Attribute und Methoden zugegriffen werden:

```
>>> a.deposit(100.00)             # Ruft Account.deposit(a, 100.00) auf
>>> b.withdraw (20.00)
>>> print(a.name)
Rainer
>>> print(a.account_type)
Basic
>>> a.inquiry ()
1100.0
>>> b.inquiry ()
99999999980.0
>>>
```

Intern wird jede Instanz mit Hilfe eines Dictionary implementiert, namens `__dict__`, welches die Instanz-Attribute speichert. Dieses Dictionary enthält die Information, die für jede Instanz individuell verschieden ist. Beispiel:

```
>>> print(a.__dict__)
{'balance': 1100.0, 'name': 'Rainer'}
>>> print(b.__dict__)
{'balance': 99999999980.0, 'name': 'Maria'}
```

Wann immer Attribute einer Instanz geändert werden, ereignen sich diese Änderungen im lokalen Dictionary der Instanz. Innerhalb von Methoden, die in einer Klasse definiert werden, werden Attribute durch Zuweisung an die Variable `'self'` verändert, wie in den Methoden `__init__()`, `deposit()` und `withdraw()` der Klasse `Account` demonstriert wird.

Neue Attribute können jedoch zu jedem Zeitpunkt einer Instanz hinzugefügt werden. Beispiel:

```
>>> a.number = 123456             # Füge 'number' zu a.__dict__ hinzu
>>> print(a.number)
123456
```

Obwohl die Zuweisung an ein Attribut immer im lokalen Dictionary der Instanz erfolgt, ist der Zugriff auf Attribute etwas komplizierter. Immer wenn auf ein Attribut zugegriffen wird, sucht der Interpreter zunächst im Dictionary der Instanz. Bleibt dies erfolglos, sucht der Interpreter im Dictionary des Klassenobjektes, mit dem die Instanz erzeugt worden ist. Schlägt dies fehl, so wird eine Suche in den Oberklassen durchgeführt. Geht auch das schief, so wird ein letzter Anlauf unternommen, indem versucht wird, die Methode `__getattr__()` der Klasse aufzurufen, falls diese definiert ist. Versagt auch dies, wird eine `AttributeError`-Ausnahme ausgelöst.

2.3 Referenzzählung und Zerstörung von Instanzen

Alle Instanzen verfügen über einen Referenzzähler. Sobald dieser auf Null fällt, wird die Instanz automatisch zerstört. Bevor die Instanz jedoch zerstört wird, sieht der Interpreter nach, ob für das Objekt eine Methode namens `__del__()` definiert ist, und ruft diese gegebenenfalls auf. In der Praxis ist es selten notwendig, in einer Klasse eine `__del__()`-Methode zu definieren. Die einzige Ausnahme besteht darin, dass bei der Zerstörung eines Objektes eine Aufräumaktion durchgeführt werden muss, z.B. das Schließen einer Datei, die Terminierung einer Netzwerkverbindung oder die Freigabe anderer System-Ressourcen. Selbst in diesen Fällen jedoch ist es gefährlich, sich auf `__del__()` für eine saubere Terminierung zu verlassen, da es keine Garantie gibt, dass der Interpreter diese Methode auch aufruft, wenn er selbst terminiert. Ein besserer Ansatz dürfte es sein, eine eigene Methode, etwa `close()`, zu definieren, die eine solche Terminierung explizit durchführt.

Gelegentlich wird ein Programm die `del`-Anweisung verwenden, um eine Referenz auf ein Objekt zu löschen. Falls dies dann den Referenzzähler eines Objektes auf Null sinken lässt, wird auch die Methode `__del__()` aufgerufen. Im Allgemeinen jedoch wird `__del__()` nicht direkt von der `del`-Anweisung aufgerufen.

Zusammengefasst:

- Eine Klasse kann wie eine Funktion aufgerufen werden: `class_name()` erzeugt ein **neues Instanzobjekt** dieser Klasse.
- Jedes Instanzobjekt enthält **alle** Klassenattribute und bekommt seinen **eigenen Namensraum** für Attribute.
- Instanzen „erben" die Attribute jener Klasse, von der sie erzeugt werden, **sowie** all derer **Oberklassen** (siehe Vererbung).

2.4 Vererbung in Python

Wie einleitend dargelegt, ist *Vererbung* ein Mechanismus, um eine neue Klasse zu erzeugen, indem das Verhalten einer existierenden Klasse spezialisiert oder angepasst wird. Die ursprüngliche Klasse wird *Basis-* oder *Oberklasse* genannt. Die neue Klasse wird *abgeleitete* oder *Unterklasse* genannt. Wenn eine Klasse mittels Vererbung erzeugt wird, »erbt« sie die Attribute, die in ihren Basisklassen definiert sind. Allerdings darf eine abgeleitete Klasse beliebige Attribute neu definieren oder neue Attribute selbst hinzufügen.

In Python wird die Vererbung in der `class`-Anweisung mit einer durch Kommata getrennten Liste von Namen von Oberklassen angegeben.

Beispiel *Vererbung*

```
class A(object):
  varA = 42
  def method1(self):
    print("Klasse A : method1")

class B(object):
  varB = 37
  def method1(self):
    print("Klasse B : method1")
  def method2(self):
    print("Klasse B : method2")

class C(A, B):              # Erbt von A und B.
```

```
  varC = 3.3
  def method3(self):
    print("Klasse C : method3")

class D(object): pass

class E(C, D): pass         # Erbt von C und D.
```

Die Suche nach einem in einer Oberklasse definierten Attribut erfolgt mittels Tiefensuche und von links nach rechts, d.h. in der Reihenfolge, in der die Oberklassen in der Klassendefinition angegeben wurden. In der Klasse *E* aus dem vorigen Beispiel werden daher die Oberklassen in der Reihenfolge *C, A, B, D* abgesucht. Für den Fall, dass mehrere Klassen das gleiche Symbol definieren, gilt, dass das zuerst gefundene Symbol genommen wird.

Beispiel *Namensgleichheit*

```
>>> c = C()                 # Erzeuge eine Instanz der Klasse 'C'
>>> c.method3()             # Ruft C.method3(c) auf
Klasse C : method3
>>> c.method1()             # Ruft A.method1(c) auf
Klasse A : method1
>>> c.varB                  # Greift auf B.varB zu
37
>>>
```

Falls eine abgeleitete Klasse das gleiche Attribut definiert wie eine ihrer Oberklassen, benutzen Instanzen der abgeleiteten Klasse das Attribut der abgeleiteten Klasse selbst. Falls es wirklich notwendig sein sollte, auf das übergeordnete Attribut zuzugreifen, kann dazu ein vollständig qualifizierter Name benutzt werden.

Zusammengefasst:

- Klassen erben Attribute von allen in der Kopfzeile ihrer Klassendefinition angegebenen Klassen (Oberklassen). Die Angabe mehrerer Klassen bewirkt **Mehrfachvererbung**.

- Der Vererbungsmechanismus durchsucht zunächst die Instanz, dann deren Klasse, dann alle erreichbaren Oberklassen (von links nach rechts) und benutzt die erste gefundene Version eines Attributnamens.

2.5 Datenkapselung in Python

Allgemein gilt in Python, dass alle Attribute „öffentlich“ sind, d.h. alle Attribute einer Klasseninstanz sind ohne Einschränkungen überall sichtbar und zugänglich. Das bedeutet auch, dass alles, was in einer Oberklasse definiert wurde, an Unterklassen vererbt wird und dort zugänglich ist.

Dieses Verhalten ist in objektorientierten Anwendungen oft unerwünscht, weil es die interne Implementierung eines Objektes freilegt und zu Konflikten zwischen den Namensräumen von Objekten einer abgeleiteten und denen ihrer Oberklassen führen kann.

Um dies zu verhindern, werden alle Namen in einer Klasse, die mit einem doppelten Unterstrich beginnen, wie z.B. `__Foo`, derart „verstümmelt“, dass der Name die Form `_Classname__Foo` annimmt. Dies erlaubt es einer Klasse, private Attribute zu besitzen, da solche privaten Namen in einer abgeleiteten Klasse nicht mit den gleichen privaten Namen einer Oberklasse kollidieren können.

Beispiel *Private Daten in Klassen*

```
>>> class A(object):
  def __init__(self):
    self.__X = 3           # Verstümmelt zu self._A__X


>>> a = A()                # Neue Instanz der Klasse A
>>> type (A)               # Welcher Typ ist das?
<type 'classobj'>
>>> type(a)
<type 'instance'>
>>> a.__X                  # zeige __X der Instanz

Traceback (most recent call last):
 File "<pyshell#199>", line 1, in -toplevel-
  a.__X
AttributeError: A instance has no attribute '__X'
>>> a._A__X                # zeige __X der Instanz der Klasse A
3
>>>
```

Obwohl dieses Schema der Namensverstümmelung (engl. *name mangling*) den Eindruck einer Datenkapselung vermittelt, gibt es jedoch **keinen** streng funktionierenden Mechanismus, um den Zugriff auf „private" Attribute einer Klasse zu verhindern. Insbesondere dann, wenn der Name der Klasse und des entsprechenden Attributes bekannt sind, kann über die verstümmelten Namen darauf zugegriffen werden, wie im Beispiel gezeigt.

Dies gilt auch beim Import von Klassenobjekten. Zwar werden Namen in Modulen, die mit einem einzelnen Unterstrich beginnen, z.B. _Spam und jene die **nicht** in der `__all__` -Liste des Moduls vorkommen, werden bei einem Import der Form `from module import *` **nicht** bekannt gemacht. Auch dies ist jedoch **keine** echte Kapselung **(*privacy*)**, da solche Namen voll qualifiziert immer noch genutzt werden können.

2.6 Überladen von Operatoren

Klassen können die Benutzung von eingebauten Operatoren abfangen und sie neu implementieren, indem sie Methoden mit speziellen Namen definieren. Diese Namen beginnen und enden mit zwei Unterstrichen. Die Namen werden von Oberklassen normal ererbt. Pro Operation wird genau eine Methode ausgeführt, sofern sie in der Suchhierarchie (siehe folgender Abschnitt 2.8) gefunden wird, sonst tritt ein `NameError` auf. Python ruft also „automatisch" die überladene Methode einer Klasse auf, wenn Instanzen in Ausdrücken und anderen Kontexten vorkommen. Wenn eine Klasse zum Beispiel die Methode namens `__add__(self,other)` definiert und `a` eine Instanz dieser Klasse ist, so ist `a + other` äquivalent zu `a.add(other)`.

Das Überladen von Operatoren ist generell möglich. Die `__add__(self,other)`-Methode muss also keine Addition oder Verkettung implementieren. Beliebige Mischungen von numerischen Typen und Sequenztypen sowie veränderlichen und auch unveränderlichen Typen sind erlaubt.

Alle Datentypen stellen folgende Methoden zur Benutzung bereit:

`__init__(self[,arg]*)`	`__hash__(self)`	`__lt__(self,other)`
`__del__(self)`	`__call__(self[arg]*)`	`__le__(self,other)`
`__repr__(self)`	`__getattr(self,name)`	`__eq__(self,other)`
`__str__(self)`	`__setattr(self,name)`	`__ne__(self,other)`
`__cmp(self,other)`	`__delattr(self,name)`	`__gt__(self,other)`
`__rcmp__(self,other)`	`__getattribute__(self,name)`	`__ge__(self,other)`
`__slots__(string)`		

Für Zahlen sowie Sequenzen und Abbildungen sind viele weitere Operatoren definiert, siehe dort.

2.7 Introspektion in Python

Durch verschiedene eingebaute Funktionen bietet Python die Möglichkeit zur Introspektion:

`callable (objekt)`	Ergibt TRUE, wenn Objekt aufrufbar ist, sonst FALSE
`dir([objekt])`	Ohne Argument wird eine Liste aller Namen im aktuellen lokalen Geltungsbereich zurückgegeben. Bei der Angabe von objekt sind dieses die aktuellen Attributnamen dieses Objektes (Modulen, Klassen, Instanzen, Listen, Dictionaries, usw.) in sortierter Form.
`globals ()`	Ergibt ein Dictionary mit allen globalen Variablen des Aufrufers.
`hasattr (objekt,name)`	Ergibt TRUE, wenn objekt ein Attribut name hat, sonst FALSE

`help ([objekt])`	Ruft das Hilfesystem auf. (Für den interaktiven Einsatz gedacht.)
`id (objekt)`	Gibt die eindeutige Identität von objekt als Ganzzahl zurück (in den meisten Implementierungen seine Adresse im Speicher).
`isinstance (objekt, klasse)`	Ergibt TRUE, wenn objekt eine Instanz von klasse (oder Typ) oder Instanz einer Unterklasse davon ist, sonst FALSE. klasse oder Typ kann auch ein Tupel von Klassen und/oder Typen sein.
`issubclass (klasse1, klasse2)`	Ergibt TRUE, wenn klasse1 von klasse2 abgeleitet wurde, sonst FALSE. klasse 2 darf auch ein Tupel von Klassen sein
`locals ()`	Ergibt ein Dictionary mit allen lokalen Variablen des Aufrufers (mit name:wert pro Eintrag)
`repr (objekt)`	Ergibt einen String mit einer druckbaren und potenziell zu parsenden Darstellung von objekt
`str (objekt)`	Ergibt einen „schönen" String zum Ausdruck der Darstellung von objekt zurück.
`super (typ[,objekt])`	Gibt die Oberklasse von Typ zurück. Falls das zweite Argument weggelassen wird, ist das zurückgegebene Objekt unbeschränkt. Ist es ein Objekt, dann muss isinstance (obj,typ) war sein. Wenn das zweite Objekt typ1 ist, muss issubclass (typ1,typ) wahr sein.
`type (objekt)`	Ergibt ein Typobjekt, das den Typ von objekt wiedergibt. Nützlich beim Testen auf Typen, z.B. type (a) == type ([]). Siehe auch Modul types und isinstance.
`vars ([objekt])`	Ohne Argument gibt vars () ein Dictionary mit den Namen des aktuellen lokalen Namensraum zurück. Wenn objekt angegeben ist, den aktuellen lokalen Namensraum des objekts. vars () ruft objekt.__dict__ auf.

Achtung: Der Ausdruck type(a) == type(b) für zwei beliebige Objekte ist immer wahr, wenn sie Instanzen einer Klasse sind (selbst, wenn sie aus zwei verschiedenen Klassen erzeugt worden sind), nämlich 'instance'.

Um auf die Zugehörigkeit zu einer Klasse zu testen, sollte die eingebaute Funktion isinstance(obj, klasse) benutzt werden. Sie ergibt logisch wahr, falls ein Objekt obj zur Klasse klasse oder irgendeiner von klasse abgeleiteten Klasse gehört. isinstance() kann ebenso zur Typprüfung bei allen eingebauten Typen verwendet werden kann.

Entsprechend dazu ergibt die eingebaute Funktion issubclass(A, B) logisch wahr, wenn die Klasse A eine Unterklasse der Klasse B ist.

2.8 Zusammenfassung der Namenskonventionen

In der Implementierung von Python gibt es spezielle Konventionen für Namen im Gebrauch im Zusammenhang mit Klassen, die hier zur Übersicht aufgeführt werden.

(1) Namen, die mit **zwei Unterstrichen** beginnen und enden (zum Beispiel `__init__`), haben eine besondere Bedeutung für den Interpreter: Klassen fangen eingebaute Operationen ab und implementieren dieses auf ihre Art (überladen die Methode), indem sie Methoden mit zwei Unterstrichen beginnen und enden lassen, die sie von ihrer Oberklasse geerbt haben.

(2) Namen, die mit **einem Unterstrich** beginnen, (z.B. `_A`) und denen auf oberster Ebene eines Moduls zugewiesen wird, werden bei `from module import *` nicht sichtbar. (Pseudo-private Attribute)

(3) Den Namen in einer class-Anweisung, die mit zwei Unterstrichen beginnen, aber nicht damit enden, z.B. `__A`, wird der **Name der Klasse** vorangestellt (Pseudo-private Attribute)

(4) **Klassennamen** beginnen normalerweise mit einem großen Buchstaben, z.B. `MeineKlasse` (Camel Casing)

(5) Der erste (am weitesten links stehende) Parameter der Methodendefinition innerhalb von Klassen wird normalerweise `self` genannt.

(6) **Qualifizierte Namen** werden als Attribute bezeichnet und unterliegen den Regeln für Objekt-Namensräumen. Zuweisungen in bestimmten lexikalischen Geltungsbereichen (beziehen sich auf die Schachtelung im Quellcode eines Programmes) initialisieren Objekt-Namensräume (Module, Klassen).

Zuweisung: `objekt.X = wert` erzeugt oder ändert den Attributnamen im Objekt `objekt`. Für `objekt..X` in einem Ausdruck sucht der Interpreter nach dem Attributnamen X im Objekt `objekt` und dann in allen Oberklassen (bei Instanzen und Klassen: im Sinne der Vererbung).

(7) **Unqualifizierte Namen** unterliegen auch lexikalischen Gültigkeitsregeln. Zuweisungen binden solche Namen an den lokalen Gültigkeitsbereich, es sei denn, sie sind als global deklariert.

Zuweisung: `X = wert` `X` ist lokal: erzeugt oder ändert den Namen `X` im aktuellen lokalen Geltungsbereich, solange `X` nicht global definiert ist. Ist X global, erzeugt oder ändert diese Anweisung `X` im zugehörigen Globalen Bereich.

Referenz: `X` Ist der Name `X` lokal, wird in folgender Reihenfolge gesucht:
1. im aktuellen lokalen Geltungsbereich (innerhalb der Funktion),
2. dann in den lokalen Bereichen aller lexikalisch umgebenden Bereiche,
3. dann im aktuellen Globalen Bereich
4. dann im eingebauten Bereich `(__builtin__)`

Bei als global deklarierten Namen beginnt die Suche im globalen Geltungsbereich!

Geltungsbereiche für unqualifizierte Namen sind:

Kontext	Lokaler Bereich	Globaler Bereich
Modul	das Modul selbst	wie lokal, das Modul selbst
Funktion, Methode	Funktionsaufruf	umgebendes Modul
Klasse	`class`-Anweisung	umgebendes Modul
Skript, interaktiver Modus	`modul __main__`	wie lokal

Achtung: Programme können „versagen“ (funktionieren anders als erwartet/gewünscht), wenn sie den gleichen Namen im globalen und in einem lexikalisch umgebenden Geltungsbereich einer Funktion benutzen, da der Name einer umgebenden Funktion den globalen Namen verdeckt!

(8) Der nur aus einem Unterstrich bestehende Name _ wird im interaktiven Interpreter genutzt und stehet für das Ergebnis der letzten Auswertung.

Die Funktion `dir`([objekt])ist ein hilfreiches Werkzeug beim interaktiven Experimentieren: Ohne Argumente wird eine Liste aller Namen im aktuellen Geltungsbereich (Namensraum) zurückgegeben. Wenn ein Objektname (Modulname, Klassenname, Instanzname) übergeben wird, ist das Ergebnis eine sortierte Liste von Attributnamen dieses Objektes. Sie enthält auch geerbte Attribute.