

Rückblick

Wir haben wichtige Datenstrukturen kennen gelernt

Felder (array)	Listen (list)
Mengen (set)	Stapel (stack)
(Warte-)Schlange (queue)	Verbund (struct)
Dictionaries	Weitere

und deren Realisierungen in Python!

Und auch eine
strukturierte (Module++), prozedurale (def) Programmierung
... mit vielen Details.

und auch das Testen behandelt → PS2

PS2-Hausübung 1-Testen Datei

- Das Übungsblatt ist **individuell** zu bearbeiten (**kein** Pair-Programming)
- Es gelten dieselben Regelungen (Style Guide, Plagiate, Header...) wie bei EPR.
- Abgabetermin ist schon **Freitag, 22.12.2017 – 16.00 Uhr**.

... und psst: Es gibt in Moodle ein **Idle-Debugger Tutorium**.

Wir wissen schon (hoffentlich):

Programmieren erfordert Disziplin, Ausdauer, abstraktes Denkvermögen, Kreativität und hohe Lernbereitschaft!

Warum sage ich Ihnen das immer wieder?

- Klausur: 30 % Programmieraufgaben
- 50% Multiple Choice – viele Python-Fragen!
- Sie können schon 20% vorab haben! Rechnen Sie selbst ... es geht theoretisch ohne Programmieren, aber es wird sehr schwer!
- Erfahrungen mit dem PRG-Praktikum (3./4. Semester) ...
Klage dort: **Wir haben zu wenig Programmiererfahrung!**

Eine kurze historische Betrachtung

"Gute Software" (!???) –

Entwicklung durch die Überwindung von Krisen (1)

Mitte der 50er Jahre: Problem: **Lösungsansätze:**

Programmieren war entsetzlich viel
Detailarbeit wg. **Assembler-
programmierung**

- geringer Abstraktionsgrad
- viele Fehler
- sehr schlecht wartbar
- Hoher Portierungsaufwand

Prozedurale Programmierung

Höhere Programmiersprachen:

- Fortran (1956)
- Algol (!1958)
- Cobol (1959)
- Lisp (1958)

Grace Murray Hopper (1906 – 1992):

"She did this, the invention of the compiler, she said, because she was lazy and hoped that "the programmer may return to being a mathematician".

Eine kurze historische Betrachtung "Gute Software" (!???) – Entwicklung durch die Überwindung von Krisen (2)

Mitte der 60er Jahre: Problem: **Lösungsansatz:**

- 1. **Softwarekrise**
 - Software war zu groß und zu komplex geworden
 - verspätete Auslieferung
 - zu teuer
- Softwareengineering** (Softwaretechnik)
1968 (Garmisch)
- Strukturierte Programmierung**
(Vermeidung des goto - 1968)
- Modulare Programmierung** ✓
(Geheimnisprinzip (1972))
- Programmiersprachen:**
- Pascal (1970) → Modula (1975)
 - C (1971)
 - Prolog (1970) und Smalltalk (1971)

Eine kurze historische Betrachtung "Gute Software" (!???) – Entwicklung durch die Überwindung von Krisen (3)

Mitte der 70er Jahre: Problem: **Lösungsansätze:**

"Einführungskrise": stark steigende Komplexität der Projekte.

→ **IT-Projektmanagement** (und Softwareengineering) immer wichtiger.

Vorgehensmodelle
Wasserfallmodell (top-down)

Eine kurze historische Betrachtung "Gute Software" (!???) – Entwicklung durch die Überwindung von Krisen (4)

Mitte der 80er Jahre: Problem: **Lösungsansätze:**

2. Softwarekrise

immer komplexere Strukturen:
Netzwerke, Multimedia,
(GUIs = WIMP-Prinzipien)

→ Paradigmenwechsel im System-
entwurf: **Objektorientierung**

Programmiersprachen:

C++ (1983)

Python (1991) Ruby (1993)

Java (1995)

(Call-back-Strukturen bei GUIs)

Vorgehensmodelle

Spiralmodell (Zyklen)

7

Programmieren 1 – V12:
OO-Programmierung

Prof. Dr. Detlef Krönker

Eine kurze historische Betrachtung "Gute Software" (!???) – Entwicklung durch die Überwindung von Krisen (5)

Mitte der 90er Jahre: Problem: **Lösungsansätze: (nur in Ansätzen)**

Beschleunigungskrise:

IT ist Wettbewerbsfaktor
WWW übernimmt das Internet

Handys, Smartphones, Tablets

Programmiersprachen:

→ mehr Skriptsprachen

Python, Ruby, Perl, php, ...

Vorgehensmodelle → agil

Extreme Programming (XP) (2000)

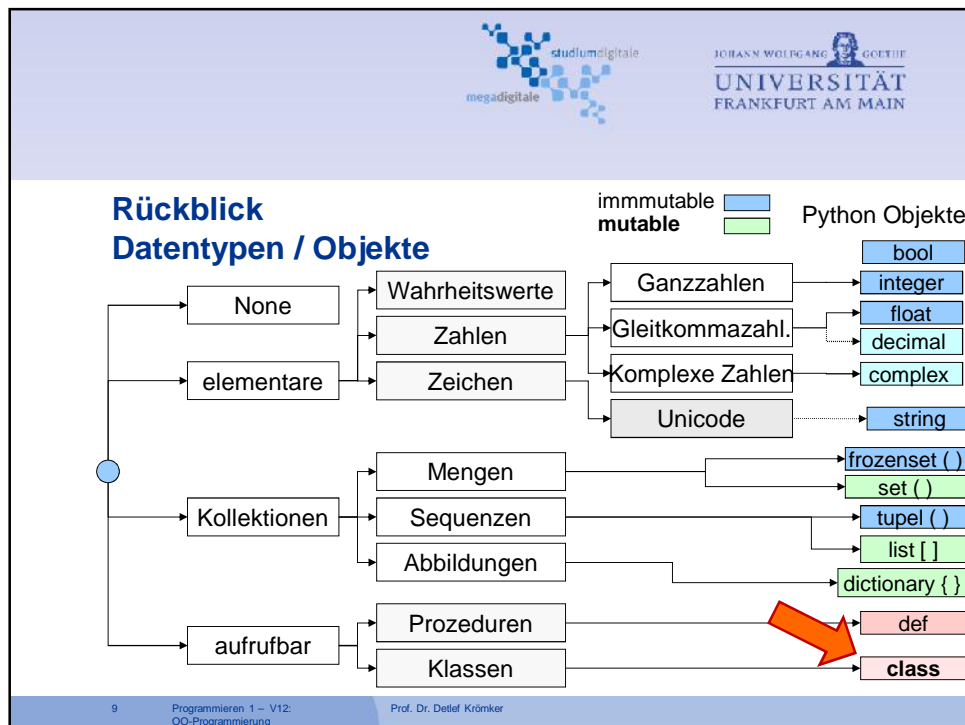
Agile Modelle: Scrum (2006)



→ **Viele Aufgaben für Sie!**

8

Programmieren 1 – V12:
OO-Programmierung

Prof. Dr. Detlef Krönker



Objekte ... kaum etwas Neues?

Aus der Blickrichtung der Datentypen ist OOP eine Variante der imperativen Programmierung, bei der

- zusammengehörige Daten (Attribute) und
- die darauf arbeitende Programmlogik (Methoden)

zu Einheiten zusammengefasst werden, den so genannten **Objekten**.

Ist aktuelles Ergebnis einer kontinuierlichen Entwicklung!

10 Programmieren 1 – V12: OO-Programmierung Prof. Dr. Detlef Krönker

Ein erster Unterschied

OO-Programmierung **strukturiert** die Software neu:

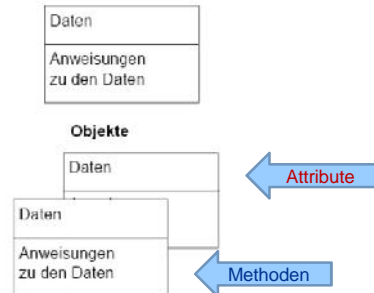
In den Objekten haben die Daten (Attribute) (nur **konzeptionell**) ihre "eigenen" Anweisungen (Methoden)

Prozedurales Programmieren



Anweisungen in einem großen Block (getrennt von den Daten)

Objektorientiertes Programmieren



Anweisungen jeweils bei den zugehörigen Daten

Unser heutiges Lernziel

In der **Objektorientierten Programmierung** ist das **Konzept der Objekte** grundlegend für die Realisierung verschiedener Paradigmen:

- Klasse (Attribute und Methoden) und Instanzen
- Abstraktion,
- Kapselung,
- Polymorphie,
- Vererbung,
- Introspektion.

Diese sind zu verstehen und deren **Realisierung in Python** kennen zu lernen.

Übersicht

OO-Paradigmen (grundsätzliche Denkweise, wissenschaftliche Schule)

- Klassen und Instanzen – Methoden und Attribute
- Abstraktion und Kapselung
- Vererbung und Polymorphie
- Introspektion

Klassen in Python

- Die class - Anweisung
- Klasseninstanzen (Zugriff auf Attribute und Methoden - Punktnotation)
- Referenzzählung und Zerstörung von Instanzen
- Vererbung
- Datenkapselung
- Überladen von Operatoren
- Introspektion
- Namenskonventionen

OO Analyse und Design → objektorientierte Softwareentwicklung und UML
(erst in 2 Wochen)

Klassen und Instanzen (1)

Klassendiagramm

Klassen sind Vorlagen, aus denen
Objekte (**Instanzen**) zur Laufzeit erzeugt werden.

Name
- Attribute
- Methoden

- Im Programm (Quellcode) werden dann nicht einzelne Objekte, sondern eine Klasse gleichartiger Objekte definiert.
- Klassen sind die „Konstruktionspläne“ (Schablonen) für Objekte.
- Die Klasse entspricht in etwa einem Datentyp, geht aber darüber hinaus: Sie legt nicht nur die Schnittstelle fest, sondern beschreibt die **Datenstrukturen**, aus denen die erzeugten Objekte bestehen, sie definiert zudem die Algorithmen (**Methoden**).

Klassen und Instanzen (2)

- Im Kontext der Datenstrukturen (Datentypen) realisieren Klassen also einen **konstruktiven Weg neue benutzerspezifische Datenstrukturen** und die darauf arbeitenden Methoden **zu beschreiben** (zu programmieren) → eigene Datentypen.
- In rein objektorientierten Sprachen wie Smalltalk und auch **in 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 nicht der „reinen Lehre“. Python aber ja!

Klassen

- Die (Daten-) **Struktur einer Klasse** bilden die **Attribute** (auch Eigenschaften).
- **Attribute** selbst können auch wieder komplexe Datentypen (z.B. Listen, Mengen, Dictionaries) oder **auch wieder Klassen sein**.
- Aus Klassen erzeugte Objekte werden **Instanzen** (Exemplare) der Klasse genannt.
- In manchen Programmiersprachen gibt es zu jeder Klasse ein bestimmtes Objekt (Klassenobjekt), das dazu dient, die Klasse (also die Schablone) zur Laufzeit zu repräsentieren; dieses Klassenobjekt ist dann z.B. zuständig für die Erzeugung (Instanzierung) von Objekten der Klasse durch einen Konstruktor.

Beispiel:

- Ein **Bankkonto** könnte beispielsweise folgendermaßen definiert werden:
- hat eine Kontonummer (Attribut: `name`)
- hat einen Kontoinhaber (Attribut: Referenz auf einen Kunden)
- hat eine Liste von Buchungen (Attribut: Liste von Referenzen auf Buchungen)
- kann eine Einzahlung durchführen (Methode `deposit`)
- kann eine Auszahlung durchführen (Methode `withdraw`)
- kann den Saldostand mitteilen (Methode `inquiry`)

Dieses und die zugehörige Syntax kennt der anwendende Programmierer
... nicht aber wie es implementiert ist!

Kapselung (encapsulation):

bezeichnet den **kontrollierten Zugriff** auf Objekte.

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

Vom Innenleben eines Objektes soll der Benutzer (Programmierer des aufrufenden Objekts) möglichst wenig wissen müssen
(**Geheimnisprinzip** (Parnas: *information hiding*)). Wie bei Modulen.

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 Realisation).

Es wird eine Schnittstelle nach außen definiert und zugleich dokumentiert.

Kapselung

Kapselung: Schutz von **Attributen** vor dem Zugriff von außerhalb der Klasse.
Zugriff auf die Attribute erfolgt nur über entsprechende Methoden
(**Zugriffsmethoden**)

- Im allgemeinen gibt es zu einem bestimmten Attribut eine Methode, die den Wert des Attributes liefert - häufig als **Getter** (von *to get*) und eine andere, mit deren Hilfe man den Wert eines Attributes verändern kann - häufig als **Setter** (von *to set*) bezeichnet.

Zentrales Modell ist hier der **abstrakte Datentyp**, in dem Daten in einer Datenstruktur zusammengefasst sind, auf die **nur** über festgelegte Zugriffsfunktionen (Prozeduren, Setter&Getter) zugegriffen werden kann. (Abstrakte Datentypen können auf verschiedene Weisen implementiert werden)

Ein **anderes Beispiel** in modernen Programmiersprachen ist das 'Verbergen von Daten' (streng genommen natürlich deren Namen) innerhalb von Gültigkeitsbereichen von Namen: Funktion, Klasse, Modul, ...

Geheimnisprinzip

Geheimnisprinzip (von Parnas: *information hiding*):
Unter dem Geheimnisprinzip versteht man das **Verbergen von internen Informationen** (Struktur und Namen) und Implementierungsdetails nach außen! - (auch schon bei Modulen benutzt!)

Vom Innenleben eines Objektes soll der Benutzer (Programmierer des aufrufenden Objekts) möglichst wenig wissen müssen (Wie schon bei Modulen benutzt.)

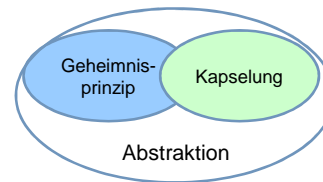
Durch das Geheimnisprinzip werden nur Informationen über das "Was" eines Objektes (was es leistet) nach außen sichtbar, nicht aber das "Wie" (die interne Repräsentation und Realisation).

Kapselung + Geheimnisprinzip = Abstraktion

encapsulation + information hiding = abstraction

Strenge Definition:

Kapselung + Geheimnisprinzip = Abstraktion



Aber:

Als Kapselung bezeichnet man in der **objektorientierten Programmierung** (siehe auch UML) **häufig** den kontrollierten Zugriff auf **Methoden und Attribute**.

(aber **Vorsicht**: in der Literatur werden **die drei Begriffe** häufig synonym verwendet)

Kapselung in UML „Universal Modelling Language“

Sichtbarkeit: Attribute und Methoden können sein:

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 der selben zugreifbar

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

Sichtbarkeiten sind in verschiedenen Programmiersprachen unterschiedlich realisiert:

JAVA: bei der Deklaration der Attribute und Methoden:

Für die Sichtbarkeiten gibt es **Schlüsselwörter**, die Sichtbarkeitsmodifizierer. *public* (öffentlich), *private* (privat) und paketsichtbar (ohne Modifizierer) *protected* umfasst (seltsamerweise) zwei Eigenschaften:

- (1) *protected*-Eigenschaften werden an alle Unterklassen vererbt.
- (2) Klassen, die sich im gleichen Paket befinden, können alle *protected*-Eigenschaften sehen.

C++ kennt sogenannte *friend(s)*

(Das Schlüsselwort *friend* teilt dem Compiler mit, dass eine Funktion oder Klasse Zugriff auf die als *private* oder *protected* deklarierten Elemente der eigenen Klasse hat.)

Frage: Wie realisiert man diese Sichtbarkeiten in Python? – Wo schauen sie nach?

Vorteile der Abstraktion (Kapselung + Geheimnisprinzip)

- Dadurch, dass die Implementierung eines Objektes anderen Objekten nicht bekannt ist, kann die Implementierung geändert werden, ohne die Zusammenarbeit mit anderen Objekten zu beeinträchtigen.
- Beim Zugriff über eine Zugriffsfunktion (*get*) 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.
- Erhöhte die Übersichtlichkeit, da nur die öffentliche Schnittstelle eines Objektes betrachtet werden muss.
- Definiert einen eigenen Namensraum (Gültigkeit): Deutlich verbesserte Testbarkeit, Stabilität und Änderbarkeit der Software

Durchatmen! Paradigmen der OO-Programmierung

- **Abstraktion** ☒
- **Kapselung** ☒
- **Polymorphie**
- **Vererbung**
- **Introspektion**
- **Klassen** ☒
- **Methoden** ☒

Vererbung und Polymorphie

Klassen können von anderen Klassen abgeleitet werden (**Vererbung**).

- Dabei erbt die Klasse die Datenstruktur (Attribute) und die Methoden von der vererbenden Klasse (Basisklasse).
- In der abgeleiteten Klasse (Subklasse) können Methoden der Basisklasse in den meisten Programmiersprachen **überschrieben** werden, d.h. einzelne Methode neu implementiert, und eigene Methoden und Daten (Attribute) hinzugefügt werden.
- Ein Objekt der abgeleiteten Klasse kann überall verwendet werden, wo ein Objekt der Basisklasse erwartet wird; überschriebene Methoden werden dann auf der abgeleiteten Klasse ausgeführt (**Polymorphie**).
- Verschiedene Objekte können auf die gleiche Nachricht unterschiedlich reagieren.

Vererbung und Polymorphie (2)

Die Nutzung der **Vererbung** bietet sich an, wenn es Objekte gibt, die konzeptionell aufeinander aufbauen oder sich spezialisieren. Gegebenenfalls lassen sich Objektdefinitionen von vorneherein so aufteilen, dass identische Merkmale in der Definition eines "vererbenden" Objektes zusammengefasst werden.

Unter bestimmten Voraussetzungen können Algorithmen, die auf den Schnittstellen eines bestimmten Objekttypes operieren, auch mit davon abgeleiteten Objekten zusammenarbeiten.

Superklasse – Subklasse ... Terminologie

- Vererbungsbeziehungen zwischen Objekten werden in der Regel mit Hilfe von **Klassendefinitionen** hergestellt.
- Dabei bezeichnet man die "vererbende" Klasse als **Basisklasse** oder auch **Superklasse** und die "erbende" Klasse als **abgeleitete Klasse** bzw. **Subklasse**: Verschiedene Begriffshierarchien sind üblich:

Superklasse	= Basisklasse	= Oberklasse
Subklasse	= abgeleitete Klasse	= Unterklasse
Methode	= Elementfunktion	= Memberfunktion
Attribut	= Datenelement	= Member
Exemplar	= Instanz	

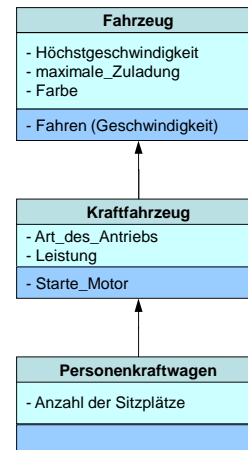
„Das“ klassische Beispiel

Ein **Fahrzeug** besitzt bestimmte Attribute, diese können z.B. Höchstgeschwindigkeit, maximale Zuladung und Farbe sein.

Die Klasse **Kraftfahrzeug** erbt all diese Attribute, kann aber noch zusätzliche Attribute besitzen. Des Weiteren kann ein Kraftfahrzeug auch zusätzliche Methoden wie *Motor starten* besitzen

Die Klasse **Personenkraftwagen** kann dann wiederum von **Kraftfahrzeug** abgeleitet werden

Durch die Ableitung von Kraftfahrzeug erbt der Personenkraftwagen automatisch alle Attribute von Kraftfahrzeug und Fahrzeug.



Kriterien

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 – weil nicht jedes Fahrzeug ein Personenkraftwagen ist!

Ein Personenkraftwagen ist auch **kein** Sitz, sondern

- ▶ ein Personenkraftwagen **besitzt einen** oder **hat einen** Sitz.
- ▶ Die Beziehung **„hat einen“** kennzeichnet die Attribute einer Klasse.

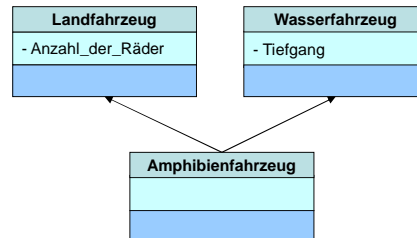
Mehrfachvererbung

Von **Mehrfachvererbung** spricht man, wenn eine Klasse mehrere unmittelbare Basisklassen hat.

Beispiel: Modellierung eines Amphibienfahrzeugs

hat Attribute von Landfahrzeug

als auch die von Wasserfahrzeug erbt.



Aus Wikipedia:
<https://de.wikipedia.org/wiki/Datei:Amphicar-stuttgart-2005.jpg>
 Urheber: Enslin
 Zuletzt zugegriffen: 15.03.2017

Implementierung der Mehrfachvererbung

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?

Eingeschränkte Implementierungen der Mehrfachvererbung

- 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 Schnittstellen zu erfüllen.
- Mit einfacher Vererbung und eingeschränkter Mehrfachvererbung in Form von Schnittstellen sind die meisten Anforderungen an ein Software-Design realisierbar, ohne die „Nachteile“ der uneingeschränkten Mehrfachvererbung in Kauf nehmen zu müssen.

Introspektion

- Introspektion (engl. *introspection*, auch Reflexion (engl. *reflection*)) bedeutet, dass ein Programm Erkenntnisse über seine eigene Struktur gewinnen kann.
- Eine wichtige Rolle spielt Introspektion im Zusammenhang mit typsicherer Programmierung, aber auch in Fragen der Persistenz (persistente Datenhaltung von Objekten und deren Beziehungen).
- 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**, der **Typ des Rückgabewertes** oder der **Typ der Übergabeparameter**. Die Umsetzung ist dabei sprachspezifisch realisiert.

Übersicht -- Durchatmen

Objekte – Klassen – Instanzen

- Klassen und Instanzen
- Kapselung
- Vererbung und Polymorphie
- Introspektion

Klassen in Python

- Die class - Anweisung
- Klasseninstanzen
- Referenzzählung und Zerstörung von Instanzen
- Vererbung
- Datenkapselung
- Überladen von Operatoren
- Introspektion
- Zusammenfassung der Namenskonventionen

Klassen -die class – Anweisung

Klassen werden in Python durch die `class` – Anweisung definiert.

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

Konvention:
Klassennamen werden in
CamelCasing notiert!

- In den Anweisungen werden

- Attribute und
- Methoden

definiert.

Attribute

- Klassen definieren eine Menge von Variablen (➔ **Attribute**)
- zu unterscheiden sind:
 - **Instanzattribute**, "nicht-statische" oder „dynamische“:
Existieren für jedes Objekt (= Instanz) genau einmal.
Kennzeichen: Werden schreibend und lesend durch
`<self>.<attributname>` innerhalb der Klasse
`<Instanzname>.<attributname>` außerhalb der Klasse
 - **Klassenattribute**, statische Attribute, gelten für eine Klasse, haben für alle Instanzen denselben Wert, Klassenattribut existieren pro Klasse nur einmal; jedem Klassenattribut muss ein Initialwert zugewiesen werden.

Als Konvention: Die Klassenattribute (statische Attribute) werden direkt unterhalb der class-Anweisung positioniert. (nach dem Docstring).

Methoden

- Klassen definieren eine Menge von Funktionen (➔ **Methoden**)
- zu unterscheiden sind normale Methoden, statische Methoden und Klassenmethoden
 - **"normale" Methoden** werden innerhalb einer class-Anweisung mit


```
def <methodname>(self, ???):
    """DocString."""
```

"normale" Methoden sind an die Instanzen gebunden und werden aufgerufen mit
`<instance_name>.<methodname>`
 - Es gibt in Python auch die Möglichkeit sogenannte "statische Methoden" oder "Klassenmethoden" zu definieren. Das führt jetzt zu weit. Sorry.

Instanzieren von Klassen

- d.h. mithilfe der Schablonen (=Klassen) konkrete Objekte erzeugen
- Klassen Instanziierung **benutzt die Funktionsnotation**.
- Man tut so, als ob das Klassenobjekt eine parameterlose Funktion wäre, die eine neue Instanz der Klasse zurückgibt.

- Zum Beispiel:

```
x = MyClass()
```

- Dies erzeugt eine neue Instanz der Klasse und weist dieses Objekt der lokalen Variable x zu.

39

Programmieren 1 – V12:
OO-Programmierung

Prof. Dr. Detlef Krönker

Am Beispiel:

```
class Account:
    """A simple class for
    account_type = "Basic"
    def __init__(self, name, balance):
        """Initializes
        self.name = name
        self.balance = balance
    def deposit(self, amount):
        """Add to balance.
        self.balance = self.balance + amount
```

Dokumentations-String: String-Konstante direkt nach class Name: Dieser String im Attribut **__doc__** des Objekts gespeichert. Auch bei Methoden anwenden!

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).

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). Gut, dass das keine Klassenvariable ist! ;-)

Innerhalb der Klasse `Account` kann man folgendes zugreifen:

- `Account.account_type`
- `Account.__init__()`
- `Account.deposit()`
- `Account.withdraw()`
- `Account.inquiry()`

40

Programmieren 1 – V12:
OO-Programmierung

Prof. Dr. Detlef Krönker

Beispiel Fortsetzung

```
def withdraw(self, amount):
    "Subtract from balance."

    self.balance = self.balance - amount

def inquiry(self):
    "Return current balance."

    return self.balance
```

help(Account) (1)

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

```
class Account(builtins.object)
    A simple class for a bank account.

    Methods defined here:

    __init__(self, name, balance)
        Initialize a new account.

    deposit(self, amount)
        Add to balance.

    inquiry(self)
        Return current balance.

    withdraw(self, amount)
        Subtract from balance
```

Superklasse, die Klasse von der Account abgeleitet wurde.

help(Account) (2)

```
-----
Data descriptors defined here:
__dict__
    dictionary for instance variables (if defined)
__weakref__
    list of weak references to the object (if defined)
-----
Data and other attributes defined here:
account_type = 'Basic'
```

Achtung:

Es ist wichtig festzuhalten, dass eine `class`-Anweisung **keine** Instanzen einer Klasse erzeugt (d.h. im obigen Beispiel werden keine Konten erzeugt).

Aber mit der Definition werden die Klassen **einmal** ausgeführt (nicht die Methoden an und für sich)

Stattdessen definiert eine Klasse nur die Menge von Attributen und Methoden, über die alle Instanzen verfügen, sobald sie erzeugt werden, ist also eine Schablone.

Normale Funktionen, die innerhalb einer Klasse definiert werden (d.h. Methoden) operieren immer auf einer Klasseninstanz, die als erstes Argument übergeben wird, laut Konvention `self` genannt.

Zwischendurch zusammengefasst:

- Die `class`-Anweisung erzeugt ein **Klassen-Objekt** und weist diesem einen Namen zu: Konvention CamelCasing
- Zuweisungen auf oberster Ebene in `class`-Anweisungen erzeugen **Klassen-Attribute**, die Zustand und Verhalten von Klassen exportieren.
- Methoden** sind (ggf. verschachtelte) `def` mit einem speziellen ersten Parameter `self`, das die "Instanzvariable" aufnimmt.
- Zuweisungen an das Attribut `x` des ersten Arguments (zum Beispiel `self.x = obj`) innerhalb von Methoden erzeugen instanzspezifische Attribute.

Instanzen einer Klasse erzeugen

werden erzeugt, indem ein Klassenobjekt (wie eine Funktion) aufgerufen wird

Dies erzeugt eine neue Instanz und ruft die Methode `__new__()` auf, die dann die Methode `__init__()` der Klasse, falls definiert.

Beispiel: zum Erzeugen von Konten:

```
a = Account("Rainer", 1000.0)
```

```
b = Account("Maria", 1000000000.0)
```

Diese Anweisung erzeugt eine Instanz

`Account.__init__(a, "Rainer", 1000.00)` respektive
`Account.__init__(b, "Maria", 1000000000.0)`

```
def __init__(self, name, balance):
    "Initialisiere eine neue Account-Instanz."
    self.name = name
    self.balance = balance
```

Beispiele für Zugriffe - Punktnotation!

```
>>> a.deposit(100.0) # Ruft Account.deposit(a, 100.0)
    auf
>>> b.withdraw (20.0)
>>> print (a.name)
Rainer
>>> print (a.account_type)
Basic
>>> a.inquiry ()
1100.0
>>> b.inquiry ()
99999999980.0
>>>
```

help(a) liefert dasselbe wie help(Account)

Referenzzählung und Zerstörung von Instanzen

- Alle Instanzen verfügen über einen Referenzzähler.
- Sobald dieser auf Null fällt, wird die Instanz zerstört (wie bei Variablen)
- 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.
- Gelegentlich wird ein Programm die `del`-Anweisung verwenden, um andere Referenzen, offene Dateien, etc. zu löschen.

Zusammengefasst:

- Eine Klasse wie eine Funktion aufzurufen `a = ClassName()` erzeugt ein neues Instanzobjekt dieser Klasse mit dem Namen `a`
- Jedes Instanzobjekt enthält alle Klassenattribute und bekommt seinen eigenen Namensraum für Attribute.
- Instanzen erhalten alle Attribute der Klasse von der sie erzeugt werden, sowie all derer Oberklassen (siehe Vererbung)

Klassen in Python

Klassen sind der wesentliche Mechanismus, um

- Datenstrukturen und neue **Datentypen** zu definieren.

Klassen sind u.a. ein Mechanismus (irgendwo zwischen Unterprogrammen und Modulen) **um ein Programm zu strukturieren.**

Klassen erlauben die charakteristische Art der OO-Analyse und des OO-Designs. Kommt später.

Vererbung in Python

- ist ein Mechanismus, um eine neue Klasse zu erzeugen, indem das Verhalten einer existierenden Klasse spezialisiert oder angepasst wird.
- Die ursprüngliche Klasse wird Basis- Ober- oder **Superklasse** 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.
- Vererbung wird in der `class()`-Anweisung mit einer durch Kommata getrennten Liste von Namen von Oberklassen angegeben.

Beispiel

```
class A:
    var_a = 42
    def method1(self):
        print("Klasse A : method1")
class B:
    var_b = 37
    def method1(self):
        print("Klasse B : method1")
    def method2(self):
        print("Klasse B : method2")
class C(A, B): # Erbt von A und B.
    var_c = 3.3
    def method3(self):
        print("Klasse C : method3")
class D: pass
class E(C, D): pass
```

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.

Die Oberklassen werden in der Reihenfolge C, A, B, D abgesucht (Tiefensuche). Für den Fall, dass mehrere Klassen das gleiche Symbol definieren, gilt, dass das zuerst gefundene Symbol genommen wird.

Beispiel (2)

```
>>> c = C()
>>> c.method3() # Ruft C.method3(c) auf
Klasse C : method3
>>> c.method1() # Ruft A.method1(c) auf
Klasse A : method1
>>> c.var_b # Greift auf B.varB zu
37
>>>
```

Überschreiben (override) von Methoden

- Überschreiben (**override**) beschreibt eine Technik in der objektorientierten Programmierung, die es einer abgeleiteten Klasse erlaubt, eine eigene Implementierung einer von der Basisklasse geerbten Methode zu definieren:
- **Man schreibt in der abgeleiteten Klasse einfach eine Methode gleichen Namens!**
- Dann ersetzt die überschreibende Methode der abgeleiteten Klasse die überschriebene Methode.
- Anmerkung: Es ist auch möglich in der überschreibenden Methode die Methode, die man überschreibt, aufzurufen.

Alles ist ein Objekt: Die Klasse `object` (davon sind alle Klassen abgeleitet)

class object: The most base type.
(sagt der Interpreter und der muss es ja wissen! ;-))

➔ Dies sind die Methoden, die jedes Objekt hat und die man ggf. anpassen kann.

```
__new__()  
__init__()  
__str__()  
__eq__(other)
```

Überladen (overloading) von Methoden

- Überladen kennt man beispielsweise in C++ und Java. Um es gleich vorweg zu sagen: **Es gibt kein Überladen von Methoden in Python.** Es wird auch nicht benötigt.
- Überladen von Methoden wird in statisch getypten Sprachen wie Java und C++ benötigt, um die gleiche Funktion mit verschiedenen Typen oder verschiedener Anzahl von Parametern zu definieren.
- ... das geht in Python direkt und immer.

Beispiel: "Überladen" (overloading) von Operatoren (nicht Methoden)

- ▶ Klassen fangen **eingebaute Operatoren** ab und implementieren sie, indem sie Methoden mit speziellen Namen definieren, die mit zwei Unterstrichen beginnen und enden. Diese Namen werden von Oberklassen normal vererbt.
- ▶ Pro Operation wird genau eine Methode ausgeführt, sofern sie in der Suchhierarchie (siehe Namenskonventionen) gefunden wird, sonst tritt ein `NameError` auf.
- ▶ Python ruft also „automatisch“ die überladene Methode einer Klasse auf.
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)`.

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 Attribut- oder Methodennamens.
- ▶ Methoden können überschrieben werden, indem in der abgeleiteten Klasse eine Methode gleichen Namens definiert wird. Diese Methode kann beliebige Parameter haben!

Datenkapselung

- ▶ 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 (Widerspruch zur Abstraktion) und zu Konflikten zwischen den Namensräumen von Objekten einer abgeleiteten und denen ihrer Oberklassen führen kann.

(Fast) Private Daten in Klassen und Modulen

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.

Dies ist jedoch **keine echte Kapselung (privacy)**, da solche Namen voll qualifiziert immer noch genutzt werden können:

```
<module>.__Spam
```

Zusammenfassung Sichtbarkeiten von Namen

Namen	Bezeichnung	Bedeutung
name	public	Attribute ohne führende Unterstriche sind sowohl innerhalb einer Klasse als auch von außen les- und schreibbar.
<u>name</u>	protected	Man kann zwar auch von außen lesend und schreibend zugreifen, aber der Entwickler macht damit klar, dass man dies nicht benutzen soll. Protected-Attribute sind insbesondere bei Vererbungen von Bedeutung.
<u><u>name</u></u>	private	Sind von außen nicht sichtbar und nicht benutzbar

aber ...

61

Programmieren 1 – V12:
OO-Programmierung

Prof. Dr. Detlef Krönker

Zusammenfassung Konventionen (1)

(1) Attribute ohne am Anfang sind public.

(2) Alle Namen in einer Klasse, die mit einem doppelten Unterstrich beginnen, wie z.B. Foo, werden „verstümmelt“, so dass Foo die Form ClassNameFoo annimmt: **name mangling**.

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.

(3) Namen, die mit zwei Unterstrichen beginnen und mit zwei Unterstrichen enden (zum Beispiel init), haben eine besondere Bedeutung für den Interpreter: Klassen fangen damit „eingebaute“ **Operatoren** ab und implementieren diese auf ihre Art (Überladen des Operators), indem sie **Methoden mit zwei Unterstrichen beginnen und enden** lassen, die sie von ihrer Oberklasse geerbt haben

62

Programmieren 1 – V12:
OO-Programmierung

Prof. Dr. Detlef Krönker

Zusammenfassung Konventionen (2)

- (4) Klassennamen beginnen normalerweise mit einem großen Buchstaben, z.B. MeineKlasse **und benutzen CamelCasing.**
- (5) Der **erste** (am weitesten links stehende) Parameter der Methodendefinition innerhalb von Klassen wird **self** genannt.
- (6) Qualifizierte Namen 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).

Zusammenfassung der Namenskonventionen (3)

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

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 <code>__main__</code>	wie lokal

Introspektion

Python stellt einige eingebaute Funktionen bereit, siehe Skript und Python Dokumentation.

In der Regel sind Programme, die so etwas benutzen relativ schwer zu verstehen (trickreich???) → selten benutzen – nur wenn nötig.

Fragen und (hoffentlich) Antworten

Ausblick

Als nächstes kümmern wir uns **um User Interfaces und GUIs.**

Wir nehmen insbesondere

**Danke für Ihre
Aufmerksamkeit!**