

Programmierung mit Python für Einsteiger:: Kapitel 7 - OO: Ableitung und Vererbung

Inhaltsverzeichnis

Wdh. Objektorientierte Programmierung

Motivation Vererbung

Motivation Vererbung 2

Nomenklatur

Beispiel Vererbung

Beispiel Ableitung

Technische Grundlagen (nach [EK])

Überschreiben von Methoden

Überschreiben von Methoden 2

Überschreiben von Methoden 3

Überschreiben von Methoden 4

Zugriffsrechte

🔗 Übung 20 - Girokonto

Komplexere Klassenhierarchien

Hinweise

Mehrfachvererbung

Beispiel Mehrfachvererbung

Mehrfachvererbung 2

Mehrfachvererbung 3

Standardklassen als Basisklassen

Beispiel Standardklasse erweitern

Wie geht es weiter?

🔗 Weitere Übungen

Referenzen

Wdh. Objektorientierte Programmierung

- Idee: Modellierung der Fachlichkeit durch „Objekte“
- Beispiel: Bankanwendung:
 - Konto
 - Kontoauszug
- Auf solchen Objekten sind Operationen möglich, z.B. Einzahlung auf ein Konto;

- *Klasse*: Allgemeine Beschreibung:
 - Was charakterisiert ein Konto (*Attribute*)?
 - Welche Operationen (*Methoden*) sind auf einem Konto möglich?
 - *Objekt*: Konkrete *Instanz* eines „Konto“
 - Prinzipien: Datenkapselung, Datenabstraktion, Geheimnisprinzip
-

Motivation Vererbung

- Das Prinzip der Vererbung ist ein weiteres wesentliches Prinzip der Objektorientierten Programmierung.
 - Idee:
 - Wiederverwendung durch Benutzung von Gemeinsamkeiten.
 - Hierarchiebildung
 - Bsp: Biologie:
 - Reich (z.B. Tiere)
 - Stamm (z.B. Wirbeltiere)
 - Klasse (z.B. Säugetiere)
 - Ordnung (z.B. Primaten)
 - Familie (z.B. Hominidae)
 - Gattung (z.B. Homo)
-

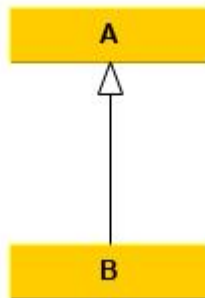
Motivation Vererbung 2

- „Jedes Säugetier ist ein Wirbeltier“ ...
 - ... und hat („erbt“) damit alle Eigenschaften eines Wirbeltieres.
 - Um ein Säugetier zu beschreiben, genügt es also, die zusätzlichen Eigenschaften zu definieren. Die Beschreibung der Eigenschaften der Wirbeltiere werden in diesem Sinne wiederverwendet.
 - Hierarchiebildung ist ein allgemeines Konzept zur Beherrschung von Komplexität.
 - Zurück zu Klassen: Hierarchiebildung über Ableitung. Leitet eine Klasse **B** von einer Klasse **A** ab, erbt **B** alle Eigenschaften der Klasse **A**. Man kann **B** dann um zusätzliche Eigenschaften erweitern (und anpassen).
-


Nomenklatur

Die folgenden Begriffe meinen alle dasselbe:

- Klasse B ist abgeleitet von Klasse A.
- B erbt von A.
- A ist Basisklasse von B.
- B ist abgeleitete Klasse (Kind-Klasse) von A.



Beispiel Vererbung

- Ein Girokonto ist ein „normales“ Konto mit zusätzlichen Eigenschaften (z.B. sind Überweisungen von einem Girokonto auf ein anderes möglich).
- Hat man eine Klasse Konto implementiert, kann man diese zur Implementierung eines Girokontos wiederverwenden, indem man die Klasse Girokonto von der Klasse Konto ableitet (und beispielsweise um eine Methode für Überweisungen erweitert).
- Eine weitere Klasse könnte ein Sparkonto sein, die ebenfalls von der Konto-Klasse ableitet.
- Ein Sparkonto hat zusätzlich zu einem Konto einen Zinssatz und eine Methode zur Berechnung der Zinsen.
-  `konto_spar`

Beispiel Ableitung

Die Idee der Kontoklasse stammt aus Johannes Ernesti, Peter Kaiser: Python 3, Das umfassende Handbuch, Rheinwerk Computing

```
class Konto:

    def __init__(self, inhaber, kontonummer, kontostand = 0):
        self._inhaber = inhaber
        self._kontonummer = kontonummer
        self._kontostand = kontostand

    # Methode hat Zugriffsrecht protected, d.h. ist in Klasse und Unterklasse aufrufbar,
    # aber nicht von außen
    def _pruefe_betrag(self, betrag):
        if betrag <= 0.0:
            raise Exception("Negativen Betrag nicht möglich!")

    def einzahlen(self, betrag):
        self._pruefe_betrag(betrag)
        self._kontostand += betrag

    def auszahlen(self, betrag):
        self._pruefe_betrag(betrag)
        self._kontostand -= betrag

    def __str__(self):
        return "Inhaber: " + self._inhaber + ", Kontonummer: " + str(self._kontonummer)
        + ", Kontostand: " + str(self._kontostand)

class Sparkonto(Konto):

    def __init__(self, inhaber, kontonummer, zinssatz, kontostand = 0):
        super().__init__(inhaber, kontonummer, kontostand)
        self.__zinssatz = zinssatz

    def berechne_zinsen_pro_jahr(self):
        # Das gilt natürlich nur, wenn sich der kontostand ein Jahr nicht ändert. Ist
        # halt ein Festgeld-Sparkonto :- )
        return self.__zinssatz / 100 * self._kontostand

    def __str__(self):
        return super().__str__() + ", Zinssatz: " + str(self.__zinssatz)

if __name__ == "__main__":
    k = Sparkonto("Sabine", 4610, 0.5, 100.00)
    print(str(k) + ": Zinsen: " + str(k.berechne_zinsen_pro_jahr()))
```

Technische Grundlagen (nach [EK])

- Definition einer Klasse B, die von einer Klasse A abgeleitet ist:

```
class B(A):
```

- Beispiel:

```
class A:
    def __init__(self):
        self.x = 123
        print("Konstruktor von A")
    def m(self):
        print("m von A. self.x = ", self.x)

class B(A):
    def n(self):
        print("n von B")
b = B() # Konstruktor von A
b.n()   # Methode n von B
b.m()   # Methode m von A. self.x = 123
```

Überschreiben von Methoden

- Die Klasse B hat die Implementierung von `__init__()` von A geerbt.
- Ein abgeleitete Klasse benötigt aber oft einen eigenen Konstruktor (um die zusätzl. Attribute von B zu initialisieren – im Beispiel den Zinssatz):

```
class B(A):
    def __init__(self):
        self.y = 4711
        print("Konstruktor von B")
    def n(self):
        print("n von B. self.y = " self.y)

b = B() # Konstruktor von B
b.n()   # n von B. self.y = 4711
b.m()   # AttributeError: 'B' object has no attribute 'x'
```

Überschreiben von Methoden 2

- B erbt zwar `__init__()` von A, „überschreibt“ sie dann aber mit einer eigenen Implementierung von `__init__()`.
- Beim Erzeugen des Objekts b wird also nur `__init__()` von B aufgerufen, nicht `__init__()` von A. Somit wird auch das Attribut x der Klasse A nicht angelegt.

- Generell lässt sich jede Methode der Basisklasse überschreiben.
- Die Methode der Basisklasse lässt sich aber explizit aufrufen.
- Für die `__init__()` - Methode sollte das auch immer gemacht werden, damit das Objekt korrekt initialisiert ist, d.h. damit alle Attribute des „Basisklassenanteils“ korrekt initialisiert sind.

Überschreiben von Methoden 3

Eine korrekte Implementierung der abgeleiteten Klasse B:

```
class B(A):
    def __init__(self):
        super().__init__() # Aufruf der Methode der Basisklasse
        self.y = 4711
        print("Konstruktor von B")
    def n(self):
        print("n von B. self.y = ", self.y)

b = B() # Konstruktor von A, Konstruktor von B
b.n()   # n von B. self.y = 4711
b.m()   # m von A. self.x = 123
```

Überschreiben von Methoden 4

- Jede Methode der Basisklasse kann überschrieben werden.
- Jede Methode der Basisklasse kann explizit aufgerufen werden (egal in welcher Methode).

```
class B(A):
    def __init__(self):
        super().__init__()
        self.y = 4711
    def m(self):
        print("Methode m von B.")
        super().m()

b = B()
b.m() # m von B. m von A. self.x = 123
```

Zugriffsrechte

Erinnerung: Attribute und Methoden

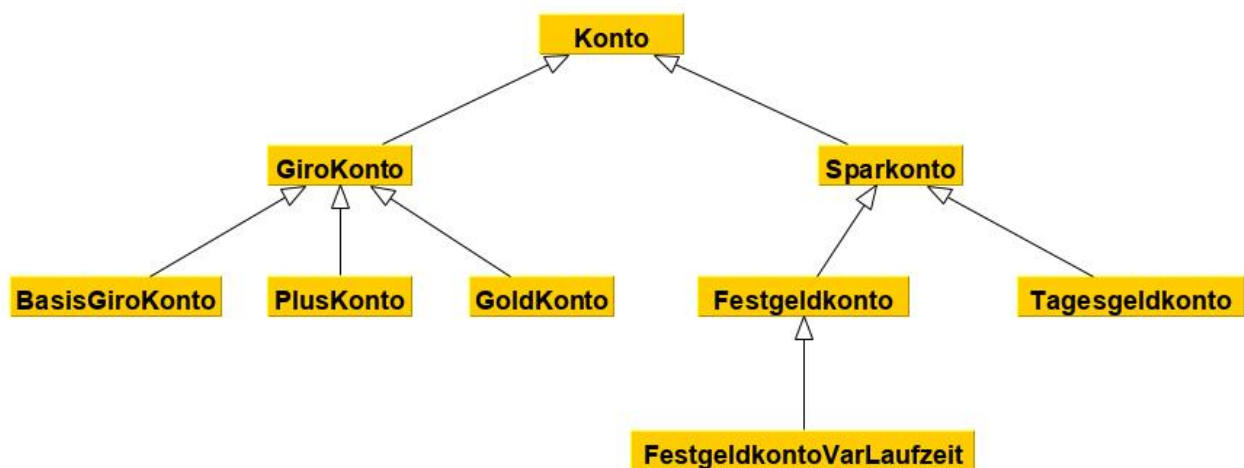
- Öffentlich: Ohne Unterstrich. Beliebiger Zugriff möglich.
- Private: Beginnend mit zwei Unterstrichen: Zugriff nur innerhalb der Klasse möglich.

- Protected: Beginnend mit einem Unterstrich: Zugriff aus Klasse und abgeleiteten Klassen möglich. Nicht von außen.

? Übung 20 - Girokonto

- Leiten Sie von der Klasse `Konto` eine weitere Klasse `GiroKonto` ab.
- Diese erweitert die Klasse `Konto` um eine Methode, um einen Geldbetrag auf ein anderes `Konto` zu überweisen.
- Parameter der Funktion:
 - das Zielgirokonto
 - der zu überweisende Betrag
- siehe Ordner `uebungen/20_uebung_ableitung_girokonto`

Komplexere Klassenhierarchien

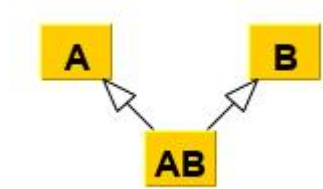


Hinweise

- Der Entwurf der „richtigen“ (angemessenen) Klassen und der Hierarchie ist die eigentliche „Kunst“ bei der OO-Software-Entwicklung.
- Keine unnötig komplexe Klassenhierarchien bilden.
- Bsp.: Braucht man im letzten Beispiel Klassen zur Unterscheidung von `PlusKonto` und `GoldKonto` oder tut es auch ein Attribut in `GiroKonto`?
- Ableitung nur bilden, wenn eine „ist ein“-Beziehung gilt. Ableitungen, die nur zum Zwecke der Code-Vererbung gebildet werden, obwohl „ist ein“ nicht gilt, sind zu vermeiden.
- Alternative zu Vererbung in diesem Fall: Verwendung eines Attributs vom Typ der Klasse, deren Methoden man wiederverwenden will.

Mehrfachvererbung

- Wie gesehen, ist es möglich, von einer Klasse mehrere Klassen abzuleiten (Girokonto und Sparkonto von Konto).
- Das umgekehrte ist auch möglich: Eine Klasse erbt von mehreren.
- Beispiel: **!** `mehrfach.py`



- Fragen:
 - Welchen Wert hat das Attribut `s` des Objekts `ab`? `"B"`: Weil der Konstruktor von `B` nach dem Konstruktor von `A` aufgerufen wird
 - Beim Aufruf von `self.zeige()` aus der Klasse `AB`: Wird `zeige()` von `A` oder von `B` aufgerufen? `zeige()` von `A`, weil `A` in der Ableitungsliste zuerst auftaucht.
-

Beispiel Mehrfachvererbung

```
class A:

    def __init__(self):
        self.s = "A"

    def zeige(self):
        print("Zeige von A: self.s = ", self.s)

class B:

    def __init__(self):
        self.s = "B"

    def zeige(self):
        print("Zeige von B: self.s = ", self.s)

class AB(A,B):

    def __init__(self):
        A.__init__(self)
        B.__init__(self)

    def zeigeAB(self):
        print("Zeige von AB: self.s = ", self.s)
        self.zeige()

ab = AB()
ab.zeigeAB()
```

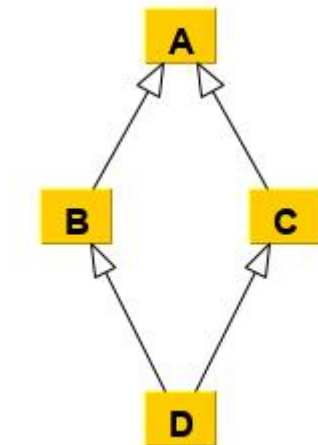
Mehrfachvererbung 2

- Will man explizit die Methode einer bestimmten Basisklasse aufrufen, so kann man diese angeben, s. `__init__()`-Methode der Klasse `AB`:

```
def __init__(self):
    A.__init__(self)
    B.__init__(self)
```

Mehrfachvererbung 3

- „Diamond-Problem“: Eine Klasse D leitet von Klassen B und C ab, die beide von A ableiten.



- Empfehlung:
 - Mehrfachvererbung vermeiden (und insbesondere Konstrukte wie beim Diamond-Problem).
 - Mehr dazu: https://www.python-kurs.eu/python3_mehrfachvererbung.php

Standardklassen als Basisklassen

- Man kann auch von Standardklassen ableiten und diese damit um Funktionalität erweitern.
- Beispiel [Kle]: Klasse `list` um eine Methode `push` erweitern: ⓘ `plist.py`
- ⓘ Übung [Kle]: Übung 5: `21_uebung_list_erweitern`:

Klasse `Plist` um Methode `splice` erweitern: `splice(offset, length, replacement)` „offset“ entspricht dem ersten Index, ab dem die Elemente gelöscht werden sollen, und der Parameter „length“ gibt an, wie viele Elemente gelöscht werden sollen. An die Stelle der gelöschten Elemente sollen die Elemente der Liste `replacement` eingefügt werden.

Beispiel Standardklasse erweitern

```
# Aus: Bernd Klein: Einführung in Python 3

class PList(list):
    def __init__(self, list):
        super().__init__(list)

    def push(self, element):
        self.append(element)

if __name__ == "__main__":
    x = PList([33,456,8,34,99])
    x.push(47)
    print(x)
```

Wie geht es weiter?

Selbst programmieren! Anregungen:

- <https://bmu-verlag.de/programmierideen-vorschlaege-fuer-die-umsetzung-eigener-programme/>
- c't-Sonderheft: Python-Projekte: Von alltagstauglich bis völlig nerdig
- s. auch Projekte aus [\[Mat\]](#)

🔗 Weitere Übungen

- Verkettete Liste (siehe letztes Kapitel): siehe `19_uebung_verkettete_liste`
- binäre Suche in einer indizierten Liste (`l[i]`):
 - gegeben: Sortierte Liste von Einträgen (z.B. Strings)
 - Finde die Position eines bestimmten Elementes
 - siehe `09A_uebung_binaere_suche`
 - Hinweis: iterativ und rekursiv lösbar
- Sortieren:
 - Bubble-Sort
 - Merge-Sort (s. `09B_uebung_merge_sort`)
 - Hinweis: gut lösbar über Rekursion

Referenzen

- [Kle] Bernd Klein, Einführung in Python 3
- [EK] Johannes Ernesti, Peter Kaiser: Python 3, Das umfassende Handbuch, Rheinwerk Computing
- [Mat] Eric Matthes: Python Crashkurs – Eine praktische, projektbasierte Programmierintroduction, dpunkt.verlag