Objektorientierte Programmierung

Theorie

Was ist objektorientierte Programmierung?

Objektorientierte Programmierung (kurz: *OOP*) ist ein Programmierstil, bei dem ein Programm so formuliert wird, dass es sich wie eine reale Lösungsbeschreibung lesen lässt.

Beispiel 1

In Python sind wir unter anderem bei Listen- und Stringobjekten bereits mit dem OOP-Konzept in Berührung gekommen.

- ```
  L = [3, 5, 8, 9]
  L.append(7) # fügt 7 am Listenende ein
  ```
- ```
  oldStr = 'abcde'
  newStr = oldStr.upper() # => Grossbuchstaben
  ```

Beispiel 2

Errate, was das folgende Programm in den einzelnen Zeilen macht.

```
1  from svg import Image
2
3  width = 600
4  height = 400
5  img = SVG(width, height)
6  A = [100, 300]
7  B = [500, 100]
8  M = [300, 200]
9  radius = 80
10 img.set_line_width(5)
11 img.set_line_color(0,0,255)
12 img.draw_line(A, B)
13 img.set_line_color(255,0,0)
14 img.draw_circle(M, radius)
15 img.save("testbild.svg")
```

Abstrakte Datentypen

In der OOP-Programmierung definieren wir eigene abstrakte Datentypen (sogenannte Klassen), mit denen wir die zu verarbeitenden realen Objekte in unseren Programmen abbilden.

Das Wort abstrakt bedeut hier, dass wir eine Abstraktionswelt erschaffen, in der wir die Objekte gedanklich einfacher verarbeiten können, so dass wir uns danach nicht mehr mit den Details der Implementierung befassen müssen.

Vorteile des OOP-Ansatzes

- Die Objektdarstellung ermöglichst es, die Programmierung gedanklich enger an die zu lösenden Aufgabe zu binden.
- Der Programmcode lässt sich besser verstehen und warten.
- Der Programmcode lässt sich (bei sorgfältigem Entwurf) weiterverwenden (code reuse) oder erweitern.

Beispiel 3

Der folgende Programmcode definiert eine Klasse (Bauplan oder Objektfabrik) zur Verwaltung von Bankkonten.

```
1  class Konto:
2
3      kontenliste = []  # Klassenvariable
4
5      def show_all():  # Klassenmethode
6          for kto in Konto.kontenliste:
7              print(kto)
8
9      def __init__(self, name, nummer):
10         self.name = name       # Instanzvariable
11         self.nummer = nummer   # ...
12         self.saldo = 0
13         Konto.kontenliste.append(self)
14
15     def einzahlung(self, betrag): # Instanzmethode
16         self.saldo += betrag
```

```
18      def ueberweisung(self, other, betrag): # Instanzmeth.
19          self.saldo -= betrag
20          other.saldo += betrag
21
22      def __str__(self): # Instanzmethode (Spezialmethode)
23          txt = ''
24          txt += 'Kontonummer: {0}\n'.format(self.nummer)
25          txt += 'Kontostand: {0}\n'.format(self.saldo)
26          return txt
27
28  # Testcode
29  k1 = Konto('Meier', 12345)
30  k1.einzahlung(500)
31  print(k1)
32
33  k2 = Konto('Müller', 44444)
34  k2.einzahlung(800)
35  print(k2)
36
37  k1.ueberweisung(k2, 100)
38  Konto.show_all()
```

Klasse

Eine Klasse ist ein bestimmter Bauplan zur Erzeugung von Objekten. In Python wird eine Klasse mit dem Schlüsselwort

```
class <Klassenname>:

    <Klassenrumpf/Klassendefinition>
```

eingeleitet, wobei die vom Anwendungsprogrammierer selber definierte Klassen nach Konvention mit einem Grossbuchstaben beginnen sollten.

Instanz (Synonym: Objekt)

Eine Instanz ist ein zur Laufzeit erzeugtes Objekt einer Klasse.

Im Beispiel 3 sind `k1` und `k2` Instanzen der Klasse `Konto`.

Instanzvariable (Synonym: Objektvariable)

Eine Variable, die zu einer Instanz gehört.

Im Beispiel 3 sind `self.name`, `self.nummer` und `self.saldo` Instanzvariablen.

Instanzmethode (Synonym: Objektmethode)

Eine Funktion, die zu einer Instanz gehört und so Zugriff auf die Eigenschaften (=Variablen) der Instanz hat.

Die Instanzmethoden in Beispiel 3 sind

- `einzahlung(self, ...)`,
- `ueberweisung(self, ...)` und
- `__str__(self)`.

Instanzmethoden erkennt man daran, dass ihr erster Parameter `self` ist. **Einzige Ausnahme:** `__init__(self, ...)` ist keine Instanzmethode.

Konstruktor

Eine spezielle Methode, mit der die Eigenschaften eines Objekts initialisiert werden können. Der Konstrukor liefert eine Referenz auf das neu erstellte Objekt zurück.

In Python wird der Konstruktor mit Hilfe der Spezialmethode `__init__(self,...)` definiert. Wobei das erste Argument in der Parameterliste den symbolischen Verweis auf die (zu erzeugende) Instanz darstellt. Üblicherweise wird dafür `self` verwendet. Mit den übrigen Paramtern kann ein Objekt individuell mit Werten versorgt werden.

Der Konstruktor wird aufgerufen, indem man den Klassennamen, als Funktion mit den Parametern *nach* `self` aufruft und einer Variablen zuweist. Diese Variable enthält danach (via `self`) die Adresse, an der die Instanz (das Objekt) gespeichert ist.

Spezialmethoden

Eine Spezialmethode erkennt man an den doppelten Unterstrichen vor und nach dem Methodennamen. Neben der Spezialmethode `__init__` gibt es weitere Spezialmethoden, mit der bestehende Operatoren und Funktionen von Python überschrieben werden können.

- `__str__(self)`: definiert die Textdarstellung von `self`
- `__add__(self, other)`: definiert `self+other`
- `__sub__(self, other)`: definiert `self-other`
- `__mul__(self, other)`: definiert `self*other`
- `__len__(self)`: definiert `len(self)`
- usw.

Klassenvariable

Eine Variable die zu einer Klasse gehört und die unabhängig von einer Instanz denselben Wert hat.

In Beispiel 3 ist dies die Variable `kontenliste`. Um zur Laufzeit auf diese Variable zuzugreifen, muss man ihr mit der Punkt-Schrebweise den Klassennamen voranstellen, also `Konto.kontenliste` schreiben.

Klassenmethode

Eine Funktion, die zu einer Klasse gehört und unabhängig von einer Instanz ist.

In Beispiel 3 ist dies die Variable `kontenliste`. Um zur Laufzeit auf diese Methode zuzugreifen, muss man ihr mit der Punkt-Schrebweise den Klassennamen voranstellen, also `Konto.show_all()` schreiben.

Aufgabe 1

1. Kopiere das Python-Modul `svg.py` aus dem Transfer-Verzeichnis in den Python-Ordner deines persönlichen Informatik-Laufwerks.
2. Erstelle im gleichen Order ein neues Python-Modul mit dem Namen `svg_client.py` und importiere mit dem folgenden Code
   ```
   from svg import Image
   ```
   die `Image`-Klasse aus dem `svg`-Modul.
3. Programmiere ein eigenes kleines Bild mit verschiedenen Farben, Linien und Kreisen wie in Beispiel 2.
4. Führe das Programm `svg_client.py` aus. Wenn es fehlerfrei ausgeführt wurde kannst du das Bild mit einem Doppelklick auf den Dateinamen öffnen. Es sollte dann in einem Browserfenster erscheinen.

Aufgabe 2

1. Kopiere das Python-Modul `konto.py` vom Transfer-Verzeichnis in den Python-Ordner deines persönlichen Informatik-Laufwerks.
2. Erweitere die `__str__(self)`-Methode so, dass auch der Name des Kontoinhabers angezeigt wird.
3. Implementiere nach der Methode
   `einzahlung(self, betrag)`
   die Methode
   `auszahlung(self, betrag)`,
   mit der `betrag` vom Konto abgehoben wird.
   Optional: Kontrolliere vor der Auszahlung, ob das Konto genügend Deckung aufweist und verweigere die Auszahlung, falls dies nicht der Fall ist.
4. Teste dein erweitertes Programm

Vererbung

Durch ein cleveres Design können wir bereits existierenden Code wiederverwenden und ihn, falls nötig, erweitern.

Wir sprechen von Vererbung wenn wir eine Klasse oder Teile davon in einer anderen Klasse (Subklasse, Kindklasse) wiederverwenden.

Wenn eine Subklasse von einer Elternklasse (Superklasse) abgeleitet wird („erbt“), dann stehen der Subklasse grundsätzlich alle Eigenschaften und Methoden der Elternklasse zur Verfügung.

Wir können in der Kindklasse aber auch eine unabhängige Variable/Methode mit dem gleichen Namen definieren, mit der wir die gleichnamige Variable/Methode der Elternklasse überschreiben.

Ebenso können wir in einer Kindklasse auch Variablen und Methoden definieren, die in der Elternklasse nicht vorkommen.

Falls nötig können wir mit `super().<variable>` oder `super().<methode()>` explizit auf eine Variable oder Methode der Elternklasse zugreifen.

Suche nach Variablen und Methoden in der Klassenhierarchie

In diesem Zusammenhang lautet die wichtigste Regel.

- Greifen wir auf eine Variable oder Methode einer Klasse zu, dann wird zuerst geschaut, ob die Variable bzw. Methode in dieser Klasse definiert ist.
- Ist sie es nicht, wird in ihrer unmittelbaren Superklasse nach der Variablen bzw. Methode gesucht.
- Dieser Prozess wird so lange fortgesetzt, bis die oberste Ebene der Klassenhierarchie erreicht wird. Erst wenn dort keine Variable oder Methode mit dem angegebenen Namen gefunden wird, gibt Python eine Fehlermeldung aus.

Beispiel 4

```
1  class Person:
2
3      def __init__(self, name, vorname, geb_dat):
4          self.name = name
5          self.vorname = vorname
6          self.geb_dat = geb_dat
7
8      def __str__(self):
9          txt = 'Name: {0}\n'.format(self.name)
10         txt += 'Vorname: {0}\n'.format(self.vorname)
11         txt += 'Geburtsdatum: {0}\n'.format(self.geb_dat)
12         return txt
13
14     def change_name(self, neuer_name):
15         self.name = neuer_name
```

```
18 class Angestellte(Person):
19
20     def __init__(self, name, vorname, geb_dat, abt, lohn):
21         self.abteilung = abt
22         self.lohn = lohn
23         super().__init__(name, vorname, geb_dat) # ruft
               Eltern-Konstruktor auf
24
25     def __str__(self):
26         txt = super().__str__() # ruft __str__() von
               Elternklasse auf
27         txt += 'Abteilung: {0}\n'.format(self.abteilung)
28         txt += 'Lohn: {0}\n'.format(self.lohn)
29         return txt
```

```
31 class Lehrling(Person):
32
33     def __init__(self, name, vorname, geb_dat, lehrjahr,
           lohn):
34         self.lehrjahr = lehrjahr
35         self.lohn = lohn
36         super().__init__(name, vorname, geb_dat)
37
38     def __str__(self):
39         txt = super().__str__()
40         txt += 'Lehrjahr: {0}\n'.format(self.lehrjahr)
41         txt += 'Lohn: {0}\n'.format(self.lohn)
42         return txt
```

```
44  # Testcode
45  A100 = Angestellte('Meier', 'Ada', '12.7.1999',
        'Marketing', 8000)
46  A100.change_name('Löffler')
47  L073 = Lehrling('Kurz', 'Tim', '24.3.2008', 1, 800)
48
49  print(A100)
50  print(L073)
```

Weitere OOP-Begriffe:

Polymorphie

Polymorphie (*Vielgestaltigkeit*) ist ein OOP-Konzept, das es erlaubt, denselben Bezeichner (für Variablen oder Funktionen) in unterschiedlichen Klassen zu verwenden.

Ist `objA` ein Objekt der Klasse `ClassA` und `objB` ein Objekt der Klasse `ClassB`, so sind

- `objA.x` und `objB.x` oder
- `objA.mymethod()` und `objB.mymethod()`

möglich, da jedes Objekt weiss, welche Werte oder Methoden ihm zugewiesen wurden.

Überschreiben (von Methoden)

Wenn man eine Kindklasse von einer Mutterklasse ableitet, so können die Methoden der Mutterklasse verwendet werden, ohne sie noch einmal implementieren zu müssen.

Möchte man aber, dass sich die Methode in der Kindklasse anders verhält, so kann man sie in der Kindklasse neu implementieren, weshalb man von Überschreiben (der Methode aus der Mutterklasse) spricht.

Bemerkung: Das Überschreiben einer Methode stellt eine Form der Polymorphie, da derselbe Name (der Methode) in einer anderen Klasse (Kindklasse) verwendet wird.

Überladen von Operatoren und Funktionen

Das Überladen ist eine weitere Form von Polymorphie, bei dem ein Sprachelement (Operator, Funktion) je nach Kontext unterschiedliche Bedeutungen haben kann.

Beispiel: In Python kann der Operator "+" sowohl für Zahlen als auch für Zeichenketten verwendet werden.

- `5 + 3` `# => 8`
- `'a' + 'b'` `# => 'ab'`

Da der Operator "+" je nach Kontext unterschiedliche Bedeutungen haben kann, ist er (mit diesen Bedeutungen) *überladen*. Ein anderes Beispiel fürs Überladen ist die Funktion len(), mit der man die Länge von Listen, Zeichenketten und anderen aufzählbaren Datentypen ausgeben kann.