Einführung in Python (Teil 3): Klassen & Objekte

Ein spielerischer Zugang

Gregor Lingl

0. Einleitung

In diesem 3. Teil der „Einführung in Python“ möchte ich Ihnen die grundlegenden Sprachelemente für objektorientierte Programmierung nahe bringen. Objektbasierte Programmierung, d. h. Programmierung mit Objekten, betreibt man in Python von Anfang an, denn in Python „ist (fast) alles ein Objekt“.

Objektorientierte Programmierung (OOP) umfasst die Programmierung benutzerdefinierter Klassen. Darüber hinaus aber auch die Techniken der objektorientierten Analyse (OOA) und des objektorientierten Entwurfs (OOD). Die beiden letztgenannten Bereiche werden in diesem Artikel nicht angesprochen.

Mir geht es in diesem Artikel darum zu zeigen, wie man in Python Klassen mit ihren Attributen programmiert und die für die OOP wesentlichen Techniken der Vererbung und des Polymorphismus von Methoden realisiert.

Ich wähle dafür einen Zugang, der an Hand von einfachen „Spielzeug“-Beispielen vollständig mit dem interaktiven Python Interpreter nachvollzogen werden können. Dies ist zugleich eine didaktisch ausgeklügelte Zusammenfassung einiger wichtiger Grundbegriffe und Grundtechniken des objektorientierten Programmierens mit dem Ziel das Verständnis dieser Begriffe zu fördern. Um die Anwendbarkeit dieses Beispiels geht es mir hier nicht. (Ich habe dieses Beispiel in ähnlicher Form in meinem Buch „Python für Kids“ verwendet.)

Unsere „Spielzeug“-Objekte werden verschiedene Typen von Boten und Agenten sein. Also setzten sie sich an den Computer, starten sie IDLE oder eine andere Python-IDE mit einem interaktiven Interpreter und spielen sie mit!

1. Was sind Objekte?

Objekte sind aktive Datenstrukturen: Objekte wissen was und können was. Das heißt, Objekte haben gewisse Daten gespeichert und sie haben Methoden, mit denen sie diese Daten manipulieren können. Diese Methoden können gegebenenfalls noch weitere Informationen, die ihnen beim Aufruf übergeben werden nützen.

Wie kommt man zu Objekten? Man erzeugt sie als Instanzen einer Klasse. Das geschieht durch Aufruf eines sogenannten Konstruktors der Klasse. Klassen beschreiben, was ihre Instanzen, die Objekte wissen und können werden. Man kann sich Klassen gewissermaßen als Objektfabriken vorstellen.

Was das konkret bedeutet und wie das in Python geschieht, wird in dem folgenden Beispiel mit Boten und Agenten vorgeführt und erklärt. Sie können das Ganze mit dem Python-Interpreter im Direktmodus nachvollziehen.

2. Definition einer Klasse: Bote

Boten sind Leute, die eine Botschaft überbringen oder eine Meldung ausgeben. Wir wollen eine Klasse Bote definieren, deren Objekte das tun können.

Zur Definition einer Klasse wird in Python die class-Anweisung verwendet. Das ist eine der 6 zusammengesetzten Anweisungen in Python (`if`, `while`, `for`, `try`, `def`, `class`). So definieren wir eine Klasse:

```
>>> class Bote:
    pass
```

Wie bei jeder zusammengesetzten Anweisung wird die Kopfzeile mit einem Doppelpunkt abgeschlossen und der Anweisungskörper eingerückt. Hier besteht er zunächst nur aus der „Nullanweisung“ `pass`. Somit können wir nun bereits Boten erzeugen:

```
>>> fritz = Bote()
```

Der Konstruktor der Klasse `Bote` wird so aufgerufen: `<Klassenname>()`. Das heißt, wie eine Funktion, deren Name der Klassenname ist. Hier müssen beim Konstruktor-Aufruf keine Argumente eingesetzt werden.

```
>>> franz = Bote()
>>> fritz
<__main__.Bote instance at 0x00B08710>
>>> franz
<__main__.Bote instance at 0x00B30D78>
```

Wir haben hier zwei verschiedene „Instanzen“ der Klasse Bote. (Sie sind an verschiedenen Stellen im Arbeitsspeicher untergebracht.) Oder?

```
>>> fritz is franz
False
```

Tatsächlich zwei verschiedene Botenobjekte, doch diese Boten wissen nichts und können nichts. Eindeutig zu wenig für brauchbare Boten.

3. Methoden

Also bringen wir den Boten bei, wenigstens eine einfache Meldung auszugeben. Wir definieren eine Methode für die Klasse Bote:

```
>>> class Bote:
        def melde(self):
            print "Ich melde: Hallo!"
>>> fritz = Bote()
>>> fritz.melde()
Ich melde: Hallo!
>>> franz = Bote()
>>> franz.melde()
Ich melde: Hallo!
```

Die Methodendefinition sieht aus wie eine Funktionsdefinition, jedoch hat sie neben dem Umstand, dass sie innerhalb einer class-Anweisung steht noch eine weitere Besonderheit: einen Parameter, der hier `self` heißt. Beim Aufruf der Methode wird für diesen Parameter nichts eingesetzt – vom Programmierer! Hinter den Kulissen aber doch: `self` verweist auf das jeweilige Objekt selbst:

4. Kleiner Exkurs über self

Sehen wir uns das in einem kleinen Exkurs rasch einmal an: (Denken Sie daran, dass sie, wenn sie die IDLE verwenden, alte Eingaben im Shell-Fenster mit (allenfalls mehrfachem) Alt - P zurückrufen und danach editieren können)

```
>>> class Bote:
     def melde(self):
         print self, "meldet:","Hallo!"
>>> fritz=Bote()
>>> franz=Bote()
>>> fritz
<__main__.Bote instance at 0x00B30E68>
>>> franz
<__main__.Bote instance at 0x00B08710>
>>> fritz.melde()
<__main__.Bote instance at 0x00B30E68> meldet: Hallo!
>>> franz.melde()
<__main__.Bote instance at 0x00B08710> meldet: Hallo!
```

Wir sehen, dass `melde()` tatsächlich so funktioniert, dass für den Parameter `self` beim Aufruf `fritz.melde()` automatisch das Objekt `fritz` und beim Aufruf `franz.melde()` das Objekt `franz` eingesetzt wird.

Ohne nähere Erklärung zeige ich, dass man sich davon auch so überzeugen kann:

```
>>> Bote.melde(franz)
<__main__.Bote instance at 0x00B08710> meldet: Hallo!
```

Der Aufruf der Objekt-Methode `franz.melde()` liefert das gleiche Ergebnis wie der Aufruf der Klassen-Methode `Bote.melde()`, wo für den Parameter `self` das Objekt `franz` als Argument eingesetzt wird.

In der Folge werden wir `self` nicht mehr direkt in `print`-Anweisungen schreiben. Ausgaben dieser Art sind ja wohl nicht sehr attraktiv.

5. Methoden mit Parametern

Der Name `self` für den ersten Parameter jeder Methode ist nicht von der Sprachdefinition vorgeschrieben, sondern eine in der Python-Welt durchgängig eingehaltene Konvention.

Kümmern wir uns weiter um unsere Boten – wie sie vor dem Exkurs ausgesehen haben. Sie kennen (bis jetzt) nur eine Methode, etwas zu melden, und melden immer dasselbe. So kann das nicht bleiben. Sie sollen uns doch eine Botschaft überbringen. Also verbessern wir die Methode melde:

```
>>> class Bote:
        def melde(self, text):
```

glingl@aon.at

```
                print "Ich melde:", text
>>> herold = Bote()
>>> herold.melde("Feuer am Dach!")
Ich melde: Feuer am Dach!
>>> herold.melde("Tornado im Anzug!")
Ich melde: Tornado im Anzug!
```

Das ist schon ein Fortschritt. Der Methode `melde` kann eine Information als Argument übergeben werden, nämlich was gemeldet werden soll. Bedauerlich ist allerdings, dass sich diese Boten nicht merken, was sie melden sollen. Für jede Meldung muss man es ihnen vorsagen. Besser wäre doch, sie könnten einen Meldungstext – ja! – speichern!

6. Instanz-Variable

Das Problem dabei ist, dass wir ja aus einer Klasse `Bote` viele Boten-Objekte machen können oder wollen. Und jedes soll sich seinen eigenen Meldungstext merken. Das heißt, jedes Objekt (jede Instanz von `Bote`) braucht seine eigene Variable für den Meldungstext.

Das geht leicht, und zwar so:

```
>>> class SchlauerBote:
        def merke(self, text):
                self.botschaft = text
        def melde(self):
                print "Ich melde:", self.botschaft
>>> schlaumeier = SchlauerBote()
>>> schlaumeier.merke("Die Aktien steigen!")
>>> schlaumeier.melde()
Ich melde: Die Aktien steigen!
>>> vifzac = SchlauerBote()
>>> vifzac.merke("Das Wetter wird kalt!")
>>> vifzac.melde()
Ich melde: Das Wetter wird kalt!
>>> schlaumeier.melde()
Ich melde: Die Aktien steigen!
>>>
```

Hier ist das von `melde()` gebrauchte `self.botschaft` das erste Mal `schlaumeier.botschaft` und das zweite Mal `vifzac.botschaft` – je nach dem was `self` gerade ist.

Wir sagen: Der Name `botschaft` ist an das Objekt gebunden. Jedes Objekt hat einen eigenen Namen `botschaft`, und jeder dieser Namen kann auf einen anderen Wert verweisen. Die `botschaft` von `schlaumeier` verweist auf einen anderen String als die `botschaft` von `vifzac`. Kurz gesagt: Jeder Bote kennt und merkt sich seine eigene Botschaft!

Wir bezeichnen solche Variablen, die zu einem bestimmten Objekt gehören, als Instanzvariable .

Zusammenfassend nennt man die Instanzvariablen und die Methoden eines Objektes seine Attribute.

Der Wert der Instanzvariablen `botschaft` kann jederzeit mit der Methode `merke` verändert werden:

```
>>> vifzac.merke("Das Wetter wird warm!")
>>> vifzac.melde()
Ich melde dir: Das Wetter wird warm!
>>>
```

In der Tat erlaubt es Python sogar, den Wert einer Instanzvariablen direkt anzusehen:

```
>>> vifzac.botschaft
'Das Wetter wird warm!'
>>>
```

Und er kann auch direkt durch eine Wertzuweisung verändert werden:

```
>>> vifzac.botschaft = "Sauwetter!"
>>> vifzac.melde()
Ich melde dir: Sauwetter!
```

Beides ist in der Technik des objektorientierten Programmierens nicht ratsam. Statt dessen greift man auf die Werte von Instanzvariablen normalerweise über `get`- und `set`-Methoden zu. `set`-Methoden funktionieren so wie die Methode `merke()`, `get`-Methoden liefern Werte von Instanzvariablen zurück, im Prinzip so wie `melde()` – nur mit `return`- statt `print`-Anweisungen. Näher will ich auf diese Thematik hier nicht eingehen.

7. Konstruktor

Jetzt wollen wir noch erreichen, dass unsere Boten ihre Namen kennen und sich auch mit ihrem Name melden. Nahe liegender weise müssen wir dazu eine Instanzvariable Name einführen. Wir wollen, dass jeder Bote schon gleich bei seiner Entstehung seinen Namen erfährt.

Für den Konstruktor von Klassen benützt Python einen standardisierten Namen und zwar `__init__`. Namen, die wie dieser mit je zwei Unterstrichen beginnen und enden, werden in Python *special names* genannt. Die Methode `__init__()` wird automatisch aufgerufen, wenn ein Objekt einer Klasse erzeugt wird. Der Konstruktor kann Instanzvariablen erzeugen und gleich mit einem Wert versehen. Beispiel:

```
>>> class SchlauerBote:
      def __init__(self, name):
            self.name = name
      def merke(self,text):
            self.botschaft = text
      def melde(self):
            print self.name, "meldet:", self.botschaft
```

Der Konstruktor hat – das wissen wir schon von weiter oben - die besondere Eigenschaft, dass er bei der Konstruktion eines Objekts nicht mit seinem Namen aufgerufen wird. Statt dessen wird der Klassenname verwendet mit der Paramterliste des Konstruktors, doch, wie bei jeder anderen Methode auch, ohne für `self` einen Argumentwert einzusetzen. Wir setzten für `name` den String `"Karl"` ein:

```
>>> karl = SchlauerBote("Karl")
>>> karl.merke("Vorwärts! Und nicht vergessen!")
>>> karl.melde()
Karl meldet: Vorwärts! Und nicht vergessen!
```

8. Vererbung

Jetzt machen wir Folgendes:

```
>>> class SehrSchlauerBote(SchlauerBote):
      def merke_dazu(self,zusatz):
            self.botschaft=self.botschaft+" "+zusatz

>>> angelo = SehrSchlauerBote("Angelo")
>>> angelo.merke("Kalt wird's!")
>>> angelo.melde()
Angelo meldet Kalt wird's!
>>> angelo.merke_dazu("Warm anziehen!")
>>> angelo.melde()
Angelo meldet: Kalt wird's! Warm anziehen!
```

Der sehr schlaue Bote kann merken und melden. Obwohl nichts Derartiges in seiner Klassendefinition steht. Weil er ja alles kann, was ein schlauer Bote kann. Er ist ein schlauer Bote. Aber er ist auch ein sehr schlauer Bote, denn er kann noch mehr:

```
>>> angelo.merke_dazu("Warm anziehen!")
>>> angelo.melde()
Ich melde dir: Kalt wird's! Warm anziehen!
```

Damit wäre ein nur schlauer Bote überfordert, denn:

```
>>> merkur = SchlauerBote("Merkur")
>>> merkur.merke("Stau auf allen Autobahnen!")
>>> merkur.merke_dazu("Reise verschieben!")
Traceback (most recent call last):
  File "<pyshell#62>", line 1, in -toplevel-
    merkur.merke_dazu(" Reisen verschieben!")
AttributeError: SchlauerBote instance has no attribute 'merke_dazu'
```

Wir haben die Klasse `SehrSchlauerBote` von der Klasse `SchlauerBote` abgeleitet: Damit erbt die Klasse `SehrSchlauerBote` alle Methoden von `SchlauerBote` und kennt noch eine zusätzliche: `merke_dazu`. Schlaue Boten wissen davon aber natürlich nichts.

Die Syntax für die Definition von Unterklassen sieht so aus:

```
class <Klassenname>(<Oberklasse1>,<Oberklasse2>,...):
      ... Körper der Klassendefinition
          (Methodendefinitionen etc.)
```

(Python gestattet sogar mehrfache Vererbung.)

9. Initialisierung von Instanz-Variablen im Konstruktor

Einen ernsten Mangel haben unsere Botenklassen noch, der sich so zeigt:

```
>>> zweistein = SehrSchlauerBote("2Stein")
>>> zweistein.melde()
2Stein meldet:
Traceback (most recent call last):
  File "<pyshell#68>", line 1, in -toplevel-
    zweistein.melde()
  File "<pyshell#51>", line 7, in melde
    print self.name, "meldet:", self.botschaft
AttributeError: SehrSchlauerBote instance has no attribute
'botschaft'
```

Die Instanzvariable `botschaft` von `zweistein` verweist auf nichts. Sie ist noch nicht initialisiert. Das ist kein guter Programmierstil. Initialisierung von Instanzvariablen gehört in den Code eines Konstruktors. Das erledigen wir gleich für SchlauerBote ...:

```
>>> SchlauerBote
<class __main__.SchlauerBote at 0x00A2C110>
```

... und schreiben eine verbesserte Version der Klassendefinition:

```
>>> class SchlauerBote:
        def __init__(self, name, botschaft="Hi!"):
              self.name = name
              self.botschaft = botschaft
        def merke(self, text):
              self.botschaft = text
        def melde(self):
              print self.name, "meldet:", self.botschaft
```

Nun haben wir also wieder eine Klasse SchlauerBote:

```
>>> SchlauerBote
<class __main__.SchlauerBote at 0x00A7CC78>
```

… aber eine andere, neu definierte. Nun haben schlaue Boten eine vor eingestellte Minimalbotschaft, können bei der Konstruktion aber auch eine andere erhalten. Damit die Klasse `SehrSchlauerBote` nicht mehr von der alten, sondern von dieser neuen erbt, müssen wir sie nochmals definieren:

```
>>> class SehrSchlauerBote(SchlauerBote):
        def merke_dazu(self, zusatz):
              self.botschaft += " " + zusatz
>>> zweistein = SehrSchlauerBote("2Stein")
>>> zweistein.melde()
2Stein meldet: Hi!
```

Die voreingestellte Meldung ist nicht gerade sehr schlau! Aber besser als ein Programmabsturz. (Hinter den Kulissen ist es ja doch schlau, denn der Konstruktor von `SehrSchlauerBote` ruft automatisch den Konstruktor von `SchlauerBote` auf!) . Oder:

```
>>> genius = SehrSchlauerBote("Genie", "Salve!")
>>> genius.melde()
Genie meldet: Salve!
>>>
```

10. Konstruktor der Unterklasse

Eine weitere Demo: Wir wollen uns eine Klasse `FreundlicherBote` machen, die von `SchlauerBote` abgeleitet ist. Bei der Erzeugung eines freundlichen Boten soll ein Grußtext für ihn festgelegt werden. Das muss irgendwie in seinen Konstruktor rein:

```
>>> class FreundlicherBote(SchlauerBote):
        def __init__(self, name, grusstext):
              SchlauerBote.__init__(self, name)
              self.gruss = grusstext
        def gruesse(self):
              print self.gruss
>>> sunny = FreundlicherBote("Sanni","Schönen Tag wünsch ' ich!")
>>> sunny.gruesse()
Schönen Tag wünsch' ich!
>>> sunny.melde()
Sanni meldet: Hi!
```

`FreundlicherBote` hat einen eigenen Konstruktor, da er mit einem Grußtext aufgerufen wird und eine Instanzvariable initialisieren muss. Damit aber auch die Initialisierungsschritte von `SchlauerBote` gemacht werden, muss zusätzlich der Konstruktor der Oberklasse aufgerufen werden. (Hier ist es so gemacht, dass `FreundlicherBote` in jedem Fall den Default-Wert für die Botschaft verwendet.)

Beachten Sie dazu: Das Objekt, das erzeugt und initialisiert wird, ist `self`. Die Form des Aufrufes des Konstruktors der Oberklasse ist nicht wie üblich

```
self.methodenname(argumente)
```

sondern:

```
SchlauerBote.__init__( ... )
```

Dieser Aufruf braucht aber auch die Information, welches Objekt initialisiert werden soll, und daher muss ihm `self` als erstes Argument übergeben werden:

```
SchlauerBote.__init__(self, ... )
```

Damit haben wir mit dem Konstruktor etwas gemacht, was in der objektorientierten Programmierung ganz allgemein mit Methoden gemacht werden kann:

11. Überschreiben von Methoden (Polymorphismus)

Sehen wir uns das in der nächsten Übung an, die wir mit der Klasse `FreundlicherBote` anstellen:

```
>>> class FreundlicherBote(SchlauerBote):
        def __init__(self, name, grusstext):
              SchlauerBote.__init__(self, name)
              self.gruss = grusstext
        def gruesse(self):
              print self.gruss
        def melde(self):
              self.gruesse()
              SchlauerBote.melde(self)
>>> sunny = FreundlicherBote("Sanni", "Schönen Tag wünsch ' ich!")
>>> sunny.melde()
Schönen Tag wünsch' ich!
Sanni meldet: Hi!
```

Hier haben wir die Methode `melde()` neu definiert, um zu erreichen, dass jeder freundliche Bote bei jeder Meldung vorher automatisch grüßt. Daran sind zwei Dinge bemerkenswert:

- Diesmal wurde die Methode `melde` als Klassenmethode der Oberklasse `SchlauerBote` aufgerufen und hat dementsprechend `self` als Argument übergeben bekommen. Warum? In der Definition der Methode `melde` in der Klasse `FreundlicherBote` bedeutet der Name `self.melde` die Methode `melde` von `FreundlicherBote`. An dieser Stelle wird aber ein anderes `melde` gebraucht, nämlich die in der Oberklasse definierte Methode.
- In der Klasse `FreundlicherBote` wurde damit die von `SchlauerBote` geerbte Methode `melde` mit einer neuen Version von `melde` überschrieben. Objekte der Klasse `SchlauerBote` verfügen nach wie vor über die alte Methode `melde`.

```
>>> sunny.merke("OOP ist eine steile Sache!")
>>> sunny.melde()
Schönen Tag wünsch' ich!
Sanni meldet: OOP ist eine steile Sache!
>>> tim = SchlauerBote("Tim")
>>> tim.merke("Morgen passiert's!")
>>> tim.melde()
Tim meldet: Morgen passiert's!
```

12. Interaktion von Objekten

Abschließend möchte ich noch zeigen, wie Objekte miteinander kommunizieren können. Hier einfacher: wie Boten miteinander Informationen austauschen können:

Ich möchte nämlich unsere schlauen Boten zu Agenten ausbilden.

Selbstverständlich ist ein Agent ein schlauer Bote. Wir werden die Agentenklasse also von unseren schlauen Boten ableiten. Aber Agenten können noch mehr – sie geben Botschaften an andere Boten weiter und sind auch im Stande, andere Boten zu belauschen!

```
>>> class Agent(SchlauerBote):
        def weitersagen(self, bote):
              bote.merke(self.botschaft)
        def belausche(self,bote):
              self.botschaft = bote.botschaft
```

In der letzten Zeile haben wir direkt auf die Instanzvariable `botschaft` der Boten zugegriffen. Zugegeben: das ist nicht die feine englische Art. Aber Agenten müssen eben manchmal, wenn sie erfahren wollen, was nicht für sie bestimmt ist, halb legale Wege gehen. (In einer Sprache mit Unterscheidung von privaten und öffentlichen Attributen könnten wir die botschaft von Agenten abhörsicher machen.)

So, jetzt kommen zwei Agenten daher:

```
>>>james = Agent("Bond 007")
>>>austin = Agent("Powers")
>>>james.melde()
Bond007 meldet: Hi!
```

und gehen gleich an die Arbeit!

```
>>> james.belausche(tim)
>>> james.melde()
Bond 007 meldet: Morgen passiert's!
>>> austin.melde()
Powers meldet: Hi!
>>> james.weitersagen(austin)
>>> austin.melde()
Powers meldet: Morgen passiert's!
>>>
```

Nicht schlecht! Wir haben Objekte erzeugt, die offenbar untereinander Informationen austauschen können! `james` hat `austin` verraten, was `tim` zu melden hatte!

Entscheidend für diese Fähigkeit der Zusammenarbeit von Boten (Objekten) ist, dass Objekte als Argumente an Methoden übergeben werden können: James kann mit Austin zusammenarbeiten, nämlich ihm etwas weitersagen, weil in James' Methodenaufruf von weitersagen `austin` als Argument eingesetzt wurde!

Anmerkung: Python gestattet auch die Verwendung privater Attribute in einer eingeschränkten Form. Auf dieses Feature wurde in diesem Artikel nicht eingegangen.