Schnelleinführung in die objektorientierte Programmierung in Python

Dieser Artikel bietet eine kurze Einführung in die [wikipedia]objektorientierte Programmierung[/wikipedia] in Python. Alle Beispiele nehmen Bezug auf Python 2.x, sollten allerdings mit geringen Aufwand zu Version 3 übertragbar sein.
Da auf eine detaillierte Erklärung der Begriffe verzichtet wird und eher die gezeigten Codes erklärend wirken sollen, richtet sich sich dieser Artikel eher an Programmierer mit fortgeschrittenen Kenntnissen.

Inhaltsverzeichnis

Die Klasse

Inspiriert durch den Artikel [wiki]OOP Objektorientierte Programmierung in PHP - Part 1[/wiki] werden wir hier ebenfalls eine Klasse für Raumschiffe erstellen und diese im Laufe des Artikels erweitern.

Quellcode

```
1. class Raumschiff(object):
2. pass
3. schiff = Raumschiff()
```

In den ersten beiden Zeilen definieren wir die noch funktionslose Klasse `Raumschiff`, die von `object` erbt (mehr dazu später). In letzten Zeile wird diese instanziert und die neue Instanz wird der Variablen `schiff` zugewiesen.

Methoden, der Konstruktor und Attribute

Erstellung von Methoden

Quellcode

```
1. class Raumschiff(object):
2. def fliegen(self):
3. print 'Das Raumschiff fliegt davon!'
4. schiff = Raumschiff()
5. schiff.fliegen()
```

Offensichtlich werden Methoden in der gewohnten Syntax für Funktionen geschrieben, wobei der einzige Unterschied der erforderliche Parameter `self` darstellt, der eine Referenz auf die aktuelle Instanz enthalten wird. Der Zugriff auf die Methode `fliegen()` erfolgt wie bei eingebauten Objekten über einen Punkt, hierbei muss nie der Wert für `self` übergeben werden.

Konstruktor und Destruktor

`__init__` und `__del__` sind besondere Methoden, sie heißen Konstruktor und Destruktor und werden nur bei der Erzeugung und Löschung einer Instanz aufgerufen.

Quellcode

```
1. class Raumschiff(object):
2. def __init__(self): # Wird aufgerufen, wenn die Instanz erstellt wird
3. print 'Das Raumschiff wurde soeben gebaut.'
4. def __del__(self): # Wird aufgerufen, wenn die Instanz entfernt wird
5. print 'Das Raumschiff wurde zerstört! :('
6. def fliegen(self):
7. print 'Das Raumschiff fliegt davon!'
8. schiff = Raumschiff()
9. schiff.fliegen()
10. del schiff # Das Raumschiff wird in eine Weltraumschlacht verwickelt und zerstört
```

Alles anzeigen

Attribute

Die Hauptaufgabe des Konstruktor ist es, die Instanz auf die Verwendung vorzubereiten. Dazu zählt anders als in anderen Sprachen auch das setzen von Attributen. Der Zugriff auf Attribute erfolgt ebenfalls über einen Punkt.
Folgender Code erweitert unser Raumschiff um einen Treibstofftank und einen Anstrich der Außenhülle.

Quellcode

```
1. class Raumschiff(object):
2. def __init__(self, farbe):
3. self.treibstoff = 100
4. self.farbe = farbe
5. def fliegen(self):
6. if self.treibstoff > 0:
7. self.treibstoff -= 1 # Ein Flug verbraucht eine Einheit Treibstoff
8. print 'Das Raumschiff fliegt davon!'
9. else:
10. print 'Der Treibstofftank ist leer.'
11. schiff = Raumschiff('rot')
12. schiff.fliegen()
13. print schiff.treibstoff, schiff.farbe # 99 rot
```

Alles anzeigen

Wie oben erwähnt wird auf die aktuelle Instanz über `self` zugegriffen. Der Umgang mit Attributen analog zu Funktionen und Methoden identisch zu variablen. Aus diesem Code wird ebenfalls ersichtlich, dass Parameter für den Konstruktor bei der Instanzierung übergeben werden.

Private Methoden und Attribute, Setter und Getter

Es kann sinnvoll sein Methoden und Attribute eines Objekts vor der »Außenwelt« zu verbergen, um die Konsistenz des Objekts zu wahren. Dies wird in Python durch die Benennung des Elements impliziert:

- *Öffentliche Methoden und Attribute* können von außen aufgerufen, gelesen und geschrieben werden. Sie werden nicht besonders gekennzeichnet.
- *Geschützte Methoden und Attribute* unterscheiden sich nicht von öffentlichen Methoden. Es ist eher eine Konvention unter Programmieren, dass diese nicht von außen verändert werden sollen. Sie werden durch einen führenden Unterstrich gekennzeichnet.
- *Private Methoden und Attribute* sind von außen nicht sichtbar, man kann sie daher weder ausführen, noch lesen, noch schreiben. Sie werden durch zwei führende Unterstriche gekennzeichnet.

An unserem Beispiel darf der Füllstand des Treibstofftanks von außen nicht direkt und ohne Prüfung modifiziert werden, da ein unachtsamer Programmierer so die Beschränkungen der Tankgröße umgehen könnte. Daher setzen wir das Attribut `treibstoff` als privat. `farbe` *sollte* eigentlich nicht verändert werden, daher setzen wir dieses Attribut als geschützt.

Quellcode

```
class Raumschiff(object):
def __init__(self, farbe):
self.__treibstoff = 100
self._farbe = farbe
def fliegen(self):
if self.__treibstoff > 0:
self.__treibstoff -= 1
print 'Das Raumschiff fliegt davon!'
else:
print 'Der Treibstofftank ist leer.'
schiff = Raumschiff('rot')
schiff.fliegen()
print schiff._farbe # Ein Zugriff auf __treibstoff ist nicht möglich
```

Alles anzeigen

Getter

Nun wollen wir aber, dass man weiterhin den Füllstand des Treibstofftanks auslesen kann. Hierfür schreiben wir eine Getter-Methode, die diesen Wert zurück gibt:

Quellcode

```
class Raumschiff(object):
def __init__(self, farbe):
self.__treibstoff = 100
self._farbe = farbe
def get_treibstoff(self):
return self.__treibstoff
# Definition von fliegen()
schiff = Raumschiff('rot')
schiff.fliegen()
print schiff.get_treibstoff() # 99
```

Alles anzeigen

Setter

Ein Raumschiff, dessen Tank man nicht auffüllen kann, ist auf Dauer nicht viel wert. Wir schreiben nun eine Setter-Methode, die einen neuen Wert für `treibstoff` setzt, nachdem sie diesen validiert hat:

Quellcode

```
class Raumschiff(object):
def __init__(self, farbe):
self.__treibstoff = 100
self._farbe = farbe
def get_treibstoff(self):
return self.__treibstoff
def set_treibstoff(self, treibstoff):
if type(treibstoff) == int and treibstoff >= 0 and treibstoff <= 100:
self.__treibstoff = treibstoff
# Definition von fliegen()
schiff = Raumschiff('rot')
schiff.fliegen()
schiff.set_treibstoff(100)
print schiff.get_treibstoff()
```

Alles anzeigen

Der Wert von `treibstoff` wird nun nur modifiziert, wenn dieser zur Tankgröße des Raumschiffs passt.

Eine kürzere und schönere Schreibweise für Getter und Setter lässt sich übrigens über Property-Attribute verwirklichen: Sie sind von außen wie normale Attribute zu behandeln und rufen intern die Getter und Setter auf.

Quellcode

```
1. class Raumschiff(object):
2. def __init__(self, farbe):
3. self.__treibstoff = 100
4. self._farbe = farbe
5. def get_treibstoff(self):
6. return self.__treibstoff
7. def set_treibstoff(self, treibstoff):
8. if type(treibstoff) == int and treibstoff >= 0 and treibstoff <= 100:
9. self.__treibstoff = treibstoff
10. treibstoff = property(get_treibstoff, set_treibstoff)
11. # Definition von fliegen()
12. schiff = Raumschiff('rot')
13. schiff.fliegen()
14. schiff.treibstoff = 100
15. print schiff.treibstoff # 100
```

Alles anzeigen

== Verebung ==