Es gibt viele Fragen und Antworten zu Python und wie man gewisse Probleme mit der Programmiersprache Python lösen kann.

Die Wahl der Programmiersprache ist dann eines, aber auch die Methode der Programmierung - ob prozedural oder objektorientiert ist wichtig. Normalerweise fängt ein jeder mit der prozeduralen Programmierung an, d.h. zerlegt seine größeren Programme in kleinere Teile, die sogenannten Funktionen, um Aufgaben zu lösen. Es gibt aber auch den objektorientierten Ansatz, der eigentlich der natürlichere Ansatz ist, zu programmieren, aber doch noch nicht so verbreitet ist.

In diesem Tutorial wird versucht anhand von praktischen aufeinander aufbauenden Codebeispielen, den objektorientierten Ansatz der Programmierung im Allgemeinen - und im speziellen in Python zu erklären. Diese werden nach und nach erstellt und bauen aufeinander auf, so dass am Ende ein vollständig objektorientiertes kleines Beispielprogramm existieren wird mit einer Klasse Sensor und weiteren Unterklassen.

Es wird Kommentare zu dem Tutorial geben, die aber bitte alle **nicht in diesem Thread posten**, denn das dient absolut nicht der Übersichtlichkeit und dem Sinn eines Tutorials. Deshalb habe ich parallel den folgenden Diskusstionsthread erstellt mit der Bitte **wirklich nur da Kommentare zu posten**. Ergebnisse von dort werden sicherlich hier im Tutorial einfliessen.

Anlass für dieses Tutorial ist dieser Thread, der exemplarisch zeigt, warum die objektorientierte Programmierung, wenn er auch zuerst aufwändig und kompliziert aussieht, sich dann doch in vielen Fällen als der bessere und flexiblerer Programmieransatz herausstellt.

Python bekam leider erst wie auch andere Programmiersprachen wie z.B. Perl zu einem späteren Zeitpunkt objektorientierte Fähigkeiten. Dadurch mussten gewisse Kompromisse in der Syntax und Semantik der Sprache eingegangen werden und gewisse objektorientierte Konzepte sind deshalb etwas komplizierter in Python zu kodieren als es eigentlich nötig wäre. Neuere objektorientierte Programmiersprachen wie z.B. Java, die von Anfang an objektorientiert konzipiert wurden, sind da einfacher zu programmieren. Mit Python 3 gibt es da ein paar Verbesserungen wie z.B. die Einführung der Methode super(). Das Tutorial basiert noch auf Python 2.7.

Was sind wesentliche Unterschiede zwischen prozeduraler und objektorientierter Programmierung?

In der prozeduralen Programmierung (zukünftig abgekürzt mit PP) besteht ein Programm aus Daten und Funktionen (def), die diese Daten bearbeiten und verändern. Die Daten werden als Parameter zu Funktionen mitgegeben bzw werden durch die Funktionen berechnet und an die aufrufenden Funktionen zur weiteren Verarbeitung zurückgeliefert. Dabei sind die Daten und die Berechnung von Daten unabhängig und in einem Programm in der Regel sehr weit zerstreut.

In der objektorientierten Programmierung (OOP) dagegen sind die Daten und die Berechnungen der Daten in einem Objekt zusammen gekapselt (Encapsulation). Ein Objekt ist eine Repräsentation einer Klasse (class in Python) und hat sogenannte Methoden (def innerhalb einer class), die den Funktionen in der PP entsprechen, Berechnungen und Veränderungen eines Objektes vornehmen.

In der OOP gibt es den Begriff der Vererbung (Inheritance). Es gibt keine Entsprechung in der PPt. D.h. man kann Klassen bilden, die hierarchisch voneinder abhängen. Dieses hat den Vorteil, dass gemeinsame Eigenschaften (entweder Daten oder Methoden) nur einmal definiert bzw implementiert werden müssen. Z.B. kann man eine Oberklasse Sensor definieren, die allgemeine Daten und Methoden eines Sensors enthält. Unterklassen davon sind dann spezielle Sensoren, die weitere oder andere Eigenschaften dazubringen. Das können z.B. weitere Daten sein oder auch die Art und Weise wie die Daten gesammelt werden. Deshalb werden Klassen sehr häufig in Klassenbibliotheken benutzt, denn die existierenden Klassen kann der Anwender sehr leicht durch Erstellen von Unterklassen seinen Bedürfnissen anpassen und dabei die Daten und Methoden der Oberklassen mitbenutzen.

Alle Instanzen eine Klasse haben Daten und Methoden. Dabei gibt es welche, die die Klasse ihren Benutzern zur Verfügung stellt und welche, die nur intern von der Klasse oder Unterklassen benötigt und benutzt werden. D.h. es gibt öffentliche(public), geschützte(protected) und private(private) Daten und Methoden. In der PP gibt es die Unterscheidung von globalen und lokalen Variablen.
Ein Benutzer einer Instanz kann nur auf öffentliche Methoden und Daten zugreifen. Der Vorteil davon ist, dass eine Klasse sicherstellen kann, dass keine Variablen bzw Stati der Klasseninstanzenvon irgendwelchem Code einfach so geändert werden können. Dieses ist nur über Methoden der Klasse möglich und somit kann die Klasse immer kontrollieren, dass alle Änderungen erlaubt sind und kein inkonsistenter Datenzustand erzeugt wird. Unterklassen können auf öffentliche und geschützte und die Instanz selbst auf alles zugreifen. In anderen OOP Programmiersprachen gibt es dazu entsprechende Schlüsselworte, die es in Python leider nicht gibt. Es gibt zwar Wege auch in Python dieses zu garantieren, aber in den folgenden Beispielen wird immer davon ausgegangen, dass alle Methoden öffentlich sind und alle Daten privat. Zugriff auf private Daten erfolgt immer über eine Methode die den Variablennamen hat mit einem vorangestellten get. Sollten Variablen direkt geändert werden können so wird in der Klasse eine Methode die den Variablennamen hat mit einem vorangestellten set definiert.

In der OOP gibt es den Begriff der Polomorpie (Polymorphy). Dass bedeutet, dass ein und dieselbe Methode beliebig häufig definiert sein kann. Welche der definierten Methoden aufgerufen wird hängt aber von dem Typ der Instanz ab. D.h. man kann z.B. eine Methode gibLaut() an TierObjekten haben, wobei die Methode bei Hunden ein Bellen erzeugt und bei einem Vogel ein Zwitschern.

Teil 1: Erstellung einer einfachen Sensorklasse

Die Beispielklasse soll einen Sensor repräsentieren, der von einer PI HW Daten ausliest. Im Beispiel wird keine reale HW benutzt sondern das Datensammeln simuliert, damit jeder die Beispielprogramme bei sich ausführen kann.
Was macht den Beispielsensor aus? Er liest die Temperatur von einem Fühler und er speichert die aktuelle Temperatur. D.h. die Klasse braucht ein Datum mit dem Namen *temperature* und eine Methode *getData*, die die Daten aus der HW liest. Zusaetzlich soll der Sensor nicht nur einen Sensor abfragen können, sondern beliebige an bestimmten Pins angeschlossene Temperatursensoren. Zusaetzlich sollte der Sensor noch die Möglichkeit bieten auf weitere Sensordaten zuzugreife, die da sind die letzte gelesene Temperatur und der Pin, der vom Sensor benutzt wird (*getTemperature* und *getPin*). Prinzipiell kann man auch direkt auf die Instanzvariablen zugreifen. Aber es ist in der OOP aus verschiedenen Gründen unüblich dieses zu tun.

Es ist wichtig den Unterschied zwischen der Klasse und einer Klasseninstanz zu verstehen: Eine Klasse ist die Backform um einen Kuchen zu erstellen, und die Klasseninstanz ist dann ein einzelner Zuckerkuchen, ein Sandkuchen, ein Kirschkuchen usw. Abhängig vom Teig, den man benutzt entsteht ein Kuchen, der aus unterschiedlichem Teig besteht. Die Klasse definiert wie Instanzen aussehen und wie man mit ihnen arbeiten kann. Aber zum Arbeiten benötigt man Klasseninstanzen. Deshalb hat jede Klasse immer einen Konstruktor mit dem man Instanzen erzeugen kann und dabei über Parameter das Aussehen der Klasseninstanzen festlegen kann. D.h. man gibt beim Erstellen des Kuchens den Teig als Parameter mit, aus dem der Kuchen erstellt werden soll.

Somit sieht unsere Sensorklasse wir folgt aus (Alle Teile im Code werden im Folgenden genauer beschrieben):

Python

```
class Sensor:
    def __init__(self, pin):
    def getData(self):
    def getPin(self):
    def getTemperature(self):
```

Eine Klassendfinition wird in Python mit dem Keyword *class* begonnen. Danach werden alle Methoden mit dem Keyword *def* definiert, so wie normale Funktionen auch in Python definiert werden. Der Unterschied ist aber, dass immer das erste Argument der Methode *self* ist. self ist immer die eigentliche Instanz der Klasse und dieser Parameter wird automatische von Python beim Aufruf einer Methode auf eine Klasseninstanz eingefügt.

Am Anfang ist die kleine Funktion *simulateData()*definiert um Simulationsdaten für den Sensor zu erzeugen. Danach ist der Konstruktor definiert. Er muss immer __init__ heissen. Der erste Parameter ist self und alle folgenden Parameter dienen dazu, den Sensor zu individualisieren. Der Parameter pin definiert auf welchem Pin der Sensor die HW abfragt.

Python

```
def __init__(self, pin):
        self.pin = pin
        self.temperature=None
```

In dem Konstruktor werden alle Daten, die der Sensor für seine spätere Arbeit benötigt, erzeugt und in der Sensorinstanz selbst abgespeichert. Sensorinstanzen müssen immer mit self. beginnen. D.h. die Zeile

Code

```
self.pin = pin
```

macht nix anderes als dass der Parameter pin aus dem Konstruktor in der Sensorinstanz abgelegt wird, so dass der Sensor später, wenn er die Daten von der HW mit der Methode getData holt, kennt und benutzen kann. In der zweiten Zeile wird die Varable temperature definiert und mit None initialisiert um anzuzeigen, dass der jungfräuliche Sensor erst einmal keine Daten hat

Code

```
def getData(self):
        self.temperature = simulateData(self.getPin())
        return self.temperature
```

Diese Methode besorgt sich die Sensordaten vom Sensor von dem Pin, der beim Erstellen einer Klasseninstanz angegeben wurde und speichert sie in der Instanzvariable temperature. Dadurch kann der Wert häufiger abgefragt werden. Die Variable des Pins wird mit der get Methode geholt und im Aufruf an simulateData weitergegeben. Man kann auch *self.pin* benutzen - sollte aber besser auch klassenintern immer die get und set Methoden benutzen. Zum Schluss wird das Ergebnis noch gleich an den Aufrufer zurückgeliefert.

Code

```
def getPin(self):
        return self.pin

def getTemperature(self):
        return self.temperature
```

Diese beiden get Methoden liefern die benutzte Pinnummer und die Temperatur zurück. Man könnte beispielsweise auch die getTemperature() Methode die getData() Methode aufrufen lassen und würde dann immer den gerade aktuellen Sensorwert erhalten. In diesem Beispiel wird aber zwischen Sensordaten lesen und der Rückgabe des letzten gelesenen Wert unterschieden.

Code

```
mySensor=Sensor(1)
while True:
    result = mySensor.getData()
    if result:
        print "Got data from sensor with pin %d - Temperature %2.2f°C" %
(mySensor.getPin(), result)
    else:
        print "No data found for sensor with pin %d" % (mySensor.getPin())
    time.sleep(3)
```

In den nun folgenden Zeilen wird eine Instanz eines Sensors angelegt. Dabei wird der Klassenname Sensor benutzt und alle Parameter für den Konstruktor __init__ mitgegeben. In diesem Falle ist es die Pinnummer 1. Der erste Parameter von __init__, das self, wird automatisch von Python im Methodenaufruf eingesetzt.

Danach wird in einer Endlosschleife alle 3 Sekunden der aktuelle Wert aus dem Sensor ausgelesen und, sofern ein Wert verfügbar ist, dieser ausgegeben. Dabei sieht man die Standardsyntax, die benutzt wird, um OOP Methoden aufzurufen: es wird der Variablenname *mySensor* genommen und gefolgt von einem Punkt wird der Methodenname angehängt sowie die erforderlichen Parameter der Methode - sofern sie welche hat.

Code

```
mySensor.getData()
```

Somit existiert nun das erste kleine OOP Programm in Python wo die grundsätzlichen Konzepte sowie die Syntax benutzt wurde.

Übungsaufgabe

Wer das eben gelesene festigen will hat kann nun mal selbst eine kleine Klasse *Auto* erstellen an die folgende Anforderungen gestellt sind:

1) Das Auto hat eine Farbe und eine maximale Geschwindigkeit (beides Parameter für den Konstruktor und diese sind dann Instanzvariablen). *color* und *maxSpeed*
2) Das Auto hat get Methoden um die Farbe und die maximale Geschwindogkeit abzufragen (getColor() und getMaxSpeed())
3) Das Auto hat die Methoden ein(), aus() und setSpeed(speed). Damit wird das Auto eingeschaltet bzw ausgeschaltet und mit setSpeed wird die Geschwindigkeit gesetzt mit der das Auto fährt. Man könnte sich vorstellen, dass die Geschwindigkeit z.B. an eine PI HW weitergeleitet wird, die dann konkret die Geschwindigkeit eines angeschlossenen Motors steuert. Die zusaetzlichen Methoden bewirken natürlich, dass man noch zwei weitere private Variablen in der Klasse Auto haben muss:

Den Zustand (*started* = True/False) sowie die Geschwindigkeit (*speed* = Zahl > 0). Bei der Initialisierung des Autos müssen diese beiden Variablen auf False bzw 0 gesetzt werden. setSpeed muss natürlich prüfen, dass die Geschwindigkeit nicht negativ ist und nicht die maximale Geschwindigkeit überschreitet. Sollte das der Fall sein wird kein Fehler erzeugt sondern einfach keine Änderung des Autozustandes vorgenommen.
4) Das Hauptprogramm erzeugt zwei Autoinstanzen: Einen blauer VW mit maximal 150 kmh und einen roter Ferrari mit 350 kmh. Beide Autos werden gestartet und in einer Schleife beschleunigt der VW von 0 auf 100 in 10er Schritten, dann startet der Ferrari und beschleunigt auf 400 in 25er Schritten. Innerhalb der Schleifen wird die Geschwindigkeit der Autos mit print ausgegeben. Zum Schluss werden die beiden Autos ausgeschaltet.

Wer will kann ja sein kleines erstes OOP in diesem Thread vorstellen.