3 Objektorientierung

3.1 Was ist Objektorientierung ?

Objektorientierung ist ein Programmierparadigma, welches vor allem die Wiederverwendbarkeit und Wartung von Programmcode erleichtern soll. Gerade bei grösseren Programmierprojekten ist die Objektorientierung sehr hilfreich.
Bei der objektorientieren Programmierung werden **Klassen** erzeugt, welche die Eigenschaften von Objekten festlegt. Die Eigenschaften sind Werte (in Form von Variablen). Werte einer Klasse werden **Attribute** genannt. Eine Klasse kann auch Funktionen enthalten - diese werden **Methoden** genannt.

Beispiele:

- Was für Attribute und Methoden hat eine Türe ?

- Ein **Dreieck** besteht aus 3 Punkten. Ein **Punkt** besteht (in 2D) aus x- und y-Koordinate. Hier haben wir die beiden Objekte "Dreieck" und "Punkt" beschrieben.

3.2 Klassen definieren

Eine Klasse beschreibt die Struktur eines Objektes, welche aus **Attributen** und **Methoden** besteht.
Achtung: Eine Klasse ist die Beschreibung eines Objekttyps, aber noch kein Objekt. Das eigentliche Objekt ist eine sogenannte **Instanz** der Klasse. Aus einer Klasse können zahlreiche Instanzen angelegt werden.

In Python wird die Klasse mit dem Schlüsselwort "class" und dahinter der Klassenname definiert. Danach folgt ein Doppelpunkt und auf den nächsten Zeilen (eingerückt) werden Methoden und Attribute definiert. Dies geschieht innerhalb des Konstruktors (init),

Beispiel: Definition der Klasse "Punkt":

```
class Punkt:   # Klassendefinition (Beschreibung Klasse)
    def __init__(self):
        self.x = 0      # Attribut
        self.y = 0      # Attribut
```

Methoden (d.h. Funktionen innerhalb einer Klasse) werden mit dem def-Schlüsselwort angelegt. Alle Methoden haben **immer** mindestens einen Parameter, nämlich das Objekt selbst. In der Python-Community gibt es die Konvention diesen Parameter "**self**" zu nennen. Es wäre möglich diesen Parameter beliebig zu benennen, aber wir halten uns immer an diese Konvention!
Eine weitere Konvention ist, dass Methodennamen immer mit Kleinbuchstaben geschrieben werden. Wir halten uns wenn immer möglich auch an diese Konvention.
Und wenn wir schon bei Konventionen sind: Klassennamen werden mit der "CapWords" Konvention definiert, das heisst Wörter beginnen immer mit Grossbuchstaben, auch wenn diese zusammengesetzt sind, z.B. Vector, VectorLayer, GeometryStyle.

(mehr zu Coding Styles: siehe PEP 8: http://legacy.python.org/dev/peps/pep-0008/, dies ist der defacto Style Guide für die Python Programmierung)
Beispiel: Methoden für die Klasse "Punkt":

```
class Punkt:
   def __init__(self):
        self.x = 0
        self.y = 0

   def ausgabe(self):
      print("Der Punkt hat die folgenden Komponenten:")
      print("  x = ", self.x)
      print("  y = ", self.y)

   def distance(self, other):
      return ((self.x-other.x)**2 + (self.y-other.y)**2)**0.5
```

3.3 Instanz anlegen: Die Instanziierung

Nachdem die Klasse beschrieben wurde, können wir nun **Instanzen** der Klasse anlegen. Um eine Klasse zu instanziieren wird die Klasse mit dessen Namen und dahinter runde Klammern aufgerufen. Der Rückgabewert ist eine Instanz der Klasse und wird normalerweise in einer Variablen gespeichert.

```
P = Punkt()   # Instanz anlegen
P.x = 5       # Attribut x setzen
P.y = 10      # Attribut y setzen
P.ausgabe()   # Methode aufrufen

P2 = Punkt()  # zweite Instanz anlegen
P2.x = 8      # Attribut x setzen
P2.y = 4      # Attribut y setzen
P2.ausgabe()  # Methode aufrufen

d = P2.distance(P)  # Methode "distance" aufrufen.
print(d)
```

3.4 Der Konstruktor

Der Lebenszyklus jeder Instanz ist immer gleich:

1. Die Instanz wird erzeugt
2. Die Instanz wird benutzt
3. Die Instanz wird gelöscht

Die Klasse ist dafür verantwortlich, dass sich die Instanz immer in einem sinnvollen Zustand befindet. Es gibt eine spezielle Methode, welche **automatisch** beim

instanziieren eines Objektes aufgerufen wird, um das Objekt in einen sinnvollen, gültigen Zustand zu bringen. Diese Methode wird **Konstruktor** genannt.
Um einer Klasse einen Konstruktor zu geben wird eine Methode mit dem Namen `__init__` definiert. Die Methode heisst init und wird links und rechts mit je zwei Unterstrichen ergänzt. Solche haben in Python eine besondere Bedeutung. Wir werden diese "Magic Methods" später noch im Detail ansehen.

3.5 Der Destruktor

Analog zum Konstruktor gibt es einen Destruktor, welcher aufgerufen wird, wenn die Klasse gelöscht wird. Der Destruktor wird in Python eher selten benötigt. Der Destruktor ist eine Methode mit dem Namen `__del__`.

```
class TestKlasse:
   def __init__(self):
      print("Hallo vom Konstruktor")
   def __del__(self):
      print("Hallo vom Destruktor")
```

Vorsicht: Python garantiert nicht, dass alle sich noch im Speicher befindlichen Instanzen gelöscht werden. Es kann unter Umständen vorkommen, dass diese erst zu einem späteren Zeitpunkt freigegeben werden. Der Destruktor sollte deshalb nicht für komplexere Aufräumarbeiten wie Dateien schliessen oder Netzwerkverbindungen beenden verwendet werden.

```
A = TestKlasse()
del A
```

3.6 Objektorientierte Beziehungen

Eine Klasse kann von einer anderen Klasse abhängig sein.

Beispiel:
Das Objekt Punkt hat die Attribute x- und y-Koordinate.
Das Objekt Dreieck hat die **Attribute** A,B,C (Punkt).
Das Objekt Dreieck hat die **Methode** "Fläche", um die Fläche des Dreiecks zu berechnen.

3.7 Mehrere Konstruktoren

In Python gibt es nur einen Konstruktor, nämlich die Methode `__init__()`. In anderen Computersprachen kann es durchaus mehrere Konstruktoren mit Unterschiedlicher Anzahl Parametern oder unterschiedlicher Typen geben.

In Python kann das nur durch Standardparameter, Keyword-Parameter, Argumentliste oder mit einer Keyword-Argumentliste realisiert werden.

In der Regel sollten optionale Attribute immer mit Standard-Parameter-Werten im Konstruktor verwendet werden. Argumentlisten dürfen nur im absoluten Notfall verwendet werden, da es die Lesbarkeit des Python-Codes stark einschränkt.

3.8 Setter- und Getter-Methoden

Wir haben bereits eine sehr einfache Klasse "Punkt" kennengelernt welche die beiden Attribute x und y besitzt. Auf diese Attribute kann direkt zugegriffen werden. Dasselbe gilt für die Klasse Temperatur, bei der wir auf den Wert („value“) direkt zugreifen können.

```
class Temperature:
   def __init__(self):
      self.value = 0  # Temperatur in Celsius



T = Temperature()
T.value = 15      # Setzen der Temperatur (Celsius)
```

Viel besser ist es, wenn sogenannte Setter- und Getter-Methoden zu verwenden. Setter-Methoden sind dazu da einen Wert zu setzen ohne direkt auf das Attribut zuzugreifen. Das Attribut darf von aussen nur über diese Setter Methode verändert werden.
Dasselbe gilt für die Getter Methode. Der Wert eines Attributes kann von aussen nur über eine Methode ausgelesen werden.
Solche Attribute sind "private" Attribute und **nach Konvention** (Coding-Style) beginnen diese mit einem Unterstrich.

```
class Temperature:
   def __init__(self):
      self._value = 0

   def setValue(self, c):
      self._value = c

   def getValue(self):
      return self._value
```

Der Vorteil von Setter-/Getter-Methoden ist es, dass bei der Zuweisung überprüft werden kann, ob die Werte sinnvoll sind, oder ob gegebenenfalls etwas korrigiert werden muss. Es kann sogar eine Fehlermeldung (Exception) erstellt werden, wenn die Daten komplett falsch sind:

```
class Temperature:
   def __init__(self):
      self._value = 0

   def setValue(self, c):
      if c<-273.15:
          raise ValueError("Darf nicht kleiner als 273.15 Grad sein.")
      self._value = c

   def getValue(self):
      return self._value
```

Wenn direkt über das Attribut „_value“ zugegriffen würde, so könnte der Wertebereich nicht überprüft werden.

3.9 Property Attribute

Mit den Getter- und Setter-Methoden ist der Zugriff auf die Daten möglich. Mit der Setter-Methode können wir die Daten verändern und mit der Getter-Methode können wir Werte zuweisen.
Auf Ebene der Instanz ist das manchmal eher mühsam, da für jedes Attribut über die Methoden darauf zugegriffen wird.

Abhilfe schafft dabei die property definition, welche die Getter und Setter Methode automatisch aufruft.

```
class Temperature:
   def __init__(self):
      self._value = 0

   def setValue(self, c):
      if c<-273.15:
          raise ValueError("Darf nicht kleiner als 273.15 Grad sein.")
      self._value = c

   def getValue(self):
       return self._value

   value = property(getValue, setValue)
```

```
sensor1 = Temperature()
sensor1.value = 10    # indirekter Aufruf des Setters
print(sensor1.value)  # indirekter Aufruf des Getters
```

3.10 Mehrere Properties

Wie könnte die Temperature Klasse aussehen, wenn neben Celsius auch Fahrenheit und Kelvin unterstützt werden soll ?

Hinweis: Formeln zur Umrechnung Celsius/Kelvin/Fahrenheit:

C = K - 273.15
K = C + 273.15
F = C * 1.8 + 32
C = (F - 32) / 1.8

C: Wert in Celsius
K: Wert in Kelvin
F: Wert in Fahrenheit
Es gibt noch weitere klassische Temperaturskalen, siehe z.B: http://de.wikipedia.org/wiki/Newton-Skala