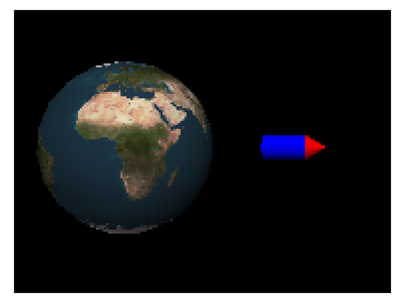
Einführung in VPython



- ☞ Ausführliche Dokumentation unter **http://www.vpython.org**
- ☞ Installation von VPython (**Achtung**: an der LMU ältere Version)
  - zuerst Python (z. B. Version 3.2) installieren
  - dann dazu passendes VPython (z. B. Version 5.74) installieren

Erzeugen einer einfachen Rakete und der Erde

Eine einfache Rakete besteht aus einem Zylinder, einem Kegel. Die Erde ist eine Kugel. Diese können in vpython mit den Klassen `cylinder()` (cylinder, engl. heißt Zylinder), `cone()` (cone, engl. heißt Kegel) und `sphere()` (sphere, engl. heißt Kugel) erzeugt werden.

Um das Visual Modul nutzen zu können, muss es zunächst eingebunden werden. Dies geschieht mit der Zeile

from visual import *

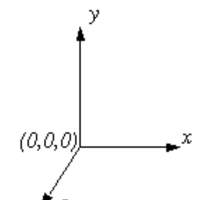
d. h. alle Klassen und Methoden des Moduls visual werden eingebunden.

Es wäre auch möglich, nur einzelne Methoden einzubinden (sinnvoll, wenn nur diese eine Methode benötigt wird). Z. B.

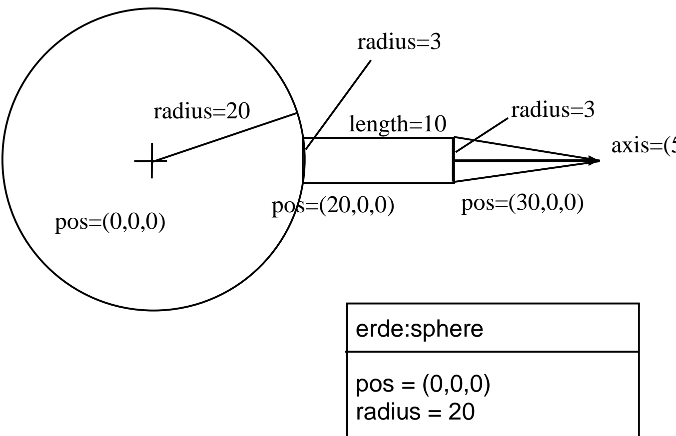
from random import choice oder z. B. **from random import random**

fügt nur die Methode "`choice`" bzw. im zweiten Beispiel die Methode "`random`" des Moduls "`random`" hinzu. Im zweiten Beispiel heißen die Methode und das Modul zufälligerweise gleich.

Im Visual-Modul gibt es 3 Achsen, wobei die z-Achse sozusagen aus dem Monitor raus zum Benutzer hin zeigt. Siehe Abbildung



Objektkarten der 2 "Raketenobjekte" und der Erde:



erde:sphere
pos = (0,0,0) radius = 20

rumpf:cylinder
pos = (20,0,0) radius = 3 length = 10

spitze:cone
pos = (30,0,0) radius = 3 axis = (5,0,0)

Das **Objekt** `erde` ist von der **Klasse** `sphere` und hat die angegebenen Attributwerte.
Um in Python ein solches Objekt zu erzeugen, werden folgende Zeilen benötigt:

```
erde = sphere()        #bedeutet: Erzeuge ein Objekt namens erde der Klasse sphere
sphere.pos = (0,0,0)
sphere.radius = 20
```

Die zweite Zeile könnte weggelassen werden, da pos = (0,0,0) die Standardbelegung ist. Default bei radius ist 1.

Das Objekt `rumpf` besitzt außerdem noch das Attribut `length`. Dieses gibt die Länge des Zylinders an. Nahezu jede Klasse in vpython hat außerdem auch ein Objekt `axis`, standardmäßíg ist `axis` (1,0,0). `axis` gibt die Richtung an, in die das Objekt zeigt, d. h. standardmäßig ist dies immer die x-Achse.

Alternativ können die Attributwerte auch beim Erzeugen des Objekts gesetzt werden:

```
erde = sphere(pos=(0,0,0), radius=20)
```

Die Reihenfolge der Attribute ist dabei egal. Weitere Attributwerte können analog gesetzt werden.

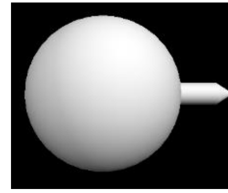
Anstatt den x-, y- und z-Wert in pos anzugeben, kann auf diese Werte auch einzeln zugegriffen werden:

```
erde.x = 0             # x-Position ist Null, bewirkt das Gleiche wie erde.pos.x = 0
erde.y = 0
```

usw.

Aufgabe (Erde und Rakete erzeugen):

Erzeuge eine Erde und eine Rakete, die aus oben angegebenen Objekten bestehen.



Darstellung:

Mit der rechten Maustaste kann die Szene gedreht werden.

Mit beiden Maustasten gleichzeitig kann gezoomt werden

Überblick VPython-Klassen

Neben den schon erwähnten Klassen gibt es noch einige weitere. Hier erhältst du einen Überblick über alle VPython-Klassen:

Arrow (Pfeil)		Curve (Linienzug)		Faces (Fase)	
Box (Quader)		Cylinder (Zylinder)		Helix	
Cone (Kegel)		Ellipsoid		Label (Text auf Objekt)	
Convex		Extrusion		Lights	
Points		Pyramid (Pyramide)		Ring	
Sphere (Kugel)		Text		frame	Gruppe

Veränderung der Fenster- /Kameraeigenschaften:

Für eine ansprechende Optik kann es sinnvoll sein, einige Änderungen am eigentlichen Fenster/Kamera vorzunehmen.

‚scene‘ ist der Standardname des ersten visual-Fensters, das automatisch erzeugt wird.

Es gibt zahlreiche Möglichkeiten die Fenstereigenschaften einzustellen bzw. die Kamera zu steuern (siehe dazu online-Dokumentation).

Hier einige Beispiele:

Fenstereigenschaften:

scene.title	Fenstertitel
scene.background	Farbe Hintergrund
scene.height	Fensterhoehe
scene.width	Fensterbreite
scene.fullscreen	Boolean-Wert, gibt Vollbildmodus an
scene.stereodepth	fuer 3D-Effekte sollte der Wert 2 sein
scene.stereo	für die Ansicht mit Anaglyphen-Brillen , z.B. „redcyan“ (oder redblue, yellowblue) einstellen

scene.ambient	Farbe des Umgebungslichts (default 0.2, erhöht man diesen Wert, wird die Szene heller, sonst dunkler)
scene.lights	Liste von "Lampen", die die Szene beleuchten, siehe dazu online-Dokumentation
scene.cursor.visible	Boolean-Wert, gibt an, ob Cursor sichtbar ist

Kamerasteuerung:

scene.center	Position, auf die die Kamera schaut, default (0,0,0)
scene.forward	Vektor, in dessen Richtung die Kamera schaut, default (0,0,-1)
scene.userzoom	Boolean-Wert, gibt an, ob der Benutzer zoomen darf
scene.userspin	Boolean-Wert, gibt an, ob der Benutzer rotieren darf
scene.autoscale	Boolean-Wert, sorgt für automatische Skalierung

Farben:

Die Farben können als rgb-Werte angegeben werden. Dabei erwartet vpython jeweils Werte zwischen 0 und 1.

Beispiel:

```
rumpf = sphere(pos=(20,0,0), radius = 3, color = (0,0,1))
```

erzeugt im Ursprung des KO-Systems einen blauen Zylinder namens rumpf mit Radius 3.

Manche Farben sind schon vordefiniert in der Klasse **color** enthalten, die wichtigsten:

color.red, color.yellow, color.black, color.green, color.orange, color.white, color.blue, color.cyan, color.magenta

Um leuchtende Farben zu erzeugen kann die Methode

```
rgb_to_hsv(c)
```

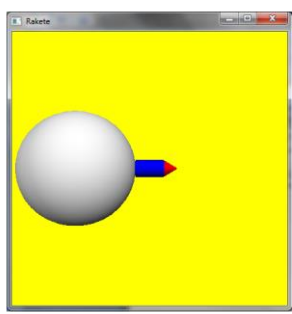
verwendet werden. c stellt dabei einen rgb-Wert, z. B. (1,1,0) dar.

Beispiel:

```
c = (1,1,0)
neue_farbe = color.rgb_to_hsv(c)
```

Aufgabe Titel und Hintergrundfarbe:

Nenne den Fenstertitel 'Rakete' und setze die Hintergrundfarbe auf gelb. Färbe zudem den Rumpf blau, und die Spitze rot. Setze das Bildzentrum auf den die Mitte des Rumpfes, also auf (25,0,0).



Material

Jeder Körper kann aus einem bestimmten Material bestehen. Attribut material, Attributwert z. B.

materials.chrome	das Material ist Chrom,
materials. marble	Marmor
materials.wood	Holz
materials.rough	
materials.plastic	
materials.earth	
materials.BlueMarble	
materials.emissive	sieht aus, als würde es leuchten

(und noch weitere - siehe Dokumentation)

Aufgabe Material:

Setze den Attributwert des Attributs `material` des Objekts `erde` auf `'materials.earth'`. Teste auch den Materialwert '`BlueMarble`'.

3D-Darstellung

Aufgabe:

Ändere scene.stereo auf 'redcyan'. Da das Weltall i. A. schwarz ist, stelle den Hintergrund zurück auf schwarz. Teste dein Programm, was ist passiert? Wann verwendet man 'redcyan', wann 'redblue' oder 'yellowblue'?

Gruppierung der Objekte

Bisher besteht die Rakete aus zwei voneinander unabhängigen Objekten. Um nur ein einziges Objekt Rakete zu erhalten, müssen diese gruppiert werden.

Dazu erstellt man zunächst eine Gruppe rakete. In Python benötigt man hierfür die **Klasse** `frame()`.

```
rakete = frame()                #eine Gruppe namens rakete wird erzeugt
```

Um der Gruppe rakete mitzuteilen, welche Objekte sie enthält, muss das Attribut `'frame'` des jeweiligen Objekts den Attributwert des Gruppennamens, hier ist dies `'rakete'`, erhalten.

Die Position der Gruppenobjekte sind dabei relativ zu den anderen Gruppenobjekten. Die gesamte Gruppe hat standardmäßig jedoch wieder die Position (0,0,0) und kann somit (als Gruppe) an einen anderen Ort platziert werden.

Beispiel:

```
rakete = frame()
rumpf = cylinder(frame = rakete, pos = ....)
spitze = sphere(frame = rakete, pos = ....)

rakete.pos=(...)
```

Aufgabe Gruppierung:

Setze die Position des vorher erzeugten Rumpfs auf (0,0,0) und die der Spitze auf (10,0,0). Gruppiere anschließend die Objekte der Rakete und ändere die Position der Gruppe Rakete so, dass die Lage wie zuvor rechts der Erde ist.

Raketenflug:

Um die Rakete „fliegen" zu lassen, benötigen wir einige algorithmische Grundkonzepte:

Die Wiederholung

☞ die **zählerbasierte Wiederholung**

hier ist die Anzahl der Wiederholungen bekannt. In Python wird dies mit Hilfe der `for`-Schleife und z. B. der Funktion `range()` realisiert.

Die Funktion range() kann einen, zwei oder auch drei Parameter haben.

Ein Parameter:

`range(n)`	liefert die ersten n Zahlen ab der 0	
z. B. `range(8)`	liefert also	0, 1, 2, 3, 4, 5, 6, 7

Zwei Parameter:

`range(a,b)` liefet alle ganzen Zahlen ab a bis ausschließlich b

z. B. `range(3,6)`	liefert also	3, 4, 5

Drei Parameter:

`range(a,b,i)` liefert alle ganzen Zahlen ab a bis ausschließlich b mit Schrittweite i

z. B. `range(4,10,2)`	liefert also	4, 6, 8

bzw.

```
for i in range(0,3):
    print (i)
```

z. B. `range(10,4,-2)`	liefert also	10, 8, 6

In Verbindung mit einer for-Schleife:

```
for i in range(0,3:
     print (i)
```

Ausgabe:

```
0
1
3
```

☞ die **bedingte Wiederholung**

eine bestimmte Sequenz wird solange ausgeführt, wie eine Bedingung erfüllt ist.
z. B.

```
i = 0
while i < 3:
     print (i)
     i = i+1
```

☞ **Weitere Anweisungen** in Wiederholungen

`break`	verlässt die Schleife sofort, im Anschluss wird der Programmcode nach der Schleife ausgeführt
`continue`	beendet diesen Schleifendurchlauf und fährt mit dem nächsten Durchlauf fort ohne evtl. noch verbleibende Anweisungen dieses Durchlaufs zu beachten.

☞ Die **Endlosschleife** und die simulierte Wiederholung mit **Endbedingung**

Mit dem Ausdruck while(1) wird eine Endlosschleife erzeugt. (Anstatt der 1 könnte auch jeder wahre Ausdruck, z. B. while(2+3 == 5) stehen). Findet sich nirgendwo in der Schleife ein break, so wird diese also nie beendet.
Eine Wiederholung mit Endbedinung kann man also folgendermaßen simulieren (der Konstrukt der Bedingung findet sich später in diesem Dokument):

```
while(1):
      Anweisungen
      if  <Bedingung>:
            break
```

Die Funktion rate(frequenz) von vpython

Diese Funktion muss immer innerhalb einer Schleife aufgerufen werden. Sie setzt die Animationsfrequenz dann genau auf die Zahl, die der Funktion als Parameter mitgegben wird.
z. B.
`rate(30)` sorgt, dafür, dass 30 Bilder pro Sekunde erzeugt werden.

Die Auswahl

Beispiel:

```
buchstabe = input("Drücke eine Buchstabentaste ")

if buchstabe == "a":
   print("Du hast ein a eingegeben")
elif buchstabe == "b":
   print("Du hast ein b eingegeben")
elif buchstabe == "c":
   print("Du hast ein c eingegeben")
else:
   print("Du hast weder ein a, noch ein b, noch ein c eingegeben")
```

Funktionen

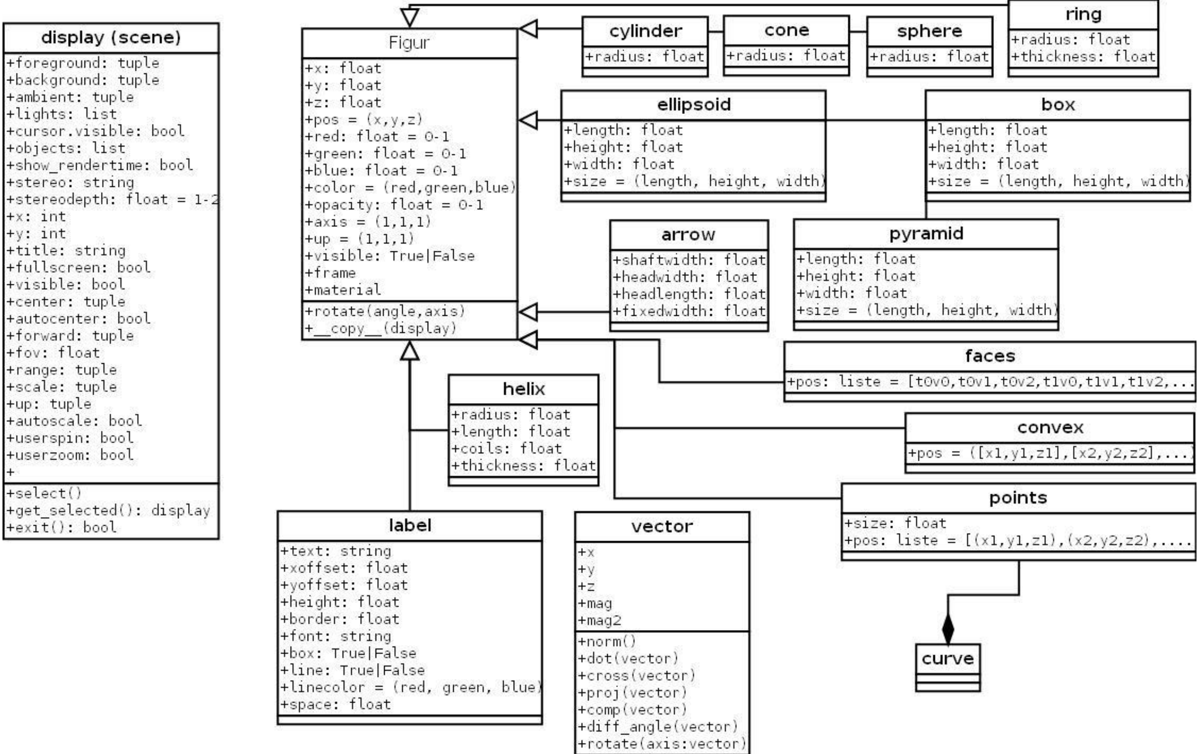
Funktionen (und Methoden von Klassen) werden mit dem Schlüsselwort `def` erstellt.
Beispiel:

```
def bewegen(objekt,treibstoff,schritt):

  if treibstoff>0:
    objekt.pos= objekt.pos+ schritt
    treibstoff=treibstoff-1
```

Klassen

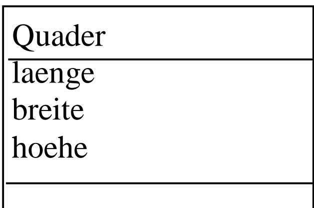
Natürlich lassen sich auch in Python Klassen erstellen. Schauen wir uns dazu zunächst einmal die gegebenen Klassen in VPython an (genauere Erklärung der Attribute siehe in der online-Hilfe unter http://vpython.org/contents/docs/index.html bzw. der in der idle eingebetteten Hilfe).



Beispielaufgabe Quader:

Erstelle eine Klassenkarte der Klasse Quader (allgemein, ohne Berücksichtigung von VPython). Methoden sollen dabei noch nicht berücksichtigt werden

mögliche Lösung:



Umsetzung in Python:

Um eine solche Klasse in Python zu erzeugen benötigt man z. B. folgende Zeilen:

```python
class Quader():
    def __init__(self):
        self.laenge = 1
        self.breite = 1
        self.hoehe = 1
```

Betrachten wir die einzelnen Zeilen des Beispiels näher:

Zeile `class Quader():`

Eine neue **Klasse** namens `Quader` wird erstellt.
Die Zeile wird durch einen Doppelpunkt abgeschlossen. Das bedeutet, dass anschließend die eigentlichen "Inhalte" der Klasse folgen. Diese müssen - wie alle Code-Blöcke - eingerückt werden.

Zeile `def __init__(self):`

Hierbei wird der **Konstruktor** erzeugt. (engl. to construct = errichten).
Das Schlüsselwort `def` bewirkt, dass eine neue Methode definiert wird.
Unsere neue Methode heißt in diesem Fall `__init__`.
In Python wird ein Konstruktor einer Klasse immer `__init__` genannt. Diese Methode wird als allererstes beim Erstellen eines neuen Objekts aufgerufen. Somit können dann die Attribute mit ihren Werten gesetzt werden.
Der Konstruktor benötigt immer den Parameter `'self'` (engl. self = selbst). Damit ist der Selbstbezug, also das Objekt der Klasse selbst gemeint. Dies ist notwendig, um Attribute mit Hilfe der Punktnotation zu setzen. Bei der Punktnotation wird zunächst der Objektname, dann das Attribut genannt (getrennt durch einen Punkt). Da bei der Klassenerstellung der Name eines zugehörigen Objekts nicht bekannt ist, nennt man dieses `'self'`.
(Als nächstes können theoretisch Werte genannt werden, die schon im Konstruktor gesetzt werden sollen - kompakte Schreibweise, also z. B.
`__init__(self, groesse=10, laenge=20)`
Dann könnte ein Objekt dieser Klasse bereits mit diesen Attributen erzeugt werden, die Punktnotation wäre zum Setzen der Attributwerte nicht notwendig. Es könnten bei der kompakten Schreibweise natürlich auch noch weitere Parameter angegeben werden, die mit vordefinierten Werten oder auch ohne Wert belegt sind. Wichtig ist dabei nur, dass **zunächst die noch nicht belegten, danach die vordefinerten Parameter erscheinen**. Diese Parameter sind jedoch nicht automatisch auch Attribute der Klasse.)
Attribute erzeugt man anschließend in der Konstruktormethode.
Der Doppelpunkt am Ende der Zeile wieder an, dass nun die durch den Konstruktor auszuführenden Zeilen folgen (eingerückt).

Zeilen
```
self.laenge = 1
self.breite = 1
self.hoehe = 1
```

Die Attribute laenge, breite und farbe der neuen Klasse Quader werden erzeugt und mit einem beliebigen Wert (hier gleich 1) vorbelegt.

Nun benötigt man natürlich noch Methoden. Exemplarisch soll die Klasse die Setter-Methoden setzeLaenge(), setzeBreite() und setzeHoehe() und eine Methode volumen() erhalten. (Hinweis: Methoden, die zum Setzen, also der Wertzuweisung, von Attributen dienen, nennt man Setter-Methoden).

Quader
laenge breite hoehe
setzeLaenge() setzeBreite() setzeHoehe() volumen()

Der Pythoncode könnte nun folgendermaßen aussehen:

```
class Quader():
    def __init__(self):
        self.laenge=1
        self.breite=1
        self.hoehe=1

    def setzeLaenge(self):
        lang=int(input("Wie lang ist der Quader? "))
        self.laenge=lang

    def setzeBreite(self):
        breit=int(input("Wie breit ist der Quader? "))
        self.breite=breit

    def setzeHoehe(self):
        hoch=int(input("Wie hoch ist der Quader? "))
        self.hoehe=hoch

    def volumen(self):
        vol = self.laenge * self.breite * self.hoehe
        return vol
```

Die hinzugekommenen Methoden benötigen als Parameter natürlich auch den Selbstbezug `self`. Die Variablen `lang`, `breit`, `hoch` und `vol` sind jedoch keine Attribute dieser Klasse, sondern existieren nur in ihren jeweiligen Methoden.

Bisher wurde nur die Klasse Quader erzeugt, jedoch noch kein Objekt davon. Dieses muss in altbekannter Schreibweise noch erzeugt werden.

Beispiel für den fertigen Programmcode:

```
class Quader():
    def __init__(self):
        self.laenge=1
        self.breite=1
        self.hoehe=1

    def setzeLaenge(self):
        lang=int(input("Wie lang ist der Quader? "))
        self.laenge=lang

    def setzeBreite(self):
```

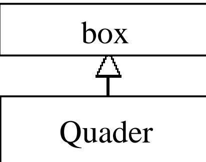
```
        breit=int(input("Wie breit ist der Quader? "))
        self.breite=breit

    def setzeHoehe(self):
        hoch=int(input("Wie hoch ist der Quader? "))
        self.hoehe=hoch

    def volumen(self):
        vol = self.laenge * self.breite * self.hoehe
        return vol


if __name__ == '__main__':
    q=Quader()
    q.setzeLaenge()
    q.setzeBreite()
    q.setzeHoehe()
    print("Das Volumen des Quaders ist",q.volumen())
```

Vererbung



Quader
laenge breite hoehe
setzeLaenge() setzeBreite() setzeHoehe() volumen()

```
from visual import *

class Quader(box):
    def __init__(self):
        box.__init__(self)
        self.laenge=self.length
        self.breite=self.width
        self.hoehe=self.height

    def setzeLaenge(self):
        lang=int(input("Wie lang ist der Quader? "))
```

```
        self.laenge=lang
        self.length=lang

    def setzeBreite(self):
        breit=int(input("Wie breit ist der Quader? "))
        self.breite=breit
        self.width=breit

    def setzeHoehe(self):
        hoch=int(input("Wie hoch ist der Quader? "))
        self.hoehe=hoch
        self.height=hoch

    def volumen(self):
        vol = self.laenge * self.breite * self.hoehe
        return vol

if __name__ == '__main__':
    q=Quader()
    q.setzeLaenge()
    q.setzeBreite()
    q.setzeHoehe()

    print("Das Volumen des Quaders ist",q.volumen())
```

Im Konstruktor ist die Zeile **box.__init__(self)** hinzugekommen. Die Klasse Quader erbt von der Klasse `box`. Alle Oberklassen müssen zunächst auch erstellt werden, sie benötigen also auch einen Konstruktor. Als Parameter muss dabei nur der Selbstbezug, also `self` angegeben werden, es wären jedoch auch hier weitere Parameter denkbar. Da dies keine neue Methode der Klasse `quader` ist, sondern der Methodenaufruf des Konstruktors der Klasse `box`, folgt am Ende dieser Zeile natürlich kein Doppelpunkt.

Die weiteren Methoden `setzeLaenge`, `setzeBreite` und `setzeHoehe` dienen zum Setzen der Attributwerte (hier im Englischen und im Deutschen).

Führt man dieses Programm aus, kann man zuschauen, wie sich der 3-dimensionale Quader sukzessiv seinen Maßen anpasst.

Vererbungen der Klasse frame in VPython

Beispiel:

```
class rakete(frame):
    def __init__(self):
        frame.__init__(self,position)
        self.rumpf=cylinder(frame=self,pos=(0,0,0), radius=3, length=10)
```

```
        self.spitze=cone(frame=self,pos=(10,0,0),axis=(5,0,0), radius=3)
        self.pos = position

    def bewegen (self,schritt):
        self.pos = self.pos+schritt

myrocket = rakete(position=(2,7,9))
```

Hier wird die Klasse `rakete` erzeugt. Dem Konstruktor (__init__(…)) wird dabei ein zusätzlicher Parameter „position" mitgegeben. Dies bewirkt, dass beim Erzeugen eines Objekts, dieser Wert gleich belegt werden kann. Solche Parameter sind damit nicht automatisch Attribute der Klasse. Möchte man dies, so müssen sie erst noch mit Hilfe von z. B. self.position = position erstellt werden.
Das im Beispiel verwendete Attribut `pos` erbt die Klasse `rakete` von der Oberklasse `frame` und muss damit nicht neu erstellt werden.
Die letzte Zeile bewirkt alse, dass ein neues Objekt namens myrocket an der Position (2,7,9) erstellt wird.