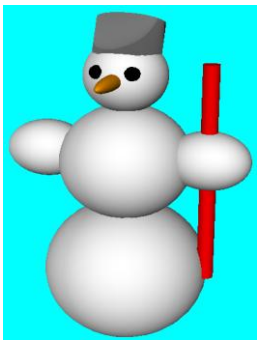


VPython - Grundlagen

- ☞ Ausführliche Dokumentation unter **http://www.vpython.org**
- ☞ Installation von VPython (siehe separate Anleitung)
  - zuerst Python (z. B. Version 3.2) installieren
  - dann dazu passendes VPython (z. B. Version 5.74) installieren

Erzeugen eines einfachen Schneemanns

Ein einfacher Schneemann besteht aus 3 Kugeln, diese können in vpython mit der Klasse **sphere()** (sphere, engl., bedeutet Kugel) erzeugt werden.

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

```
from visual import *
```

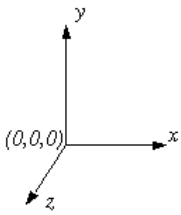
d. h. alle Klassen und Methoden des Moduls visual werden eingebunden.

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

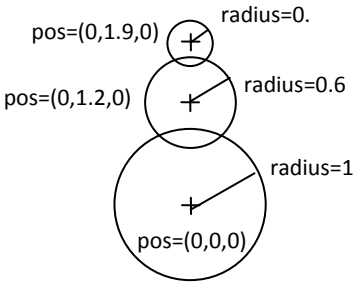
from random import choice *oder z. B.* *from random import random*

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 3 Schneemannkugeln:



bauch:sphere
pos = (0,0,0) radius = 1

brust:sphere
pos = (0,1.2,0) radius = 0.6

kopf:sphere
pos = (0,1.9,0) radius = 0.3

Das Objekt bauch ist von der Klasse sphere und hat die angegebenen Attributwerte.
Um in Python ein solches Objekt zu erzeugen, wird folgende Zeile benötigt:

```
bauch = sphere(pos=(0,0,0), radius=1)
```

Die Reihenfolge der Attribute ist dabei in diesem Fall egal. Weitere Attributwerte werden analog gesetzt. In diesem Fall könnten die Attribute sogar weggelassen werden, das pos mit (0,0,0) und radius mit 1 standardmäßig vorbelegt sind.

Die einzelnen Atributwerte könnten nach erstmaligem Erstellen des Objekts auch mittels Punktnotation gesetzt werden. Man würde also schreiben:

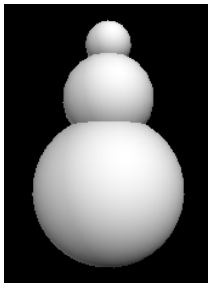
```
bauch = sphere()
bauch.pos = (0,0,0)
bauch.radius = 1
```

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

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

Aufgabe01 (Schneemann erzeugen):

Erzeuge einen Schneemann, der aus oben angegebenen Objekten besteht.



Darstellung:

Mit der rechten Maustaste kann die Szene gedreht werden.
Mit beiden Maustasten gleichzeitig kann gezoomt werden

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:

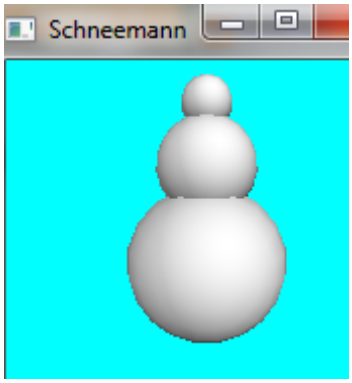
```
bauch = sphere(pos=(0,0,0), radius = 1, color = (1,1,1))
```

erzeugt im Ursprung des KO-Systems eine weiße Kugel namens Bauch mit Radius 1..

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)
```

Aufgabe02 (Titel und Hintergrundfarbe):

Nenne den Fenstertitel Schneemann und setze die Hintergrundfarbe auf cyan.

Gruppierung der Objekte

Bisher besteht der Schneemann aus drei voneinander unabhängigen Objekten. Um nur ein einziges Objekt Schneemann zu erhalten, müssen diese gruppiert werden.

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

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

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

Beispiel:

```
schneemann = frame()
bauch = sphere(frame = schneemann, pos = ....)
brust = sphere(frame = schneemann, pos = ....)
....
```

Aufgabe03 (Gruppierung und Verschieben):

a) Gruppiere die vorher erstellten Objekte des Scheemanns.
Lass anschließend deinen Schneemann 10 Schritte in x-Richtung wandern. Ein Schritt soll dabei 0.3 Einheiten lang sein.

Hinweise:
Um die Bewegung sehen zu können, setzen wir die Animationsfrequenz (also Anzahl der Animationen pro Sekunde) innerhalb der Schleife auf 30. Verwende hierfür die Methode

```
rate(30)
```

Um den Schneemann wandern zu lassen, verändere den x-Wert der Schneemannsposition sukzessive. Verwende also folgende Zeile (anstatt ... muss natürlich ein Wert stehen):

```
schneemann.pos = (...,0,0)
```

b) Aufgrund der automatischen Skalierung läuft der Schneemann nicht exakt in waagrechter Richtung. Verwende daher

```
scene.autoscale = False
```

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)

Aufgabe04 (Material):

Erzeuge eine Kugel und gib ihr den Materialwert earth bzw. auch BlueMarble

eigene Klassen erzeugen

Bevor auf das Erzeugen neuer Klassen eingegangen wird, soll der Begriff der Vererbung näher erläutert werden:

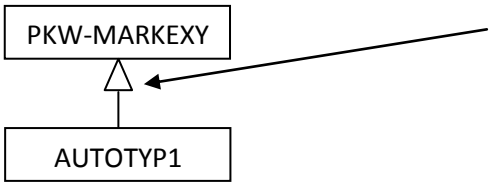
Definition **Vererbung**:

Klassen können von anderen Klassen *abgeleitet* werden (Vererbung).
Dabei erbt die Klasse (**Unterklasse**) die Attribute und Methoden von der *vererbenden* Klasse (**Oberklasse**). D. h. sie besitzt die gleichen Attribute und Methoden. Auf diese kann damit zugegriffen werden, ohne sie neu definieren zu müssen.
Die Unterklasse kann jedoch auch noch weitere zusätzliche Attribute bzw. Methoden besitzen. Methoden können auch überschrieben werden. Dies nennt man dann **Polymorphismus**. Dazu verwendet man den gleichen Methodennamen wie in der Oberklasse. Der zugehörige Codeblock, der nach Aufruf der Methode ausgeführt wird, unterscheidet sich jedoch.

Beispiel aus dem Alltag:

Oberklasse sei die Klasse PKW-MARKEXY, eine mögliche Unterklasse AUTOTYP1.

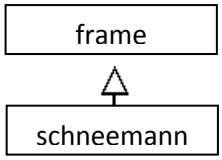
Grafische Darstellung:

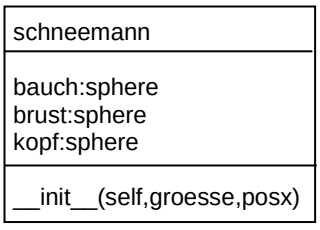


Vererbungspfeil
Dieser zeigt immer zur Oberklasse

Jeder PKW hat eine bestimmte PS-Zahl, einen Hubraum, ein Gewicht usw. Diese Attribute können somit in der Oberklasse definiert werden uns müssen in der Unterklasse AUTOTYP1 nicht noch einmal neu erstellt werden. Wir können jedoch trotzdem auch in der Unterklasse auf sie zugreifen. Zusätzlich kann in der Unterklasse beispielsweise das Attribut Farbe definiert werden. Auf die Farbe kann nun nur ein Objekt der Klasse AUTOTYP1 zugreifen, ein Objekt der Klasse PKW-MARKEXY kennt die Farbe nicht.
Bei jedem PKW lässt sich ein Rückwärtsgang einlegen. Standardmäßig geschieht dies durch Verschieben des Schalthebels nach links-oben. Bei manchen PKWs liegt der Rückwärtsgang jedoch rechts unten. Es ist nun möglich, in der Oberklasse eine Methode rückwärtsgang_einlegen() zu definieren und in dieser den Schalthebel nach links oben stellen zu lassen. Damit wird auch beim Aufruf dieser Methode in einer zugehörigen Unterklasse der Schalthebel nach links oben gestellt. Wird nun in der Unterklasse eine neue Methode gleichen Namens (rückwärtsgang_einlegen()) definiert und in dieser der Schalthebel nach rechts unten gestellt, so wird für jedes Objekt dieser Unterklasse beim Aufruf dieser Methode der Rückwartsgang durch Stellung des Schalthebels nach rechts-unten eingelegt. Die Methdode rückwärtsgang_einlegen() wurde überschrieben.

Zurück zu unserem Schneemannbeispiel:





Die erste Grafik gibt an, dass die Klasse 'schneemann' von der Klasse 'frame' erbt.

Die zweite Grafik zeigt eine Klassenkarte der Klasse 'schneemann'

```
class schneemann(frame):
        def __init__(self, groesse = 1, posx = 0):
                frame.__init__(self)
                self.bauch= sphere(frame = self, pos = (posx,0,0),radius = groesse)
                self.brust = sphere(frame = self,pos = (posx,groesse*1.2,0), radius = groesse*0.6)
                self.kopf = sphere(frame = self,pos = (posx, groesse*1.9,0), radius = groesse*0.3)

berta = schneemann(groesse = 1, posx = 0)
xaver = schneemann(groesse =0 .75, posx = -3)
```

Möchte man viele gleiche Schneemänner haben, so ist es sinnvoll, eine **eigene Klasse** Schneemann zu haben. Alle Werte, die man gleich beim Erzeugen setzen möchte, schreibt man in die Init-Methode (ähnlich wie bei der Kugel, als man die Attributwerte sofort beim Erzeugen des Objekts gesetzt hat - z. B. durch sphere(pos=(0,0,0)). (Die __init__-Methode nennt man Konstruktor, siehe unten). Anschließend definiert man die zugehörigen Objekte.

Betrachten wir die einzelnen Zeilen des Beispiels näher:

Zeile `class schneemann(frame):`

Eine neue Klasse namens schneemann wird erstellt.
Da die enthaltenen Objekte eine Gruppe bilden sollen, muss die neue Klasse über die Attribute und Methoden der Gruppenbildung verfügen.
Dies geschieht, indem sie diese von der Klasse frame (nicht zu verwechseln mit dem weiter oben genannten Attribut frame) erbt.
Eine Vererbung wird in Python implementiert, indem man die Oberklasse einfach in Klammern zu dem neuen Klassennamen schreibt.
Es ist auch möglich, eine Klasse von mehreren Oberklassen gleichzeitig erben zu lassen, dies nennt man dann **Mehrfachvererbung**.
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, groesse = 1, posx = 0):`

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 Werte genannt werden, die schon im Konstruktor gesetzt werden sollen. Hier z. B. groesse, die den vorbelegten Wert 1 erhält und posx, vorbelegt mit Null. Es könnten 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. Möchte man Attribute mit diesen Werten haben, muss man sie innerhalb des Konstruktors definieren.

Z. B. def __init__(self, groesse =1, posx =0):
...
self.groesse = groesse
self.positionx = posx
...

Hier werden zwei Attribute namens 'groesse' bzw. 'positionx' erstellt. Das erste Attribut, groesse, hat den gleichen Namen wie der Parameter groesse des Konstruktoraufrufs, Beim zweiten Attribut, positionx, wurde ein anderer Namen gewählt. Auf diese Attribute können dann alle weiteren Methoden der Klasse zugreifen. Dies wäre nicht der Fall, würde man die Parameterwerte nicht Attributen zuweisen. Ebenso können später deklarierte Objekte dieser Klasse darauf zugreifen.
Der Doppelpunkt am Ende der Zeile gibt wieder an, dass nun die durch den Konstruktor auszuführenden Zeilen folgen (wieder eingerückt).

Zeile frame.__init__(self)

Die Klasse schneemann erbt von der Klasse frame. 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 schneemann ist, sondern der Methodenaufruf des Konstruktors der Klasse frame, folgt am Ende dieser Zeile natürlich kein Doppelpunkt.

Zeilen self.bauch= sphere(frame = self, pos = (posx,0,0),radius = groesse)
self.brust = sphere(frame = self,pos = (posx,groesse*1.2,0), radius = groesse*0.6)
self.kopf = sphere(frame = self,pos = (posx, groesse*1.9,0), radius = groesse*0.3)

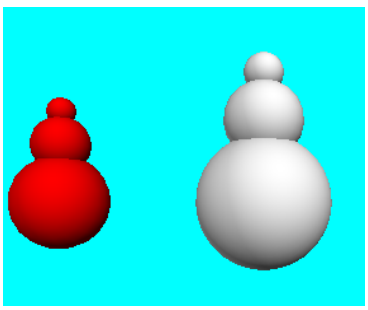
Die Attribute bauch, brust und kopf der neuen Klasse Schneemann werden mit Werten versehen. Alle Attribute sind in diesem Fall Kugeln. Deshalb beginnen die Zeilen jeweils mit self....=sphere(...).
Zunächst wird das Attribut frame der Klasse sphere gesetzt. Alle Kugeln sollen zu einer Gruppe des neuen Objekts der Klasse schneemann gehören. Dies wird durch 'frame=self' erreicht.
Anschließend werden die weiteren Attributwerte analog oben (als der Schneemann noch keine Klasse war) gesetzt. Der einzige Unterschied dabei ist, dass die Position bzw. der Radius von posx bzw. groesse abhängig sind. Dadurch können unterschiedlich große Schneemänner an verschiedenen Positionen der x-Achse erzeugt werden.

Zeilen berta = schneemann(groesse = 1, posx = 0)
xaver = schneemann(groesse =0 .75, posx = -3)

Diese beiden Zeilen sind nicht mehr eingerückt. Das heißt, sie gehören nicht mehr zur Klasse schneemann.
Durch diese Zeilen werden zwei Objekte der Klasse schneemann unterschiedlicher Größe und Position erzeugt.

Aufgabe06 (Klasse Schneemann):

Erzeuge analog obigem Beispiel eine Klasse Schneemann. Füge dem Konstruktor noch zusätzlich eine farbe, vorbelegt mit color.white hinzu. Erzeuge mit dieser neuen Klasse anschließend zwei Schneemänner, berta und xaver. xaver soll jedoch nicht weiß, sondern rot sein. Die übrigen Werte werden obigem Beispiel entnommen.



Einer Klasse weitere Methoden zufügen

Neben der Methode des Konstruktors kann eine Klasse natürlich noch weitere Methoden erhalten. Diese werden mit Hilfe des Schlüsselworts **def** erstellt und haben mindestens 'self' als Parameter.

Beispiel:

```
    def essen(self):
        if self.dick == False:                  # falls der Schneemann noch nicht dick ist
        i=0                                     # Laufvariable i definieren und auf 0 setzen
        while i < 10:                           # insgesamt wird die Schleife 10 mal
                                                #durchlaufen
            rate(3)                             # nur 3 Animationen pro Sekunde

             #der Radius von Bauch, Brust und Kopf wird nun schrittweite vergroessert
            #anstatt z. B. zahl = zahl + 1 zu schreiben, kann man auch kurz zahl += 1
            #schreiben
            self.bauch.radius += 0.2/10         # Bauch wird insgesamt 0.2 Einheiten groesser
            self.brust.radius += 0.1/10         # Brust wird insgesamt 0.1 Einheiten groesser
            self.kopf.radius += 0.05/10          # Kopf wird insgesamt 0.05 Einheiten groesser
             i += 1                              # Laufvariable i um 1 erhoehen
        self.dick = True                        # die while-Schleife ist fertig durchlaufen, jetzt
                                                #ist der schneemann dick
```

Eine Klasse von einer selbst erstellten Klasse erben lassen

Eine weitere Klasse namens schneekind wurde erstellt. 'schneekind' ist eine Unterklasse von schneemann, d. h. sie besitzt alle Attribute und Methoden von schneemann.
Die Klassendefinition und der Konstruktor lauten folgendermaßen:

```
class schneekind(schneemann):
    def __init__(self, posx = 0, posy = 0, groesse = 1, farbe = color.white):
        schneemann.__init__(self, groesse,posx)
        self.kindgroesse=groesse
```

Zum Erzeugen eines Schneekinds müssen keine Anweisungen für das Erzeugen der Kugeln erstellt werden. Diese Anweisungen wurden schon in der Klasse schneemann implementiert.
Der Konstruktor der neuen Klasse enthält in diesem Beispiel die gleichen Parameter wie die Oberklasse. Theoretisch wären jedoch auch noch weitere möglich.
Die erste Anweisung im Konstruktor des schneekinds ist der Konstruktor des schneemanns.
Die Klasse schneekind enthält das zusätzliche Attribut kindgroesse.
(kindgroesse deshalb, um zu erkennen, ob das Schneekind schon ausgewachsen ist)
Die Methode 'essen' ist schon in der Oberklasse implementiert und existiert somit auch für die Unterklasse. Weitere Methoden können jederzeit hinzugefügt werden bzw. kann natürlich auch die Methode 'essen' überschrieben werden.