



ANWENDUNGSORIENTIERTER PYTHONKURS FÜR INGENIEURE

Einführung (2)

Sebastian Voigt, Carsten Knoll

Dresden, 21.10.2013

Inhalt heute:

- Schreiben und Lesen von Dateien
- Funktionen in Python
- Komplexeres Einführungsbeispiel:
Mehrkörpersimulation mit PyMbs

Schlüsselwort `open`

```
fid = open('datei.txt', 'w')  
  
fid.write('hallo Datei')  
fid.writelines(['hallo', 'Datei'])  
fid.write(str(5))  
  
fid.close()
```

Schreiben

```
fid.open('datei.txt', 'r')  
a = fid.readline()  
  
print a
```

Lesen

Widerkehrende Teilaufgaben kapseln

→ Komplexität beherrschen

Häufige Fehler:

→ Doppelpunkt vergessen

→ Ergebnis `return` Vergessen (`None` wird zurückgegeben)

```
def tutnix():  
    pass                # dummy-statement  
  
print tutnix()         # None  
  
def quadrat1(z):  
    """Quadrieren einer Zahl"""  
    return z**2  
  
quadrat1(7)            # 49  
quadrat1(7-12)        # 25
```

```
def quadrat2(z):  
    x = z**2 # lokale Variable  
    print x  
    return x  
  
x, a = [5, 3] # "Entpacken" einer Liste  
quadrat2(a) # 9  
quadrat2(x) # 25  
print x # 5  
  
def quadrat3(z):  
    print x # jetzt globales x (nur lesend)  
    return z**2  
  
def quadrat4(z):  
    print x # Fehler (lokale Variable unbekannt)  
    x = z**2 # lokale Variable (Schreibzugriff)  
    return x
```

```
def quadrat5(z = 8):  
    return z**2
```

```
quadrat5()           # 64  
quadrat2(2.5)        # 6.25
```

```
def quadrat_summe(a, b = 0):  
    return a**2 + b
```

```
quadrat_summe(10)     # 100  
quadrat_summe(10, 3)  # 103
```

```
# Explizite Benennung der Argumente ("keyword args")  
quadrat_summe(b= -3, a = 10) # 97 (hier: Reihenfolge egal)  
# Hilfreich bei Funktionen mit vielen Argumenten
```

```
# non-keyword args
def sum(*args):          # diese Funktion gibt es schon!
    # args ist ein Tupel
    s = 0:
    for x in args:
        s+=x
    return s
sum(5, 20, 3)            # 28
sum(*range(4))           # 6      # Entpacken der Liste

# keyword args
def satz(**kwargs):
    print type(kwargs) # dict
    for key, value in kwargs.items():
        print "Das %s ist %s." % (key, value)

satz(Wasser = 'kalt')    # Das Wasser ist kalt.
satz(u = 8.2)            # Das u ist 8.2.
```

Funktionsaufrufe sind sehr flexibel

Fehlerquelle falsche Argumente

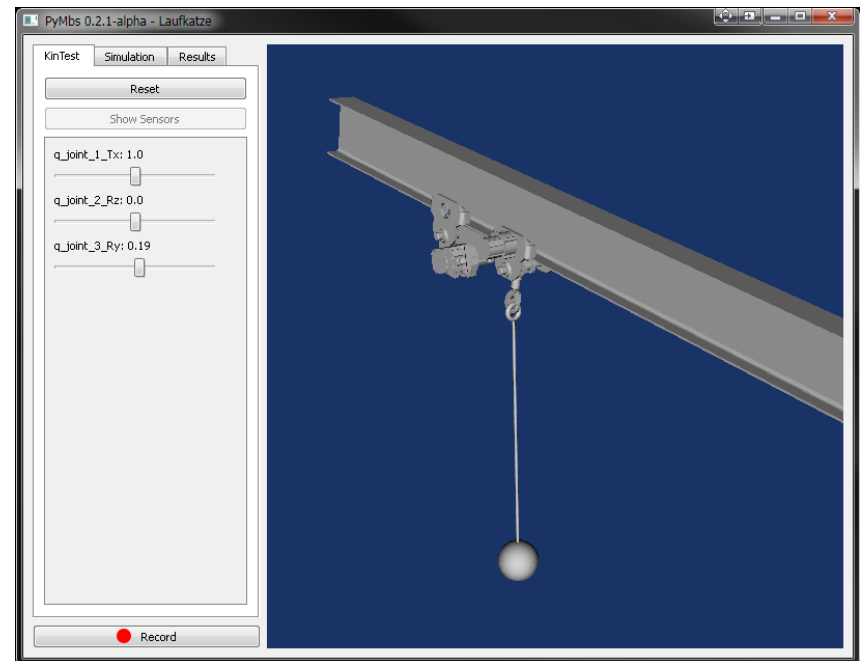
Typ-Prüfung oft sinnvoll (Aufwand-Nutzen-Abwägung)

```
def eine_funktion(satz, ganzzahl, zahl2, liste):  
    if not type(satz) == str:  
        print "Datentpyfehler: satz"  
        return -1  
    if not isinstance(ganzzahl, int):  
        print "Datentpyfehler: ganzzahl"  
        return -2  
    if not isinstance(liste, (tuple, list)):  
        print "Datentpyfehler: liste"  
        return -3  
  
# Kompakteste Variante (empfohlen):  
assert zahl2 > 0 # Assertation-Error bei Nichterfuellung
```

- Funktionen sind wichtig und mächtig
- Docstrings bei nichttrivialen Funktionen empfehlenswert
- Verschiedene Varianten für Funktionsaufruf
- Sicherheitsabfragen ersparen ggf. lange Fehlersuche
- Code lesen, ausprobieren → Potential nutzen

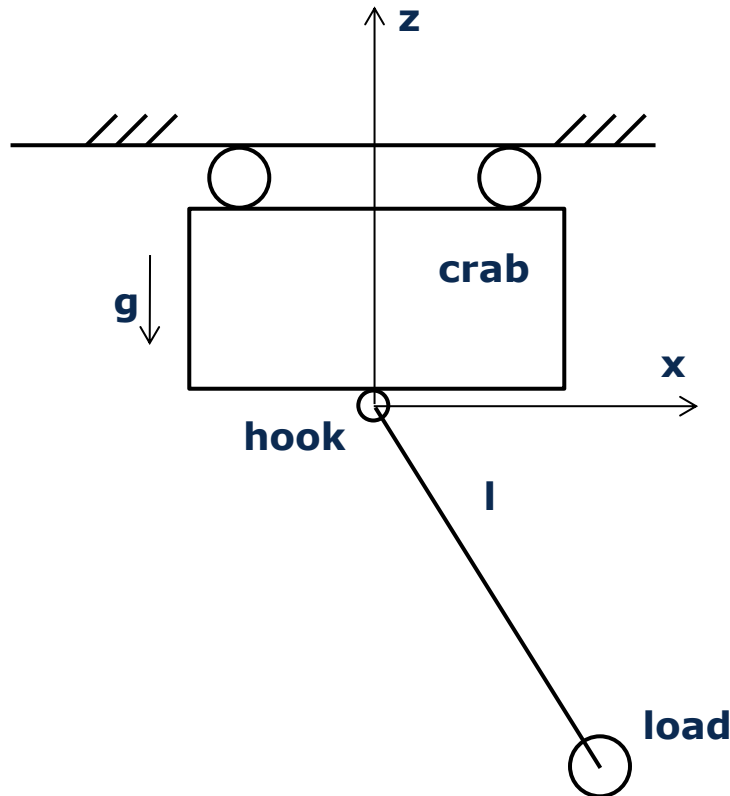
Mehrkörpersimulation mit **Python Multibody Systems** (PyMbs)

- Es werden Körper, Gelenke, Kraftelemente und Sensoren definiert
- PyMbs visualisiert das beschriebene System
- Kontrolle der definierten kinematischen Struktur
- Es wird Berechnungscode erzeugt und simuliert



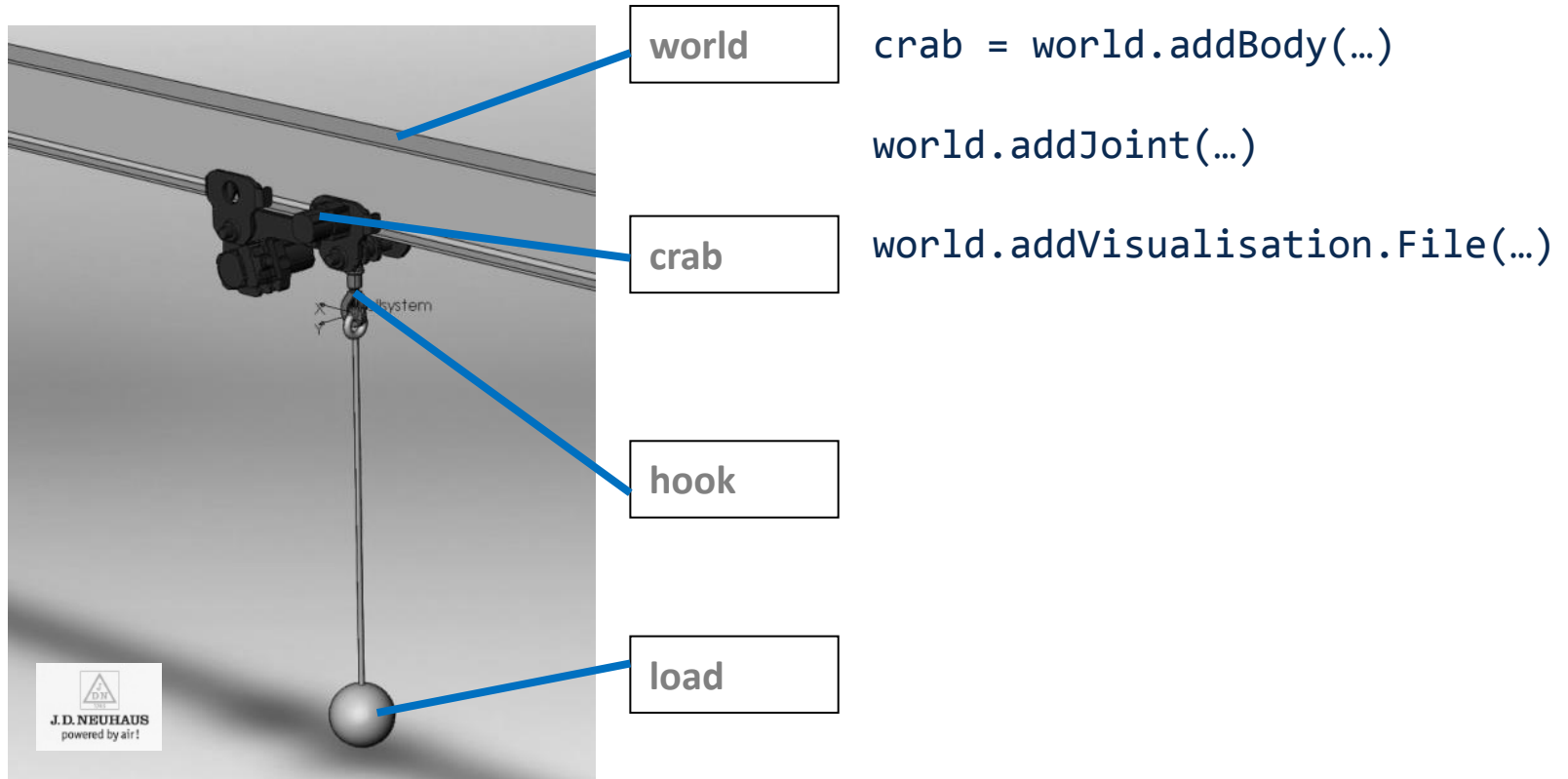
Modelle = einfache Python Programme

Einschienenlaufkatze = Pendel mit Schubgelenk



- Körper: Masse, Schwerpunkt, Trägheitstensor
- Koordinatensysteme: Lage und Orientierung
- Gelenke (Verbindungen)
Translation: T_x, T_y, T_z
Rotation: R_x, R_y, R_z
- Lastelemente: Aufbringen externe Lasten (Kraft, Moment)
- Sensoren (Messen von Systemgrößen)

Mehrkörpersimulation Einschienenlaufkatze mit PyMbs





»Wissen schafft Brücken.«