Inhalt

Module und Pakete

Sinn und Zweck:

- Bereitstellung von Funktionen
- Strukturierung von Programmen
- Wiederverwendung
- einfache Weitergabe

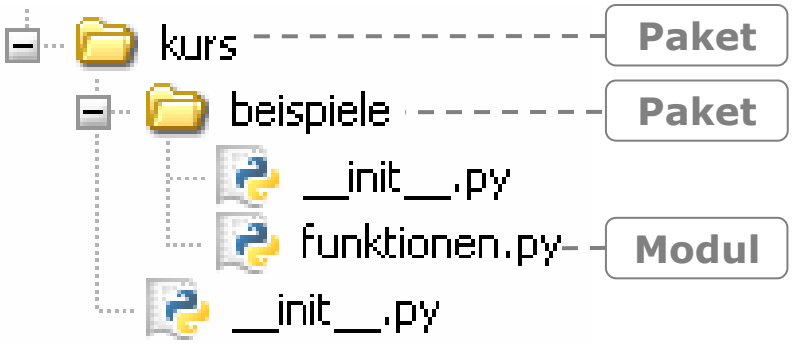
Umsetzung in Python:

- Orientiert sich am Dateisystem
- Modul = Python-Datei (*.py, *.pyc)
- Paket = Ordner mit Python-Dateien
- Pakete brauchen __init__.py

Woher kommen Pakete?

- Standardbibliothek – bei jeder Python Installation dabei
- externe – PyPI (Python Package Index)
- selbst erstellt
- müssen per PYTHONPATH Umgebungsvariable bekannt sein

Beispiel:



Wichtig: Namen für Ordner und Pakete müssen gültige Bezeichner sein!

Einbinden von Modulen und Paketen: `<import>` - Statement

```
>>> import kurs
```

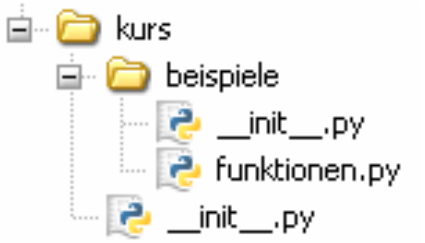
```
>>> import kurs.beispiele.funktionen
>>> kurs.beispiele.funktionen.summe(1,2)
```

```
>>> import kurs.beispiele.funktionen as fkt
>>> fkt.summe(1,2)
```

```
>>> from kurs.beispiele.funktionen import *
>>> differenz(2,1) + summe(1,2)
```

```
>>> from kurs.beispiele.funktionen import summe, differenz
```

```
>>> m = __import__('kurs.beispiele.funktionen', fromlist=['summe'])
>>> m.summe(1,2)
```



Achtung: imports überschreiben bereits existierende Namen!

Eine (kleine) Auswahl aus der Standardbibliothek:

os, sys	Betriebs- und Dateisysteminteraktion
datetime, time, calender	Datums-, Zeit- und Kalenderfunktionen
re	Regular Expressions
struct	Interpretieren von Strings als Binärdaten
types	Namen für Datentypen (Exceptionhandling)
copy	Kopieren von Instanzen
math, random	Mathematische Funktionen, Zufallsgenerator
ctypes	Einbinden externer Funktionen (dll, so)
thread, threading	Arbeiten mit threads
subprocess	Externe Prozesse
csv, xml, htmllib	Dateiformate lesen und schreiben
pickle	Serialisierung von Python Objekten
ssl, socket	Netzwerkfunktionen

Alle Pakete: http://docs.python.org/library/

Infos über System und OS

```
>>> import platform
>>> platform.machine()          Prozessorarchitektur
>>> platform.processor()        Prozessortyp
>>> platform.architecture()     32/64 bit, Betriebssystemtyp
>>> platform.version()          Betriebssystemversion
>>> platform.platform()         komplette Bezeichnung Betriebssystem
>>> platform.node()             Rechnername
>>> platform.uname()            Alle Infos zusammen
```

```
>>> import getpass
>>> getpass.getuser()           aktuell angemeldeter Benutzer
```

Standardpaket für Betriebssysteminteraktion: `import os`

- Ausführen von Prozessen
- Dateioperationen
- Zugriffsrechte
- Arbeiten mit Dateipfaden: `os.path`
- für Pfade und Dateinamen werden immer Strings verwendet

Rechte unseres Programms prüfen

```
>>> os.access(path, mode)
```

Rechte ändern mit

```
>>> os.chmod(path, mode)
```

mode	Bedeutung
os.F_OK	Existiert der Pfad?
os.R_OK	Leserechte?
os.F_OK	Schreibrechte?
os.X_OK	Ausführbar?

Arbeiten mit dem Dateisystem

`>>> os.chdir(path)`	Wechsel aktuelles Arbeitsverzeichnis
`>>> os.getcwd()`	Gibt aktuelles Arbeitsverzeichnis
`>>> os.listdir(path)`	Gibt den Inhalt eines Verzeichnisses als Liste von Strings zurück
`>>> os.mkdir(path [,mode])`	Legt ein Verzeichnis an
`>>> os.mkdirs(path [,mode])`	Legt einen Verzeichnisbaum an
`>>> os.remove(path)`	Löscht **Datei**
`>>> os.removedirs(path)`	Löscht Struktur aus **leeren Ordnern**
`>>> os.rmdir(pahth)`	Löscht **Verzeichnis**
`>>> os.rename(src, dst)`	Umbenennung
`>>> os.environ()`	Dictionary mit Umgebungsvariablen

Alternativ kann für Dateioperationen auch das Paket `shutils` verwendet werden

Arbeiten mit Pfaden: `os.path`

`>>> os.path.abspath(path)`	Absoluter Pfad von path
`>>> os.path.basename(path)`	Extrahiert Dateinamen aus Pfadangabe
`>>> os.path.dirname(path)`	Extrahiert Verzeichnisname aus Pfadangabe
`>>> os.path.exists(path)`	Existiert der Pfad?
`>>> os.path.isdir(path)`	Ist path ein Verzeichnis?
`>>> os.path.isfile(path)`	Ist path eine Datei?
`>>> os.path.split(path)`	Teilt path in Verzeichnis und Dateinamen
`>>> os.path.getsize(path)`	Größe der Datei in Bytes

Python Suchpfad und Kommandozeilenparameter: `sys`

`>>> sys.path`	Liste der Suchpfade für Python Pakete
`>>> sys.argv`	Liste der Kommandozeilenparameter des Skriptes

Übungsaufgaben

- Inhalt des aktuellen Verzeichnisses ausgeben
- Nur Unterverzeichnisse des aktuellen Verzeichnisses ausgeben
- Nur bestimmte Dateien im aktuellen Verzeichnis ausgeben (.py, .txt) (Tip: filter())
- Dateien im Arbeitsverzeichnis nach der Gößе sortiert ausgeben
- Eine neues Verzeichnis im Arbeitsverzeichnis erstellen und eine beliebige Datei hineinkopieren
- Die Systemvariable PATH als sortierte Liste ausgeben
- Ein Programm schreiben, dass zwei per Kommandozeile übergebene Zahlen addiert.

Dateien erstellen/öffnen/bearbeiten

```
>>> fid = open(path [,mode])
```

mode	Bedeutung
'r' (Standard)	Nur Lesezugriff
'w'	Nur Schreiben, vorhandene Datei wird gelöscht
'a'	Nur Schreiben, Vorhandene Datei wird erweitert
'rb', 'wb', 'ab'	Jeweils im Binärmodus

Lesen

`>>> fid.read(n)`	liest n Bytes aus der Datei
`>>> fid.readline()`	liest eine Zeile aus der Datei
`>>> fid.readlines()`	liest alle Zeilen, ergibt Liste mit Strings

Schreiben

`>>> fid.write(str)`	schreibt String in Datei
`>>> fid.writelines()`	schreibt Liste mit Strings zeilenweise in Datei

Iterieren

- Dateiobjekt ist iterierbar, d.h. es kann Zeilenweise durchlaufen werden:

```
for line in fid:
    print line
```

Navigieren (meist in binären Dateien erforderlich)

`>>> fid.tell()` gibt aktuelle Position in der Datei wieder

`>>> fid.seek(n [,w])` springt in der Datei um n Bytes ausgehend von w

w	Bedeutung
0	Relativ zum Dateianfang
1 (Standard)	Relativ zur aktuellen Position
2	Relativ zum Dateiende

Datei schließen

```
>>> fid.close()
```

Strings als Binärdaten interpretieren: `struct.unpack`

- gibt Tupel mit Werten entsprechend Formatstring zurueck
- Beispiel: 2 Byte als short interpretieren:

```
>>> struct.unpack('h', '\x00\x00')
>>> (0,)
```

- Infos und Tabelle mit Formatstrings: http://docs.python.org/library/struct.html
- Gegenstück: `struct.pack` → Wandelt Strings in Binärdatenrepräsentation um

Pakete für spezielle Dateiformate (Auszug)

gzip, zipfile	Direkt in Archiven schreiben und lesen
xml.dom, xml.sax	XML-Parser
csv	Comma Separated Values
scipy.io	Matlab
h5py	Hdf5
pyphant.fmfile	FMF Format
xlutils	Excel Dateien erstellen, lesen, bearbeiten
ConfigParser	Konfigurationsdateien erstellen und lesen

Das Modul `pickle`

- Serialisierbare Objekte können direkt als Bytestream in Datei geschrieben werden (ein Objekt pro Datei)

```
import pickle
```

- Speichern:

```
a = 'Knacker'
fid = open('datei.pkl', 'w')
pickle.dump(a, fid)
fid.close()
```

- Laden:

```
fid = open('datei.pkl')
b = pickle.load(fid)
fid.close()
```

Übungsaufgaben

- Datei mit zeilenweisen Zahlen von 1 bis 1000 anlegen und wieder lesen
- Dictionary in Textdatei schreiben und wieder auslesen
- Dictionary mit pickle in Datei schreiben und wieder auslesen
- Datei mit Zahlen 1 bis 1000 in Binärform speichern und auslesen. Was fällt im Vergleich zur ersten Aufgabe auf?

Programme ausführen

Kommandozeilenbefehle wie in Shell ausführen:

```
>>> os.system(command)
```

geht, ist aber veraltet, ebenso wie

```
>>> os.popen(command)
```

Externen Prozess starten: `subprocess`

- light Variante:

```
>>> retCode = subprocess.call([executable, args])
```

- volle Packung, damit können bspw. Ausgaben des Prozesses für Logging oder Fehleranalyse in Datei umgeleitet werden:

```
>>> p = subprocess.Popen(exec, args, stdin, stdout, stderr, **kwargs)
```

Übungsaufgaben

- Eine Pdf Datei öffnen
- Webseite mit Firefox öffnen
- Das Programm aus erstem Übungsblock zum addieren zweier Zahlen (als Kommandozeilenparameter) ausführen

Folie 6:

mode	**Bedeutung**
os.F_OK	Existiert der Pfad?
os.R_OK	Leserechte?
os.W_OK	Schreibrechte?
os.X_OK	Ausführbar?

Folie 7:

os.environ → ist keine Funktion sondern Variable (d.h. ohne Klammern)

Folie 10:

Modi unvollständig:

mode	**Bedeutung**
'r' (Standard)	Nur Lesezugriff
'w'	Nur Schreiben, vorhandene Datei wird gelöscht
'a'	Nur Schreiben, Vorhandene Datei wird erweitert
'r+', 'w+', 'a+'	Datei wird zum Lesen und Schreiben geoeffnet
'rb', 'wb', 'ab', 'r+b', 'w+b', a+b'	Jeweils im Binärmodus