Vorbemerkungen

```
ssh-keygen # ein paar mal Enter drücken..
cd .ssh
cat id_rsa.pub >> authorized_keys
gedit config
```

In die Datei config nun folgendes eintragen:

```
Host zeus
User <fünfstelliger Username>
HostName nf2-zeus.physik.uni-halle.de
```

Jetzt können wir uns auf den Zeus-Server einloggen und Python einrichten:

```
ssh zeus
```

Module

Nachdem wir in der letzten Woche gelernt haben, wie man den selbstgeschriebenen Code besser organisieren kann durch Einsatz von Funktionen und Klassen, wird heute gezeigt, wie man nun Dateien mit Code als sogenannte Module speichern kann.

Importieren aus anderer Python-Datei

Um aus einer weiteren Pythondatei Klassen, Funktionen, oder Variablen zu laden benutzt man

```
import <Dateiname ohne .py>.<Klassen/Funktions/Variablenname>
```

Wichtig: damit das funktioniert, muss der Python-Interpreter im selben Ordner gestartet worden sein wie die Datei aus der importiert wurde!

Importieren aus Ordnerstruktur

Um aus einer komplexeren Ordnerstruktur importieren zu können, geht man wie folgt vor:

```
import <Ordner>.<Unterordner>.<Unterunterordner>.<...>.<Dateiname>.<Klassen/Funktions/Variab
```

from ... import ...

Möchten wir etwas direkt in den aktuellen Namespace importieren, benutzen wir *from ... import*

import ... as ...

Mittels *import ... as ...* können wir dem importierten Modul einen anderen (oft gekürzten) Namen geben.

Beispiel

Die aus einer Übung bekannten Funktionen *write_file* und *read_file* haben wir in einer Datei *input_output_file.py* gespeichert, und sind so von ihnen überzeugt, dass wir sie in Zukunft öfters benutzen möchten.

Sind wir im selben Ordner wie die Datei, ist es ganz einfach, von einer anderen Python-Datei aus auf den schon vorhandenen Code zuzugreifen: wir schreiben einfach

```
import input_output_file # Achtung, die Endung .py nicht schreiben!
```

und können dann folgendermaßen auf die Funktionen (sowie Klassen und Variablen), die in der Datei *input_output_file.py* geschrieben wurden, zugreifen:

```
data = input_output_file.read_file("dateiname")
```

Wem es zu lästig ist, bei jedem Funktionsaufruf den Modulnamen zu schreiben, kann am Anfang folgendes tun:

```
from input_output_file import read_file, write_file # , ...

data = read_file("dateiname")
```

Auf diese Weise werden aber auch nur die Funktionen importiert, die wir explizit nach import auflisten. Wir können sie dann aber benutzen, ohne *input_output_file.* voranzustellen.

Es geht aber noch eine Stufe höher: Angenommen, wir haben nicht nur eine Datei für Input und Output, sondern noch eine weitere (z.B. für das Einlesen von komprimierten Daten wie Zip-Files zuständig sein). In diesem Fall bietet es sich an einen eigenen Ordner *input_output* anzulegen, und in diesen die beiden Python-Dateien zu legen.

Ganz ähnlich wie zuvor wollen wir nun die Funktionen read_file und write_file importieren. Um das tun zu können, müssen wir es Python jedoch möglich machen, den Ordner *input_output* als Modul zu erkennen. Das tun wir, indem wir eine leere Datei mit dem Namen *___init___.py* im Ordner *input_output* anlegen. Wir können unseren Code in beliebig tief geschachtelten Ordnerstrukturen vergraben, solange wir nur daran denken, in jedem Ordner eine *___init___.py* Datei anzulegen.

Danach können wir, wenn wir uns außerhalb des Ordners befinden, per

```
from input_output.input_output_file import read_file, write_file
```

die Funktionen *read_file* und *write_file* wieder importieren.

Alternativ:

```
import input_output.input_output_file
```

Oft ist es ratsam, die Funktionen/Klassen/Variablen nicht direkt in den globalen Namespace zu laden, da es dann zu Konflikten mit schon bestehenden Namen kommen kann.

Um in diesem Fall zu viel Schreibarbeit aufgrund langer Modulnamen zu vermeiden, lässt sich folgender Trick anwenden:

```
import input_output.input_output_file as iof # 'as' keyword gefolgt von Aliasnamen
```

Im weiteren Verlauf lässt sich dann per

```
iof.read_file("bla")
```

auf die Funktionen zugreifen.

Selbstgeschriebene Module für den Python-Interpreter sichtbar machen

Haben wir nun ein selbsterstelltes Modul irgendwo auf unserer Festplatte liegen, müssen wir nun Python noch sagen, wo es sich befindet. Das machen wir, indem wir den Speicherort zum *Pythonpath* hinzufügen.

Die einfachste Variante ist vermutlich, den Pythonpfad direkt im Code zu setzen:

```
import sys
sys.path.append("/Pfad/zum/Ordner/in/dem/das/Modul/liegt")
import mein_modul # ersetze mein_modul durch gewünschten Modulnamen
```

Möchte man den Pfad dauerhaft hinzufügen, tun Linux-User folgendes im Terminal:

```
cd ~
echo 'export PYTHONPATH="/Pfad/zu/meinem/Modul:$PYTHONPATH"' >> .bashrc
```

Windows-User können folgendes ausprobieren:

In die Datei autoexec.bat wird folgende Zeile geschrieben:

```
set PYTHONPATH=%PYTHONPATH%;C:\Pfad/zu/meinem/Modul
```

Die Python Standard Library

Netterweise kommt Python schon mit einer Menge eigener Module daher. Die wichtigsten (persönliche Meinung des Dozenten) werden hier kurz vorgestellt:

sys

Das Modul sys enthält vor allem Variablen, die vom Python-Interpreter benutzt werden oder mit ihm interagieren.

Hilfreich sind vor allem

- **sys.argv**: Diese Variable speichert die Kommandozeilenargumente, mit denen ein Programm aufgerufen wurde.

Folgendes Programm sei unter dem Namen main.py gespeichert:

“‘python import sys

print “ich wurde aufgerufen mit:”, sys.argv ““

Starten wir dieses Skript folgendermaßen:

```
bash   python main.py hier ganz viele kommandozeilen argumente 1
2 3 4 5
```

so ist die Ausgabe:

```
ich wurde aufgerufen mit: ['test.py', 'hier', 'ganz', 'viele',
'kommandozeilen', 'argumente', '1', '2', '3', '4', '5']
```

Alle Argumente werden also als Strings in einer Liste gespeichert.

- **sys.path**: In dieser Variablen werden alle Pfade gespeichert, in denen der Interpreter nach nutzbaren Python-Modulen sucht. Möchte man einen weiteren Pfad hinzufügen, tut man das einfach per

```
python   sys.path.append("/mein/neuer/pfad")
```

Der neu hinzugefügte Pfad verschwindet allerdings wieder aus sys.path wenn das Skript endet!

- **sys.exit()**: Diese Funktion beendet das Skript sofort.

pickle

Erlaubt es, Python-Objekte zu serialisieren, d.h. sie als Datei zu speichern.

Beispiel:

```
import pickle

class MeineKlasse:
    def __init__(self):
        self.x = 1
        self.y = 2

mk = MeineKlasse()
```

```
with open("meine_klasse.pickle", "w") as f:
    pickle.dump(mk, f)
```

Das gespeicherte Objekt kann anschließend folgendermaßen geladen werden:

```
with open("meine_klasse.pickle", "r") as f:
    pickle.load(f)
```

collections

In diesem Modul sind diverse nützliche Container-Datentypen, die teilweise Erweiterungen der Standard-Container dict, list und tuple sind.

- **Counter**: Damit lassen sich Elemente sehr einfach zählen. Beispiel mit dem Text aus der Übung *word_counter*:

“‘python from collections import Counter

c = Counter(text.split()) print c.most_common(10) # gibt die 10 häufigsten Wörter in text aus “‘

- **deque**: Eine Art Liste, bei der sich effizient an beiden Enden Elemente hinzufügen oder entfernen lassen.
- **defaultdict**: Ähnlich wie dict, aber legt für einen Key, der bisher nicht gespeichert war, automatisch einen entsprechenden Default-Value an.

Beispiel: “‘python from collections import defaultdict

int_dict = defaultdict(int) print int_dict[“blub”] # gibt den Defaultwert 0 aus

list_dict = defaultdict(list): print list_dict[“irgendwas”] # gibt eine leere Liste aus “‘

- **OrderedDict**: Ein Dictionary, das sich die Reihenfolge der eingefügten Elemente merkt

types

Hilfreiches Modul, falls man doch mal gezwungen ist, Datentypen zu prüfen.

Beispiel:

```
import types

x = 3
y = 1.23
z = True

def f():
```

```
    print "hello world"

print type(x) is types.IntType # True
print type(y) is types.FloatType
print type(z) is types.BooleanType
print type(f) is types.FunctionType
```

math

Dieses Modul enthält die wichtigsten mathematischen Operationen Für Berechnungen komplexer Zahlen, benutze man cmath

```
from math import sin, cos, exp, log, pi

sin(pi)**2 + cos(pi)**2
exp(0)
log(1)
# usw.
```

random

Modul zur Erzeugung von Zufallszahlen

Die wichtigsten Funktionen im Überblick:

```
from random import random, uniform, randint, shuffle, choice

random() # erzeugt eine gleichverteilte Pseudozufallszahl zwischen 0 und 1 (ohne die 1).
uniform(3.4, 7.8) # erzeugt eine gleichverteilte Pseudozufallszahl (float) zwischen den gege

def wuerfel():
    return randint(1, 6) # erzeugt zufällig (und gleichverteilt) einen Integer zwischen 1 u

l = [1, 2, 3, 4, 5]
shuffle(l) # vertauscht die Elemente von l zufällig
choice(l) # zieht zufällig ein Element aus l
```

time

Dieses Modul beinhaltet diverse Funktionen, die für Zeitmessungen benutzt werden können.

Beispiel:

```
import time
```

```
print time.time() # Gibt die Sekunden an, die seit dem 1. Januar 1970 vergangen sind
time.sleep(5) # Das Programm stoppt für 5 Sekunden
```

Für Datumsberechnungen ist darüberhinaus das Modul datetime hilfreich.

argparse

Möchte man ein Programm mit mehreren Parametern von der Kommandozeile aufrufen, wird es recht bald komplex, die Eingaben des Users auf Korrektheit zu prüfen und richtig zu parsen.

Hier hilft das Modul argparse.

Beispiel:

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument("argument") # hier wird festgelegt, dass es genau ein notwendiges Argume
args = parser.parse_args()
```

Im obigen Beispiel haben wir festgelegt, dass das Programm mit genau einem Parameter aufgerufen werden muss. Nicht mer und nicht weniger ist möglich.

Manchmal ist es aber auch hilfreich, optionale Parameter festzulegen. Das geht so:

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument("--argument") # hier wird festgelegt, dass es genau ein notwendiges Argu
args = parser.parse_args()

if args.argument:
    print args.argument
else:
    print "Kein Argument angegeben"
```

Dieses Mal können wir das Programm entweder ohne jeglichen Parameter aufrufen, oder wir rufen es folgendermaßen auf (unter der Annahme, dass die Datei main.py heißt):

```
python main.py --argument irgendwas
```

pdb

pdb ist der Python Debugger. Er ist ein sehr hilfreiches Tool, um Fehler im eigenen Python-Code zu finden.

Typischerweise ruft man den Debugger auf, indem man an der Stelle seines Codes, an der man etwas inspizieren will, folgende Zeile schreibt:

```
pdb.set_trace()
```

An dieser Stelle hält der Debugger das Programm an und wechselt in den interaktiven Modus, in dem man sich die Werte aller Variablen anschauen kann, aber auch neue Variablen definieren oder Funktionen ausführen kann.

Mit folgenden Befehlen kann man weiter durch den Code navigieren:

- *next* führt die nächste Zeile des Programms aus und pausiert dann wieder. Enthält die nächste Zeile einen Funktionsaufruf, wird die komplette Funktion ausgeführt, ohne dass in ihr gestoppt wird
- *step* ist wie _next, "betritt" aber auch Funktionen wenn sie aufgerufen werden
- Befindet man sich in einer Funktion, sorgt *return* dafür, dass das Programm erst wieder angehalten wird, wenn das return statement der Funktion erreicht wird.
- *until* führt den Code weiter aus
- *until* führt den Code weiter aus, bis die angegebene Zeilennummer erreicht wird
- *jump* erlaubt schließlich noch das direkte Springen zu einer beliebigen Zeilennummer

Weitere Module installieren

Es gibt noch unzählige weitere Module für Python, die sich leicht nachinstallieren lassen.

Ubuntu- und Debian-User mit Administratorrechten finden viele Pakete in ihrem Paketmanager, z.B. python-numpy, python-scipy, ...

Alternativ ist es möglich eine sogenannte virtualenv zu benutzen. Diese kopiert den Python-Interpreter, der oftmals in einem geschützten Pfad liegt, in das Home-Verzeichnis des Benutzers. Das erleichtert die Modul-Installation deutlich.

Alle mit Linux können folgendes tun:

```
cd ~
mkdir virtualenvs
virtualenv virtualenvs/pythonkurs
source virtualenvs/pythonkurs/bin/activate
```

Nach diesen vier Schritten wird der Python-Interpreter benutzt der im Verzeichnis ~/virtualenvs/pythonkurs/bin/ liegt, benutzt.

Nun kann man Module mittels *pip* installieren:

```
pip install <Modulname>
```

Exceptions

Bisher gingen wir immer davon aus, dass unsere Programme fehlerfrei laufen, oder haben Fehlermeldungen erstmal ignoriert.. Hier wird nun gezeigt, wie man seinen Code so schreiben kann, dass er auf Fehler direkt reagiert.

Beispiel:

```
zahl = float(raw_input("Bitte Zahl eingeben: "))
```

Führen wir dieses Zahl aus und geben brav eine Zahl ein, kommt es zu keinen Problemen. Hat der Anwender jedoch nicht aufgepasst und gibt z.B. “Hallo” ein, wird er von folgender Meldung überrascht:

```
ValueError: could not convert string to float: Hallo
```

Ganz logisch, float() kann den String “Hallo” nicht in eine Zahl umwandeln, daher erscheint ein sogenannter ValueError.

Nun möchten wir aber gerne ein anderes Verhalten unseres Codes. Wir möchten, dass er den User so lange nach einer Zahl fragt, bis er sie bekommt. Die Lösung könnte so aussehen:

```
while True:
    try:
        zahl = float(raw_input("Bitte Zahl eingeben: "))
        print 5 / zahl
        break
    except ValueError:
        print "Hey, das war gar keine Zahl!"
```

Damit hätten wir schonmal das Problem der falschen Benutzereingabe gelöst. Allerdings gibt es immer noch eine Möglichkeit, das Programm zum Absturz zu bringen: Gibt der User 0 ein, grüßt ihn der **ZeroDivisionError**!

Also fangen wir auch den ab:

```
while True:
    try:
        zahl = float(raw_input("Bitte Zahl eingeben: "))
        print 5 / zahl
        break
    except ValueError:
        print "Hey, das war gar keine Zahl!"
```

```
    except ZeroDivisionError:
        print "Null ist keine gute Idee..."
```

Damit haben wir uns erstmal gegen alle Gefahren abgesichert. Eine Liste der möglichen Exceptions gibt es hier.

Doch damit nicht genug, wir können auch selbst Exceptions “raisen”, d.h. Code auf Probleme mit einer Exception reagieren lassen:

```
import types
def add_integers(a, b):
    if type(a) != types.IntType or type(b) != types.IntType:
        raise ValueError("So nicht, mein Freund!")
    else:
        return a + b
```

Diese Funktion beschwert sich beispielsweise lautstark mit einem ValueError, sobald wir ihr ein Argument liefern, das kein Integer ist. Die Wahl der Exception ist dabei uns überlassen