Module

- Funktionalität zusammenfassen in einer separaten `.py` Datei
- Importieren Bibliotheksmodul (Schon verwendet: `math`)

Module

- Funktionalität zusammenfassen in einer separaten `.py` Datei
- Importieren Bibliotheksmodul (Schon verwendet: `math`)
- Beispiel (`tools.py`):

```
"""This module provides some helper tools.
Try it."""

counter = 42

def printCounter():
  "just prints counter value to command line"
  print counter

def do_nothing():
    "Do really nothing"
    pass
```

Importieren

- Modul importieren

```
import tools
tools.do_nothing()
print tools.counter
```

Importieren

- Modul importieren

```
import tools
tools.do_nothing()
print tools.counter
```

- Modul unter neuem Namen importieren

```
import tools as t
t.do_nothing()
```

Importieren

- Modul importieren

```
import tools
tools.do_nothing()
print tools.counter
```

- Modul unter neuem Namen importieren

```
import tools as t
t.do_nothing()
```

- Einzelne Funktionen direkt importieren

```
from tools import do_nothing, printCounter
from tools import counter as cntr
do_nothing()
printCounter()
print cntr
```

Import beeinflussen

- Alle Symbole in den Namensraum importieren

```
from tools import *
do_nothing()
print counter
```

- Module können bestimmen, welche Symbole mit `from module import *` importiert werden:

```
# module tools.py
__all__ = ['printCounter', 'counter']
```

Die Funktion `do_nothing()` ist nach `import *` nicht bekannt!

Import beeinflussen

- Alle Symbole in den Namensraum importieren

```
from tools import *
do_nothing()
print counter
```

- Module können bestimmen, welche Symbole mit `from module import *` importiert werden:

```
# module tools.py
__all__ = ['printCounter', 'counter']
```

Die Funktion `do_nothing()` ist nach `import *` nicht bekannt!

Funktionalität abfragen

-

```
dir(tools)
dir(math)
```

Hilfe zu Modulen und Funktionen

- Docstrings abfragen mit `help`

```
import tools
help(tools)
help(tools.do_nothing)
```

Hilfe zu Modulen und Funktionen

- Docstrings abfragen mit `help`

```
import tools
help(tools)
help(tools.do_nothing)
```

Ausführbares Programm als Modul?

- `tools.py` soll sowohl Bibliothek sein, als auch selbst ausführbar

```
# tools.py
...
if __name__ == '__main__':
    print "tools.py executed"
else:
    print "tools.py imported as module"
```

Module debuggen

- Wenn ein Modul geändert wird, werden Änderungen nur wirksam, wenn Python neu gestartet wird, oder `reload` (Python$<$3.0) verwendet wird.

```
reload(tools)
```

Module debuggen

- Wenn ein Modul geändert wird, werden Änderungen nur wirksam, wenn Python neu gestartet wird, oder `reload` (Python<3.0) verwendet wird.

```
reload(tools)
```

Pfad für Module

- Python sucht im Suchpfad und im aktuellen Verzeichnis
- Suchpfad kann erweitert werden

```
import sys
sys.path.append("folder/to/module")
import ...
```

Pakete

- Module können gruppiert werden
- Hierbei ist die Verzeichnisstruktur wichtig

```
tools_Paket/
    __init__.py      # Inhalt für "import tools"
    files.py         # für "import tools.files"
    graphics.py      # für "import tools.graphics"
    stringtools/
        __init__.py  # für "import tools.stringtools"
        ... weitere Verschachtelung möglich
```

- Wenn `from tools_Paket import *` Untermodule enthalten soll, muss `tools_Paket/__init__.py` folgende Zeile enthalten:

```
__all__ = ["files", "graphics"]
```

Pakete

```
from tools_Paket import *
files.foo()
graphics.bar()
```

Hands-On: Erstellen Sie ein Modul MyBelovedPhysics.py:

- Das Modul enthält zwei Funktionen
  - `physics()`: gibt den Text “I love physics” aus
  - `informatics()`: gibt den Text “Informatics is great, too” aus
- Importieren Sie das Modul unter dem Namen mbp
- Rufen Sie jede Funktion einmal auf

Hands-On: Erstellen Sie ein Modul MyBelovedPhysics.py:

- Das Modul enthält zwei Funktionen
  - `physics()`: gibt den Text “I love physics” aus
  - `informatics()`: gibt den Text “Informatics is great, too” aus
- Importieren Sie das Modul unter dem Namen mbp
- Rufen Sie jede Funktion einmal auf

MyBelovedPhysics.py:

```
def physics():
  print "I love physics"

def informatics():
  print "Informatics is great, too"
```

Aufruf der Funktionen:

```
  import MyBelovedPhysics as mbp
  mbp.physics()
  mbp.informatics()
```

Teil V

IO und weitere Datentypen

Datei Ein- und Ausgabe

Dateien öffnen

- Textuelles Datei-Objekt erstellen

```
datei = open("testfile.txt")          # read
datei = open("testfile.txt", 'r')     # read
datei = open("testfile.txt", 'w')     # write
datei = open("testfile.txt", 'a')     # append
```

Datei Ein- und Ausgabe

Dateien öffnen

- Textuelles Datei-Objekt erstellen

```
datei = open("testfile.txt")          # read
datei = open("testfile.txt", 'r')     # read
datei = open("testfile.txt", 'w')     # write
datei = open("testfile.txt", 'a')     # append
```

- Binäres Datei-Objekt erstellen

```
datei = open("testfile.txt", 'rb')    # read
datei = open("testfile.txt", 'wb')    # write
datei = open("testfile.txt", 'ab')    # append
```

- `open` Hat noch weitere Optionen (Codierung, Behandlung von newlines, ...)

Methoden von Datei-Objekten

- Lesen

```
datei.read()        # komplett einlesen
datei.read(n)       # n Byte einlesen
datei.readline()    # eine Zeile lesen
datei.readlines()   # Liste von Zeilen
```

Methoden von Datei-Objekten

- Lesen

```
datei.read()        # komplett einlesen
datei.read(n)       # n Byte einlesen
datei.readline()    # eine Zeile lesen
datei.readlines()   # Liste von Zeilen
```

- Schreiben

```
datei.write("Neuer␣Text\n")
datei.writelines(["Erste␣Zeile\n", "Zweite␣Zeile\n"])
```

Methoden von Datei-Objekten

- Lesen

```
datei.read()        # komplett einlesen
datei.read(n)       # n Byte einlesen
datei.readline()    # eine Zeile lesen
datei.readlines()   # Liste von Zeilen
```

- Schreiben

```
datei.write("Neuer␣Text\n")
datei.writelines(["Erste␣Zeile\n", "Zweite␣Zeile\n"])
```

- Nicht vergessen: newlines (`\n`)
- Datei schließen

```
datei.close()
```

Textdateien zeilenweise bearbeiten

- `for` Schleife über die Zeilen der Datei

```
datei = open("fulltext.txt")
for line in datei:      # alt.: in datei.readlines()
    print line
# oder:
while True:
    line = datei.readline()
    if not line:
        break
    print line
```

Textdateien zeilenweise bearbeiten

- `for` Schleife über die Zeilen der Datei

```
datei = open("fulltext.txt")
for line in datei:      # alt.: in datei.readlines()
    print line
# oder:
while True:
    line = datei.readline()
    if not line:
        break
    print line
```

Weitere Methoden von Dateiobjekten

- `datei.tell()` – aktuelle Position ausgeben
- `datei.seek(pos)` – Zur gegebenen Position gehen
- `datei.flush()` – Ausgabepuffer leeren (Pufferung wegen Effizienz)

Standard Input, Output, und Error

- Modul `sys` stellt drei Standard-Dateiobjekte bereit
  - `sys.stdin` – nur lesen
  - `sys.stdout`, `sys.stderr` – nur schreiben

```
import sys

sys.stdout.write("Text␣eingeben:␣")   # = print "..."
line = sys.stdin.readline()
# oder: line = raw_input("Text eingeben: ")

if error:
    sys.stderr.write("Error!\n")
```

- `write` Hängt nicht automatisch einen Zeilenumbruch an
- `raw_input([text])` entfernt Zeilenumbrüche
- Ein- und Ausgabe sind gepuffert

Kommandozeilenparameter

sys.argv

- `sys.argv` ist eine Liste mit Kommandozeilenparametern
- Erstes Element (`sys.argv[0]`) ist der Programmname

```
import sys
print "Programmname: %s" % (sys.argv[0])
if len(sys.argv) < 2:
    print "Keine Parameter angegeben"
    sys.exit(1)
print "Parameters:"
print ", ".join(sys.argv[1:])
```

- Bei sehr wenigen Parametern ist diese Methode praktikabel
- Schlecht bei vielen Parametern, oder wenn Parameter nur optional sein sollen
  $\Rightarrow$ Modul `argparse`

nochmal Sequenzen: Dictionaries und Sets

Bisherige Container zur Speicherung größerer Datenmengen

- Tupel: unveränderlich
- Liste: veränderlich
- Beide werden indiziert mit `0...n-1` (`n` Anzahl an Elementen)

nochmal Sequenzen: Dictionaries und Sets

Bisherige Container zur Speicherung größerer Datenmengen

- Tupel: unveränderlich
- Liste: veränderlich
- Beide werden indiziert mit `0...n-1` (`n` Anzahl an Elementen)

Vorteile

- Tupel: sehr effizient, da unveränderlich
- Liste: höherer Overhead, aber immer noch effizient
- Optimal, wenn eine Folge von Werten gespeichert werden soll

nochmal Sequenzen: Dictionaries und Sets

Bisherige Container zur Speicherung größerer Datenmengen

- Tupel: unveränderlich
- Liste: veränderlich
- Beide werden indiziert mit `0...n-1` (`n` Anzahl an Elementen)

Vorteile

- Tupel: sehr effizient, da unveränderlich
- Liste: höherer Overhead, aber immer noch effizient
- Optimal, wenn eine Folge von Werten gespeichert werden soll

Beschränkungen

- Keine Mengen darstellbar
- Keine Bezeichner für Objekte möglich
  z.B. suche Buch mit bestimmter ISBN oder bestimmtem Titel

Dictionary

- Bildet einen Schlüssel (key) auf einen Wert (value) ab
- Werte können beliebigen Typ haben
- Schlüssel nur unveränderliche Objekte (int, float, string, tuple,...)
- Kann als primitive Datenbank verwendet werden
- Erstellung mit geschweiften Klammern
- Optional in den Klammern: Folge von `key:value` Paaren

```
>>> nummern = {}
>>> nummern["Tobias"] = "089␣289␣18632"
>>> nummern["Alfredo"] = "089␣289␣16829"
>>> nummern["Alfredo"]
"089␣289␣18629"
```

Methoden von Dictionaries

```
>>> d = {7:'A', 'Muh': 'Maeh', (4, 5, 6):[6, 5, 4]}
>>> d.keys()
[(4, 5, 6), 'Muh', 7]
>>> d.values()
[[6, 5, 4], 'Maeh', 'A']
>>> d.items()
[((4, 5, 6), [6, 5, 4]), ('Muh', 'Maeh'), (7, 'A')]
>>> del d['Muh']
>>> for (key, value) in d.items():
>>>     print key, value
>>> for i in d.items():
>>>     print i[0], i[1]
>>> for key in d.keys():
>>>     print key, d[key]
```

Fehlerhafte Zugriffe

- Tupel und Listen: gesamter Indexbereich mit Werten belegt
- Bei Dictionaries gibt es keinen Indexbereich
- Was passiert, wenn ein unbekannter Schlüssel gesucht wird?

```
>>> nummern['Andreas']
KeyError: 4
>>> nummern.has_key('Alfredo')
True
>>> 'Alfredo' in nummern
True
```

Fehlerhafte Zugriffe

- Tupel und Listen: gesamter Indexbereich mit Werten belegt
- Bei Dictionaries gibt es keinen Indexbereich
- Was passiert, wenn ein unbekannter Schlüssel gesucht wird?

```
>>> nummern['Andreas']
KeyError: 4
>>> nummern.has_key('Alfredo')
True
>>> 'Alfredo' in nummern
True
```

Standardwerte

- Methode `get` liefert `None` bei nicht existierendem Schlüssel
- Optionaler zweiter Parameter für `get` definiert Standardwert

```
>>> nummern.get('Alfredo')
"089 289 18629"
>>> nummern.get('Andreas', 'keine Nummer vorhanden')
'keine Nummer vorhanden'
```

Hands-On: Wörterbuch

- Schreiben Sie ein Wörterbuch `GutenMorgen`, dass die folgenden Begriffe enthält:
  “Morgen”$\rightarrow$ “Morning”, “Kaffee”$\rightarrow$ “Coffee”,
  “Ueberleben” $\rightarrow$ “Survive”
- Übersetzung soll nur von deutsch nach englisch möglich sein
- Eingabe des deutschen Wortes (als Kommandozeilenparameter oder über stdin) soll das englische Pendant zurückliefern
- Fehlermeldung falls deutsches Wort nicht bekannt

Hands-On: Wörterbuch

- Schreiben Sie ein Wörterbuch `GutenMorgen`, dass die folgenden Begriffe enthält:
  "Morgen"→ "Morning", "Kaffee"→ "Coffee",
  "Ueberleben" → "Survive"
- Übersetzung soll nur von deutsch nach englisch möglich sein
- Eingabe des deutschen Wortes (als Kommandozeilenparameter oder über stdin) soll das englische Pendant zurückliefern
- Fehlermeldung falls deutsches Wort nicht bekannt

```
import sys
GutenMorgen = { "Morgen":"Morning", \
                "Kaffee":"Coffee", \
                "Ueberleben":"Survive" }

myKey = sys.argv[1]
if GutenMorgen.has_key(myKey):
  print GutenMorgen[myKey]
else:
  print "Key␣unbekannt!"
```

Set

- Repräsentiert eine Menge
- Jedes Element ist maximal ein Mal in der Menge
- `set` ist veränderbar, Elemente müssen aber unveränderlich sein
- Kann initialisiert werden mit beliebiger Sequenz

Set

- Repräsentiert eine Menge
- Jedes Element ist maximal ein Mal in der Menge
- `set` ist veränderbar, Elemente müssen aber unveränderlich sein
- Kann initialisiert werden mit beliebiger Sequenz

```
set("Hallo")                    # set(['a', 'H', 'l', 'o'])
s=set([3,1,2,3,"Bla",2])        # set([1, 2, 3, 'Bla'])
l = [2,8,7]
s.difference(l)                 # set([1, 3, 'Bla'])
s.intersection(l)               # set([2])
s.union(l)                      # set([1, 2, 3, 7, 8, 'Bla'])
s.symmetric_difference(l)       # set([1, 3, 7, 8, 'Bla'])
s.issubset([3,"Bla"])           # False
s.issuperset([3,"Bla"])         # True
```

- Weitere Methoden: `add`, `remove`, ...
- bestimmte Operatoren auf sets unterstützt: -, ^, |, &

Interaktion mit dem Betriebssystem

Modul `os`

- Schnittstelle zu Funktionalitäten des Betriebssystems
- Variablen:

```
import os
os.environ # Dictionary mit Umgebungsvariablen
os.linesep # Zeilenseparator; \r\n win, \n linux
os.name    # Name des OS-spez. Moduls (posix, dos,...)
```

- Some general purpose methods

```
os.getcwd()     # current working directory
os.chdir(path) # wechseln ins Verzeichnis path
os.getgroups() # Gruppen (UNIX)
os.uname()      # Systeminformation (UNIX)
os.times()      # (usr, sys, c_usr, c_sys, wct)
...
```

OS Methoden für Dateien

```
chmod(path, mode)          # chmod
chmod("tmp.py", 0664)      # Modus erfordert vier Zeichen
listdir(path)              # ls
mkdir(path)                # mkdir
makedirs(path)             # mkdir -p
rename(src, dst)           # mv
remove(path)               # rm
rmdir(path)                # rmdir
removedirs(path)           # rm -rf
stat(path)                 # stats (atime, ...)
```

Beispiel: Sloan Digital Sky Survey

Aufgabe:

- Mehrere csv-Dateien mit Messdaten verschiedener Läufe (Runs)
- Eine Datei pro Run (Dateiname=Runname)
- Spalte mit spez. Messung (Obs4) wird ermittelt
- Methode zum Einlesen der Daten in interne Datenstruktur
- Grafische Darstellung der Daten
- Ermittlung von Mittelwerten etc.

Beispiel: Sloan Digital Sky Survey

Aufgabe:

- Mehrere csv-Dateien mit Messdaten verschiedener Läufe (Runs)
- Eine Datei pro Run (Dateiname=Runname)
- Spalte mit spez. Messung (Obs4) wird ermittelt
- Methode zum Einlesen der Daten in interne Datenstruktur
- Grafische Darstellung der Daten
- Ermittlung von Mittelwerten etc.

Folgendes wird geübt:

- Kontrollstrukturen
- Funktionen
- Module
- Dictionaries
- Dateizugriff
- Stringmanipulation mit `s.split(chr)`
- Reduce-Operationen (`min`, `max`, `sum`)

Format der csv-Dateien

```
u-g g-r r-i i-z istQuasar
3.8514e-01 4.1646e-01 3.8446e-01 3.7584e-01 0.0000e+00
5.7312e-01 4.7187e-01 3.8309e-01 4.1701e-01 0.0000e+00
4.1483e-01 4.1564e-01 3.8504e-01 3.7169e-01 0.0000e+00
4.0781e-01 4.1518e-01 3.8584e-01 3.7138e-01 0.0000e+00
6.0205e-01 5.5492e-01 4.4768e-01 4.1354e-01 0.0000e+00
3.3698e-01 3.7033e-01 3.6643e-01 4.0009e-01 1.0000e+00
4.8028e-01 4.4429e-01 3.9734e-01 3.7241e-01 0.0000e+00
5.6424e-01 5.2134e-01 4.2436e-01 3.9920e-01 0.0000e+00
4.1535e-01 4.2181e-01 3.9402e-01 3.8748e-01 0.0000e+00
...
```

Runs manuell erzeugt, Daten vgl.
`https://de.wikipedia.org/wiki/Sloan_Digital_Sky_Survey`
`http://cas.sdss.org/dr5/en/`
etc.

Nötige Module

- `os` für `listdir`, `pylab` zum plotten

```
import os
from pylab import *
```

Ermitteln aller Runs

- Im Verzeichnis liegen beliebig viele csv-Dateien

```
def liesRuns(dir):
  runliste = []
  dateien = os.listdir(dir)
  for dateiname in dateien:
    if dateiname[-4:] == ".csv":
      runliste.append(dateiname[:-4])
  runliste.reverse()
  print runliste
  return runliste
```

Liste mit Messwerten (Obs4) eines Runs einlesen

- Datei zum Lesen öffnen
- Datei zeilenweise durchgehen
- Elemente der vorletzten Spalte extrahieren

```
def liesMessung4(run):
  datei = open(run+'.csv', 'r')
  datei.readline() # Kopfzeile
  observations4 = []
  for zeile in datei:
    spalten = zeile.split('␣')
    obs = spalten[-2]
    observations4.append(float(obs))
  datei.close()
  return observations4
```

Automatisch alles einlesen

- Runnamen herausfinden
- Dateien Einlesen
- Speichern der Messwerte in einem Dictionary
- Berechnen von Mittelwerten etc.

```
def init():
  daten = {}
  runs = liesRuns(".")
  for run in runs:
    daten[run] = {}
    daten[run]['obs4'] = liesMessung4(run)
    daten[run]['avg_obs4'] = sum(daten[run]['obs4']) \
                             / len(daten[run]['obs4'])
    daten[run]['min_obs4'] = min(daten[run]['obs4'])
    daten[run]['max_obs4'] = max(daten[run]['obs4'])
  return daten
```

Plotten der Daten

- Details zum Plotten gibt es später

```
def zeichneObs4(daten, runliste):
  for run in runliste:
    plot(daten[run]['obs4'])
  show()
```

Plotten der Daten

- Details zum Plotten gibt es später

```
def zeichneObs4(daten, runliste):
  for run in runliste:
    plot(daten[run]['obs4'])
  show()
```

- Verwendung als Programm und Modul

```
if __name__ == '__main__':
  daten = init()
  maxlen = max([len(key) for key in daten.keys()])
  for run in daten.keys():
    name = run+" "*(maxlen-len(run))
    run_min = daten[run]['min_obs4']
    run_max = daten[run]['max_obs4']
    run_avg = daten[run]['avg_obs4']
    print "%s (min/max/avg): %6g, %6g, %6g" \
          %(name, run_min, run_max, run_avg)
```

Plotten der Daten

- Details zum Plotten gibt es später

```
def zeichneObs4(daten, runliste):
  for run in runliste:
    plot(daten[run]['obs4'])
  show()
```

- Verwendung als Programm und Modul

```
if __name__ == '__main__':
  daten = init()
  maxlen = max([len(key) for key in daten.keys()])
  for run in daten.keys():
    name = run+" "*(maxlen-len(run))
    run_min = daten[run]['min_obs4']
    run_max = daten[run]['max_obs4']
    run_avg = daten[run]['avg_obs4']
    print "%s (min/max/avg): %6g, %6g, %6g" \
          %(name, run_min, run_max, run_avg)
```

- **Hands-On:** Was machen wir genau mit `maxlen`?

- Ausführung als Programm

```
>python SDSS.py
sdss_run2 (min/max/avg): 0.278671, 0.612426, 0.392477
sdss_run3 (min/max/avg): 0.285167, 0.577561, 0.395393
sdss_run1 (min/max/avg): 0.191704, 0.652062, 0.396325
```

- Ausführung als Programm

```
>python SDSS.py
sdss_run2 (min/max/avg): 0.278671, 0.612426, 0.392477
sdss_run3 (min/max/avg): 0.285167, 0.577561, 0.395393
sdss_run1 (min/max/avg): 0.191704, 0.652062, 0.396325
```

- Ausführung als Modul

```
>>> from SDSS import *
>>> daten = init()
>>> zeichneObs4(daten, daten.keys())
```