Einführung in Python
(Programmieren für Naturwissenschaftler)

Tobias Schwabe

Zentrum für Bioinformatik und Physikalische Chemie

Sommersemester 2014

Diese Veranstaltung

Kontakt:

Tobias Schwabe
E-Mail: schwabe@zbh.uni-hamburg.de
Tel.: 42838-7333
ZBH, R. 109

Ein bisschen (hilfreiche ?) Literatur

- https://docs.python.org/3.3/tutorial/index.html
- https://docs.python.org/3.3/library/index.html
- www.google.de
- T. Theis: Einstieg in Python 3 (Galileo Computing)
- B.W. Kernighan, R. Pike: The Practice of Programming (Pearson)
- www.wikipedia.de

Allgemeines zu Skript-Sprachen

- grundlegende Idee: Skriptsprachen sollen die Implementation (kleiner) Programme schnell und einfach gestalten
- da die Ausführung von Skripten nicht zeitkritisch ist (sein sollte), werden sie nicht kompiliert, sondern zur Laufzeit durch einen Interpreter ausgeführt
- der Quellcode (= das Skript) kann also direkt auf jedem Rechner (unabhängig von Architektur und Betriebssystem) ausgeführt werden, solange ein gültiger Interpreter installiert ist
- ursprünglich dienten Skripte vor allem zum Verknüpfen bzw. automatisierten Ausführen von (binären) Programmen (vgl. BASH etc.)
- typische Skriptsprachen: BASH, Perl, Ruby, PHP, Python ...
- viele Skript-Sprachen (wie auch Python) sind aber in der lange, auch komplexere Programme umzusetzen
- Nachteile sind eine geringere Performance gegenüber kompilierten Programmen, was z.B. durch die Sicherstellung einer höheren Stabilität, abstraktere Datenhandhabung oder auch schlechteres Speichermanagement (*Garbage Collection*) verursacht sein kann

Allgemeines zu Python

- Grundlage für diese Vorlesung ist der Python 3.x-Standard
- seit Python 3 sind Skripte nicht mehr abwärtskompatibel zu vorherigen Versionsnummern (2.x etc.)
- als Hinweise vorweg seien kurz die wichtigsten Unterschiede erwähnt:

Python 3

```
print("Text")

3/2 = 1.5
3//2= 1

input()

öväriable = 23
```

bis Python 2

```
print "Text"

3/2 = 1


raw_input()

oevaeriable = 23
```

Erste Schritte in Python: Spielen mit dem Interpreter und Skripte ausführen

- der Aufruf des Python-Interpreters (ohne Argument) öffnet eine interaktive Konsole, die direkt Python-Anweisungen auswertet und ausführt: praktisch, um Kleinigkeiten zu test
- Python 3 im ZBH: `/usr/bin/python3`
- ACHTUNG: standardmäßig (also bei Eingabe von `python`) wird Python 2.7 genutzt
- längere Skripte können in einer einfachen Textdatei gespeichert und durch den Aufruf des Interpreters mit dem Dateipfad als Argument ausgeführt werden
- Konvention: Benennen Sie Python-Skripte mit der Endung `.py`
- Bsp: `python3 mein_skript.py`
- über den Header-Eintrag: `#!/usr/bin/python3` und Änderungen der Dateirechte können Skripte auch ausführbar gemacht werden

Erste Schritte in Python: Anweisungen, Kommentare, Datentypen und Deklarationen (unvollständig!)

Regeln zu Anweisungen in Python

- jede Anweisung steht auf einer Zeile und wird mit einem Zeilenumbruch abgeschlossen, Zeilenfortführung ist aber durch \ möglich:
  ```
  a = b + \
      c
  ```
- zusammengehörende Code-Blöcke **müssen** die gleiche Einrücktiefe aufweisen
- Kommentare werden durch # eingeleitet (muss nicht zu Beginn der Zeile stehen)
- Zuweisungen erfolgen über Gleichheitszeichen (`a = Wert`); nur eine Zuweisung pro Anweisung
- es gibt auch Anweisungen, die keine Zuweisungen sind (z.B. wenn der Wert, der mit einer Variable verknüpft ist, auf dem Bildschirm angezeigt werden soll)
- in Python erfolgt die Variablendeklaration implizit durch den Datentyp, der der Variable zugewiesen wird; somit kann sich der Typ einer Variable zu Laufzeit dynamisch ändern
- Python kennt standardmäßig boolsche Variablen (`True, False`), Integer, Float, Sting (durch ' oder " eingeschlossen) und einige höhere Datentypen (dazu später mehr)

Erste Schritte in Python: Standard-Anweisungen, -Operatoren, -Funktionen

- `+-/*()`: mathematische Operatoren in Verbindung mit Zahlen, Evaluierung nach mathematischer Reihenfolge
- `pow(a,n)`: bildet den Wert von a^n, a,n: bliebige Zahlen
- `'<text>' + '<text>'`: Verknüpfung von zwei Strings
- `and, or`: logische Operatoren
- `print('text',a)`: Ausgabe von Text bzw. der Stringrepräsentation des Datentyps der Variable a; mehrere Argumente werden durch Kommata getrennt
- `len(a)`: bestimmt die Länge eines Sequenzdatentyps (z.B. Liste oder String)
- `range(n)`: erzeugt die Zahlen 0,1,2,...,(n-1), allgemeiner: `range(start, stop, schritt)`, wobei alle Argumente Integer sein müssen (die Bedeutung der Argumente sollte offensichtlich sein)

Ablaufkontrollen: Bedingungen

Ablaufkontrollen im Programm für eine bedingte Ausführung: Verzweigung oder if-Bedingung:

- ```
  if (Bedingung):
      (Anweisungen)
  else: # optional
      (Anweisungen)
  ```
- ```
  if (Bedingung):
      (Anweisungen)
  elif (Bedingung):
      (Anweisungen)
  else:
      (Anweisungen)
  ```
- if-Anweisungen müssen in Python eingerückt werden (z.B. mit `Tab`)
- gleiche Einrücktiefe zeigt, dass alle Anweisungen zu einem Block gehören
- if/elif/else-Bedingungen werden von oben nach unten abgearbeitet und die Anweisungen bei der **ersten** erfüllten Bedingung ausgeführt
- für jedes weitere (verschachtelte) Kontrollelement $\Rightarrow$ zusätzliche Einrückung

Ablaufkontrollen: Bedingungen

Bedingungen ergeben immer eine wahre oder falsche Aussage. Das erfolgt über:

- die Abfrage eines Variableninhalts: `if( var ):`
- einen (Zahlen-)Vergleich: `if( 3 <= var):`
  Es gibt: `<, <=, ==, >=, >`
  Achtung: Ein Variablenzuweisung ist immer wahr. Schreibt man daher `if(var = 3)` statt `if(var == 3)`, ist die Bedingung immer erfüllt. Kehrt man den Test um: `if(3 == var)`, erhält man aber für die Zuweisung `if(3 = var)` eine Fehlermeldung.
- einen Test, ob eine Variable/Teilmenge in einer Menge enthalten ist:
  ```
  string = 'abc'
  char = 'd'
  if( char in string ):
  ```
- Tests/Bedingungen können durch `not` negiert werden:
  `if not(char in string):`
- mehrere Tests können durch `and` oder `or` verknüpft werden

Ablaufkontrollen: Bedingungen (Beispiel)

```
# Seitenlaengen des Rechtecks
a = 2
b = -5

# Gueltigkeitspruefung
if( (a <= 0) or (b <= 0) ):
  print("Die Seitenlängenangaben sind ungültig.")
  print("Keine Berechnung des Flächeninhalts.")
else:
  A = a * b
  print("Der Flächeninhalt beträgt", A)
```

Ablaufkontrollen: Schleifen

Grundsätzlich gibt es zwei verschiedene Schleifentypen in Python:

- die for-Schleife: hier wird eine bestimmte Anzahl von Schleifendurchgängen (die zu Beginn der Schleife bekannt ist) durchgeführt. Dabei gibt es einen Schleifenindex (oder eine Schleifenvariable), die in jedem Durchgang einen neuen Wert annimmt:

```
for idx in 1,2,3,4,5:
    print('Der Index ist', idx)
```

- die while-Schleife: wird solange ausgeführt, wie eine Bedingung wahr ist. Prinzipiell kann sie daher unendlich lange laufen ...

```
a = 9
while (a > 3):
    print('Jetzt hat a den Wert', a)
    a = a - 1
```

- `break`: mit dieser Anweisung wird die aktuelle Schleife beendet, und das Skript nach dem Schleifenende weiter ausgeführt
- `continue`: der Schleifendurchlauf wird beendet und mit dem nächsten Element der Schleife fortgesetzt

Ablaufkontrollen: Schleifen

- Schleifenanweisungen müssen in Python eingerückt werden (z.B. mit `Tab`)
- gleiche Einrücktiefe zeigt, dass alle Anweisungen zu einem Block gehören
- für jedes weitere (verschachtelte) Kontrollelement oder jede weitere Schleife $\Rightarrow$ zusätzliche Einrückung
- um eine bestimmte Anzahl von Schleifendurchläufen eine for-Schleife zu erzeugen, gibt es die Funktion `range( n )` (s. oben):

```
for idx in range( 5 ):
    print('Der Index ist', idx)
```

Achtung: Das Ergebnis ist ein anderes als im vorherigen Beispiel, da die erzeugten Zahlen nun 0,1,2,3,4 sind. Das selbe Ergebnis erzielt man mit:

```
for idx in range( 5 ):
    print('Der Index ist', (idx+1) )
```

Schleifen und Bedingungen

```
# Schleife mit fester Schrittzahl
for i in range(100):
  print( 'Beim Lesen immer in der Reihe beleiben!')

# konditionelle Schleife
counter = 0
while counter < 10:
  print( 'Der Zaehler hat den Wert ', counter )
  # damit sich die Bedingung aendert:
  counter = counter + 1

# Bedingte Ausfuehrung einer Anweisung
for counter in range(10):
  print( 'Der Zaehler mit dem Wert ', counter )
  if (counter % 2) == 0:
    print( ' ... ist gerade' )
  elif (counter % 2) == 1:
    print( ' ... ist ungerade' )
```

Eingabeabfrage

Eingabe (über Standardeingabe):

- Interaktion mit Programm: Benutzer bestimmt Daten durch Eingabe bei Laufzeit (EVA!)
- in Python kann man über die Funktion `input()` eine Eingabe lesen und einer Variable zuweisen:

```
# Aufforderung zu Eingabe
print( ' Bitte die Zahl 23 eingeben und Enter drücken ')
# Eingabe lesen
zahl = input()
if (23 != int(zahl) ):
  print( zahl, ' ist leider knapp daneben!' )
else:
  print( ' Die Antwort ist richtig! ')
```

- ACHTUNG: Die Eingabe wird immer als Typ String behandelt, Umwandlung in Zahlenerfolgt mit:
  - Integer: `int_zahl = int( eingabe )`
  - Kommazahl: `float_zahl = float( eingabe )`
- über `input(' hier tippen-> ')` kann man einen Prompt vor der Eingabe erzeugen

Einstieg in Listen

Listen sind ein sehr praktischer Datentyp:

- Listen sind dynamisch, ihre Länge kann sich während des Programmdurchlaufs ändern (im Unterschied zu Vektoren/Arrays $\Rightarrow$ stets gleiche Länge)
- Listen können unterschiedliche Datentypen enthalten (also z.B. Zahlen und Wörter)
- eine Liste wird erzeugt, in dem ihr Inhalt angegeben und mit eckigen Klammern umschlossen wird; einzelne Elemente werden dabei durch Kommata getrennt:
  `liste = [1, 2, 3]`
  `leere_liste = []`
  Wird einer Variablen eine Liste zugewiesen, erhält sie den Datentyp *Liste*.
- auf ein Listenelement greift man zu, indem man seinen Index angibt:
  `wert = liste[2]`
- ACHTUNG: Index einer Liste läuft von 0 bis (Anzahl der Elemente -1)
- Länge einer Liste bestimmen: `len(liste)` gibt die Anzahl der Elemente der Liste an

mehr zu Listen

- Listen in Python sind *Objekte*, die eigene *Objektmethoden* besitzen
- Liste erweitern: `liste.append( neuer_wert )`
- die Methode wird aufgerufen, indem an den Objektnamen durch einen Punkt der Name der Methode angehängt wird. Eventuelle Optionen werden in Klammern an die Methode übergeben, ansonsten stehen dort leere Klammern
- man kann auch auf Teile von Listen zugreifen: `liste2 = liste1[0:4]` Das Beispiel entspricht: erzeuge eine neue Liste mit den Elementen mit Index 0 bis *ausschließlich* 4 von `liste1` und weise diese der Variablen `liste2` zu
- `liste[:3]` bedeutet vom Anfang der Liste bis ausschließlich Index 3
- `liste[3:]` bedeutet von Index 3 bis zum Ende der Liste

Weitere sequentielle Datentypen: *Tupel*

Ein weiterer sequentieller Datentyp in Python: Tupel

- Idee: eine Zusammenfassung von Werten, die **nicht** durch weitere Zuwweisungen geändert werden können
- Erzeugen: durch Kommata getrennte Werte/Variablen zuweisen:
  `mein_tupel = a, b, c, 1, 2, 3`
  `a, b, c` müssen vorher definiert sein
- Zugriff: über den Index des einzelnen Werts oder durch Entpacken:
  `i, j, k = mein_tuple` (implizite Zuweisungen i=a, j=b, k=c)
- die Zuweisungen der Ergebnisse von Funktionen, die über `return x, y, z` mehrere Werte zurückgeben, ergeben Tupel:
  ```
  x,y,z = find_coordinates( a_pt_in_3D_space )
  ```
  oder
  ```
  pt_coords = find_coordinates( a_pt_in_3D_space )
  ```

Weitere sequentielle Datentypen: *Dictionaries*

Ein weiterer hilfreicher Datentyp in Python: Dictionaries

- Idee: eine Sammlung von Wertepaaren (z.B Name/Telefonnummer, Stadt/Einwohnzahlt, Atomtyp/Gewicht, etc.), generell: Schlüssel/Wert
- Abfragen erfolgen über den ersten Eintrag und liefern den dazugehörigen Werte
- nach dem Konzept: Anfrage = Wie alt ist Peter? Anwort = 36 Jahre.
- die Werte in einem Dictionary lassen sich nur über die Schlüssel addressieren (nicht wie bei Listen, wo man z.B. sich Listenplatz 7 ausgeben lassen kann . . .)
- Dictionary erstellen: `dict = { schlüssel1:wert1 , schlüssel2: wert2, ... }`
- Dictionary erweitern: `dict[schlüssel3] = wert3`
- `dict[schlüssel2]`: gibt wert2 aus
- `dict.keys()`: zeigt alle enthaltenen Schlüssel an
- `del dict[schlüssel1]`: löscht den Eintrag für schlüssel1
- Schlüssel können aus Strings, Zahlen oder Tupeln bestehen

Strings

- String-Objekte – genau wie Listen – gehören zu den sequentiellen Datentypen; sie haben daher viel gemeinsam
- Länge eines Strings: `len(s)`
- Index eines Strings läuft von 0 bis `len(s)-1`
- auf Zeichen im String zugreifen: `ein_zeichen = s[2]`
- Unterschiede existieren auch:
  - String-Objekte können **nicht** direkt verändert werden $\Rightarrow$ `s[0] = 'a'` erzeugt einen Fehler
  - String-Objekte haben eigene (andere) Methoden als Listen

String-Methoden

wichtige String-Methoden:

- Teilstring: `s2 = s1[ 2 : 5 ]`, enthält die Zeichen 2, 3 und 4 von `s1`, also alle Elemente vom ersten angegebenen Index bis zum zweiten Index *ausschließlich*
- `s[:3]` enthält alle Zeichen von Index 0 bis 2
- `s[3:]` enthält alle Zeichen von Index 3 bis Stringende
- `s1 + s2` fügt zwei Strings zusammen
- `s.split([string])`: erzeugt eine Liste von Teilstrings, welche beim Zerschneiden von `s` an den Stellen mit dem Suchstring `string` entstehen. Der Suchstring ist nicht mehr in der Liste enthalten; wird das optionale Argument `string` weggelassen, so wird an Leerzeichen getrennt
- `for zeichen in s`: iteriert über alle Zeichen in `s`, pro Schleifendurchlauf stellt `zeichen` das aktuellen Zeichen in `s` dar.
- `for i in range( len(liste) )`: Schleife, wobei *i* von 0 bis zum Index des letzten Stringzeichen läuft. Aktuelles Zeichen: `s[i]`

String-Methoden

weitere wichtige String-Methoden:

- `s.replace(s_old,s_new)` erzeugt einen neuen String, in dem der Teilstring `s_old` durch den Teilstring `s_new` ersetzt ist
- `s.is[irgendwas]()` testet, ob der String `s` zu einer bestimmten Klasse von Strings gehört bzw. bestimmten Bedingungen genügt (z.B. `isnumeric()`: ergibt `True`, wenn nur Ziffern im String enthalten sind; `islower()`: ergibt `True`, wenn alle Buchstaben im String Kleinbuchstaben sind und min. 1 Buchstabe enthalten ist)
- `s.strip([string])`: erzeugt einen neuen String, in dem am Anfang und Ende der längst mögliche Teilstring mittels der Zeichen in `string` entfernt ist (Auslassungen und Wiederholung von Zeichen sind möglich); wird das optionale Argument `string` nicht angegeben, werden alle Arten von *whitespace* angenommen
- Übersicht: http://docs.python.org/3.3/library/stdtypes.html#string-methods

Datenumwandlung in Strings

Ein wichtige Funktion: `str(var)`

- die Funktion erzeugt aus der Variable `var` die String-Repräsentation des entsprechenden Datentyps, also z.B. eine Ziffernfolge aus Zahlen.
- ist keine Repräsentation bekannt, gibt es einen Fehler

Beispiel:

```
text = '. Zeile'
for i in range(3):
  print( (i+1),text )
```

Ausgabe:

```
1 . Zeile
2 . Zeile
3 . Zeile
```

Beispiel:

```
text = '. Zeile'
for i in range(3):
  neuertext = str(i+1)+text
  print( neuertext )
```

Ausgabe:

```
1. Zeile
2. Zeile
3. Zeile
```

Zwei Worte zu Funktionen

- Funktion, mathematisch: wandelt einen Wert (x) in einen Funktionswert um ($y = f(x)$)
- Funktion, in Programmiersprachen: siehe oben (na ja, fast). Es existieren mathematische Funktionen, etwa:

  `sin(bliebige_variable_die_aber_eine_gültige_Zahl_sein_muss)` ,

  aber auch Funktionen, die allgmeiner sind:

  `print( 'text und ',var )`.

  Die Funktion `print` wandelt `'text'` und die Variable `var` in ein String-Objekt um **und** gibt dieses aus (schreibt es auf den Bildschirm)
- Funktionen können in Programmen beliebig kompliziert werden ...

Funktionen und Unterprogramme

- häufig werden die gleichen Schritte in einem Programm an unterschiedlichen Stellen wiederholt
- praktisch wäre, an diesen Stellen immer auf einen (bestimmten) Teil im Code verweisen, wo die entsprechenden Anweisungen stehen
- möglich durch die Definition von Funktionen und Unterprogrammen
  - Funktion: ein Rückgabewert (-objekt), direkte Zuweisung an eine Variable
  - Unterprogramm: verschiedene Ergebnisse möglich, kein Rückgabewert nötig, kein Zuweisung an Variable
  - in Python nicht streng unterschieden

Definieren von Funktionen und Unterprogrammen

- die Definition einer Funktion muss **vor** ihrem ersten Aufruf erfolgen
- Beispiel einer Funktionsdefinition:
  `def neue_funktion(a, b, c):`
  - `def` leitet die Definition ein
  - `neue_funktion` ist der gewählt Name der Funktion, darüber wird die Funktion in späteren Anweisungen aufgerufen
  - `(a, b, c)` ist die Liste der Variablen (auch Argumente genannt), die an die Funktion übergeben wird; die neue Funktion **muss** immer mit drei Parametern aufgerufen werden
  - `:` leitet einen neuen Block ein; alle folgenden Anweisungen mit gleicher oder größerer Einrücktiefe gehören zu diesem Block

Definieren von Funktionen und Unterprogrammen

- bei späteren Funktionsaufrufen können als Argumente feste Werte oder auch Variablen übergeben werden (ACHTUNG: die Variablennamen im Hauptprogramm und in der Funktion müssen nicht gleich sein):
  `ergebnis = neue_funktion(c, 1.23, a)`
- innerhalb der Funktion gilt dann: `a` wird der Wert der Variable `c` des Hauptprogramms, `b` der Wert 1.23 und `c` der Wert der Variable `a` des Hauptprogramms zugewiesen
- soll eine Ergebnis zurückgegeben werden, erfolgt dies über die Anweisung `return` plus den gewünschten Wert (die Werte):

```
def add1zu( zahl ):
    return (zahl+1)

zwei = add1zu( 1 )
```

- soll die Funktion ohne Ergebnisrückgabe (vorzeitig) verlassen werden, nutzt man `return` ohne Argument

Funktionen und Unterprogramme: Beispiele

```
#!/usr/bin/python

def print_counter(cnt):  # ein Unterprogramm
  print 'Der Zaehler hat den Wert ', cnt

def ist_gerade(a):       # eine Funktion:
  rest = a%2
  if rest == 0:
    return True
  else:
    return False

# Schleife über Unterprogrammaufrufe
for counter in range(10):
  print_counter(counter)

# Funktion nutzen
for counter in range(10):
  print 'Der Zaehler mit dem Wert ', counter
  if ist_gerade(counter):
    print ' ... ist gerade'
  else:
    print ' ... ist ungerade'
```

mehr zu Funktionen und Unterprogrammen

Funktionen, Variablenübergabe, lokale und globale Variablen

- Variablen, die in einer Funktion geändert werden, müssen als Argument übergeben werden
- Variablen, die nicht in der Funktion definiert oder als Argument übergeben werden, sind als *Parameter* bekannt
- einfach Variablen übergeben nur ihren Wert, Objekte (z.B. Listen), werden als Verweis übergeben $\Rightarrow$ ändert sich das Objekt in der Funktion, so ist die Veränderung auch im Hauptprogramm wirksam
- Funktionsaufrufe mit unterschiedlicher Argumentenanzahl möglich, wenn man Defaultwerte nutzt:
  `def func(arg1 = 1, arg2 = 2, arg3 = ' bla '):`
- ...oder beliebige Anzahl an Argumenten übergeben:
  `def func(*args):`

  Die Variable `args` ist eine Liste, die die Argumente enthält:

```
for element in args:
    print( element )
```

Rekursive Funktionen

Rekursive Funktionen:

- Spezialfall für Funktionen: die Funktion ruft sich selbst auf
- sehr hilfreich für einige sehr effiziente Algorithmen
- wird nicht von jeder Programmiersprache untersützt (Python schon)
- in Python: einfach anwenden!

```
def print1biszu( zahl ):
# gibt 1 bis zahl aus
   if zahl < 1:
     return
   else:
     print1biszu( zahl-1 )
     print( zahl )
```

```
def fak( zahl ):
# berechnet Fakultät von zahl
   if 1 >= zahl:
     return 1
   else:
     return zahl*fak( zahl-1 )
```

Arbeiten mit Dateien

- Ein- und Ausgabedatein vereinfachen die Arbeit mit Programmen
- Eingabedateien nützlich, wenn z.B. Datensätze von anderen Quellen genutzt werden sollen
- Ausgabedateien nützlich, wenn Informationen an unterschiedelichen Orten abgelegt werden sollen bzw. langfristig archiviert werden sollen

Arbeiten mit Dateien: Lesen

- zum Lesen erzeugt man ein Dateiobjekt mit der Funktion: `datei = open('dateinname')`
- der Dateiname bezieht sich dabei auf eine Datei, die in dem selben Verzeichnis wie das Pythonskript liegt (bzw.: in dem selben Verzeichnis, in dem das Skript aufgerufen wurde)
- der Dateiname kann auch durch eine Variable angegeben werden, die einen entsprechenden String enthält
- mittels Dateiobjekt kann auf den Inhalt der Datei zugegriffen werden
- hierzu stellt das Dateiobjekt verschiedene Methoden zur Verfügung
- wird ein Dateiobjekt nicht mehr benötigt, schließt man es über die entsprechende Methode `datei.close()`

Arbeiten mit Dateien: Lesen

```
#!/usr/bin/python

# oeffnen einer Datei
datei_in = open('file01.txt')

# Datein zeilenweise lesen
for zeile in datei_in:
  print( zeile, end = '')

print '=== keine weiteren Zeilen ==='

file_in.close()
```

Arbeiten mit Dateien: Lesen

weitere Möglichkeiten des Lesens:

- `inhalt = file_in.read()` $\Rightarrow$ liest die gesamte Datei ein
- `listezeilen = file_in.readlines()` $\Rightarrow$ erzeugt eine Liste mit alle Zeilen in der Datei

Arbeiten mit Dateien: Schreiben

- genau wie das Einlesen einer Datei ist das Schreiben ebenso wichtig
- Vorgehen ist ähnlich wie beim Lesen einer Datei
- damit man in eine Datei schreiben darf, muss sie mit der entsprechenden Option geöffnet werden: `open('dateiname', <opt>)`, wobei `<opt>` entweder `'w'` für (Über-)Schreiben oder `'a'` für Anhängen ist (Kleinschreibung beachten)
- die Option kann auch in einer Variablen angegeben werden
- wird keine Option angegeben, wird die Datein nur mit Leserechten geöffnet, Schreiben führt zu einem Fehler
- im Prinzip wird die `print`-Ausgabe nur durch eine entsprechende Methode des Dateiobjekts ersetzt: `datei.write( string )`
- mit `write()` können **nur** Strings geschrieben werden, während `print()` Variablen selbst umformt, man muss also ggf. mit `str()` arbeiten

Arbeiten mit Dateien: Schreiben

```
#!/usr/bin/python

# öffnen einer Datei mit Option w
file_out = open('file01.out', 'w')

# Datei schreiben
for i in range(5):
  line = ' eine neue Zeile mit der Nummer ' + str(i+1) + '\n'
  file_out.write(line)

file_out.write( '=== keine weiteren Zeilen ===')

file_out.close()
```

Arbeiten mit Dateien: Schreiben

Besonderheiten: das weiße Nichts (whitespace)

- es gibt bestimmte Zeichen, die man im gedruckten Text nicht (unbedingt) sieht, aber dennoch Teil eines Strings sein können: Leerzeichen, Tabulatoren, Zeilenumbrüche
- sind notwendig zur Textformatierung
- müssen entsprechend explizit auch angegeben werden
- in Python: `' '` (Leerzeichen), `'\t'` (Tab), `'\n'` (Zeilenumbruch)
- das `'\'`-Zeichen ist ein Formatierungssymbol, daher kann es nicht direkt in einem String stehen, sondern muss selbst als Sonderzeichen angegeben werden: `\\`

Formatiertes Schreiben

Wir kennen bereits:

- allgemeine Ausgabe mit `print( <string> )`

. . . und heute kommt hinzu:

- schön schreiben (= formatierte Ausgabe, also Einhalten von Spalten, feste Breite von Zahlen etc.)

Wo ist das Problem?

- Wenn wir eine schöne Ausgabe wollen, müssen wir uns eben mit den Strings, die wir erzeugen, Mühe geben . . .
- unschön wird es, wenn wir Variablen in die Strings einbauen, da oft vorher deren Länge nicht bekannt ist (Wie viele Nachkommastellen hat eine Zahl? Etc.)

Formatiertes Schreiben

Python bietet auch hier (natürlich) eine Lösung an:

- Jeder String hat die Methode `<string>.format( <arg> )` .
- Im String werden Platzhalter makiert: `'Hier ist Platz für den Inhalt einer Variable: {}'`
- Die Methode übernimmt ein Argument oder eine Liste von Argumenten und fügt sie der Reihe nach an Stellen der Platzhalter ein. Dabei werden Objekte, die keine Strings sind, in solche umgewandelt

Formatiertes Schreiben

Beispiele:

- `'Die erste Variable enthält {}, die zweite {}'.format( 0,'nichts' )`
  erzeugt den String:
  `'Die erste Variable enthält 0, die zweite nichts'`
- Man kann auch die Positionen aus der Liste der format()-Argumente spezifizieren:
  `'Die erste Variable enthält {1}, die zweite {0}'.format( 0,'nichts' )`
  erzeugt den String:
  `'Die erste Variable enthält nichts, die zweite 0'`
- ist eins der format()-Argumente ein Sequenz-Objekt (Liste), so können einzelne Elemente referenziert werden:
  ```
  liste = [ 0, 'nichts' ]
  'Die erste Variable enthält {0[1]}, die zweite
  {0[0]}'.format( liste )
  ```
  erzeugt ebenfalls den String:
  `'Die erste Variable enthält nichts, die zweite 0'`

Formatiertes Schreiben

Und wo ist jetzt der Vorteil?

- alle vorherigen Beispiele auch möglich durch die Umwandlung von Variablen mit `str()` und das Zusammenfügen von Strings: `<string1> + <string2>`
- aber: die Platzhalter können mit Formatangaben versehen werden:
  `'Die erste Variable enthält {:4.2f}, die zweite {:.>10}'.format( 0,'nichts' )`
  erzeugt den String:
  `'Die erste Variable enthält 0.00, die zweite ....nichts'`
- die Formatangaben werden durch `:` eingeleitet und enthalten verschiedene Informationen; referenziert der Platzhalter eine bestimmte Position der Argumente, steht diese vor den Formatangaben: `{2:4.3f}`

Formatangaben

- die Formatangaben erlauben eine recht hohe Flexibilität, hier konzentrieren wir uns auf das Wesentliche
- String-Variablen einfügen: [[Füllzeichen]Positionierung][Breite]
  - Füllzeichen: ein alternatives Zeichen, mit dem anstatt von Leerzeichen der Platz aufgefüllt wird (optional, erfordert Positionierung)
  - Positionierung: Zeichen, welches die Position angibt ('<', rechtsbündig, '>' linksbündig, '^' zentriert; optional)
  - Breite: Zahl, welche die Anzahl reservierter Zeichen angibt (optional)
- Zahl-Varibale: [[Füllzeichen]Positionierung][Breite][.Nachkommastellen][Typ]
  - Füllzeichen, Positionierung, Breite: siehe oben
  - Nachkommastellen: Zahl, welche die Anzahl von Nachkommastellen angibt (optional, Standard: 6)
  - Typ: Darstellungsform der Zahl: 'd' – (Dezimal-)Integer, 'f' – Fließkommazahl, 'e' – Exponential- oder wissenschaftliche Notierung, 'g' – generelles Format (Python entscheidet über beste Darstellung) (optional)

(http://docs.python.org/3.3/library/string.html#format-specification-mini-language)

Formatangaben

Beispiele:

- `zahl = 42.07289`
  `' Die Anwort ist {:5.0f}'.format( zahl )` ergibt:
  `' Die Anwort ist    42'`
- `' Die Anwort ist{:.>7.1f}'.format( zahl )` ergibt:
  `' Die Anwort ist...42.1'`
- `wort = 'bla'`
  `' Hier steht nur{:5}{:3}'.format( wort, wort )` ergibt:
  `' Hier steht nur  blabla'`

Prinzipiell sind die Platzhalter und Formatangaben nur normale String-Zeichen. Man kann entsprechende Strings daher z.B. zu Laufzeit des Skripts erst konstruieren und dann die format()-Methode auf die erzeugten Strings anwenden ...

Einschub: Zeichenkodierung

- Datein bestehen nur aus Bits (0/1)
- Zeichen werden aus diesen 0en und 1en generiert
- die Regeln dazu nennt man Kodierung; sie besagen, wie viele Bits ein Zeichen repräsentieren und was die Zahl, die sich aus den Bits ergibt, bedeutet
- das hat zur Folge, dass ein und dieselbe Datei unterschiedlich wiedergegeben werden kann ...
- die Standardkodierung ist das ASCII-Format, sie enthält das Englische Alphabet und einige Satz- und Sonderzeichen
- ASCII kennt z.B. keine Umlaute

Einschub: Zeichenkodierung

- um nun die Vielfalt der Schriftzeichen aller Kulturen abzubilden, wurde UNICODE entwickelt
- Unicode enthält alle bekannten Zeichen (bzw. wird laufend erweitert)
- UTF-8 ist ein Standard, wie die Unicodezeichen in Bit-Code übersetzt werden könnnen (und umgekehrt)
- da es so viele Unicodezeichen gibt, muss die Zeichenrepräsentation mit mehr Bits erfolgen als im ASCII-Format
- ASCII und UTF-8 sind daher nicht kompatibel

Einschub: Zeichenkodierung

- Python kann mit beiden Kodierungen (und weiteren) umgehen, aber man muss diese explizit angeben
- Unicode-Strings können entsprechend markiert:
  `string=u'unicode zeichen'`
- ein vorangestelltes
  `#-*- coding: utf-8 -*-`
  zu Beginn eines Skripts erlaubt direkt das Verwenden einer Kodierung (hier: Unicodezeichen) in dem Skript
- bei der Ausgabe von Text muss auch wieder angegeben werden, für welches Zeichen der Bitcode eigentlich steht. Dies geschieht mit:
  `string.encode(Coding)`
  wobei Coding ein String mit der Kodierungsbezeichnung ist

Einschub: Zeichenkodierung

```
#-*- coding: utf-8 -*-
txt=u'Hällo Wörld'
print( txt )
for codec in 'iso-8859-15', 'euc_jp', 'utf-8':
    # generate byte representation
    print( txt.encode(codec))
    # re-encode to UTF-8
    print( txt.encode(codec).decode(encoding='utf-8', \
           errors='replace') )
```

Python: *Batteries Included*

- Python bietet viel Lösungen für Aufgaben, die nicht jeder Programmierer ständig braucht
- um die Sprache übersichtlich und effizient zu halten, sind diese Funktionen und Methoden in Module gepackt, die nur bei Bedarf angefordert werden
- diese Module bilden die Standard-Biliothek (*Standard Library*) und sind der Grund für die Aussage:

  Python comes with batteries included

- die Verwendung der Module ist recht einfach – das Problem liegt eher darin, sich einen Überblick zu verschaffen ...

Module verwenden

- um ein Module zu laden, nutzt man folgende Anweisung:
  `import <Modulname>`
- das Laden kann jederzeit im Skript erfolgen (jedoch muss dies geschehen, bevor man das Modul benötigt), erfolgt aber normaler Weise zu Beginn des Skripts
- auf ein Objekt bzw. eine Funktion des Moduls greift man dann wie folgt zu:
  `<Modulname>.<Name_Modulfunktion>()`
- . . . also analog zu den Methoden eines Objekts

Module verwenden

möchte man nicht so viel Tippen, gibt es einige Alternativen:

- man kann einen Verweis auf die Funktion definieren:
  `lokale_funktion = <Modulname>.<Modul-Funktion>`
  Achtung: Nur den Name, keine Klammern `()` nutzen, sonst wird das Ergebnis des Funktionsaufrufs der neuen Variablen zugewiesen
- `from <Modulname> import <Modul-Funktion>` (Achtung: keine Klammern), dann kann die Funktion direkt benutzt werden
- `import <Modulname> as <Kurzname>`; `<Kurzname>` verweist nun auf das gewünschte Modul, Funktionen lassen sich über `<Kurzname>.<Methode>()` nutzen.
- Sonderfall: `from <Modulname> import` $*$, was das Laden aller Funktionen des Moduls erzeugt; dies kann allerdings unerwünschte Nebeneffekte haben, wenn dadurch zuvor definierte Verweise bzw. Funktionsnamen überschrieben werden ...

Module verwenden

```
import sys, random

lowercase = 'abcdefghijklmnopqrstuvwxyz'
uppercase = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
letters     = lowercase + uppercase

# only the exit and uniform functions are of interest
exit = sys.exit
uf   = random.uniform

letter = letters[int( uf(0,51.999999) )]

while True:
  print( ' > Guess the letter to escape ')
  tryletter = input('   ')
  print( ' > Your choice:',tryletter )
  if letter == tryletter:
    exit( '   Congratulations!')
  else:
    print( '   Almost... but no.')
```

Eigene Module verwenden

Python erlaubt auch die Definition von eigenen Modulen:

- `import <Modulname>` lässt den Python-Interpreter nach einer Datei `<Modulname>.py` in bestimmten Standardverzeichnissen (-pfaden) suchen; das aktuelle Verzeichnis gehört dazu
- legt man eine entsprechende Datei im selben Verzeichnis wie das aktuelle Skript an, kann man so sein eigenes Modul definieren
- der Zugriff und die Benutzung ist dann völlig analog zu den Standardmodulen

meinmodul.py:

```
def saghallo(name):
  print( 'Hallo', name + '!')
```

meinskript.py:

```
import meinmodul
nutzername = input('Wie heißt Du?\n > ')
meinmodul.saghallo(nutzername)
```

Scope von Variablen

- der innerste Scope, welcher zuerst durchsucht wird, enthält *lokale* (`local`) Namen
- der Scope von umschließenden Funktionen, welche ausgehend vom nächsten einschließenden Scope aus durchsucht werden, enthält *nicht-lokale* (`nonlocal`) aber auch *globale* (`global`) Namen
- der vorletzte durchsuchte Scope enthält die globalen Namen des gegenwärtigen Moduls (einschließlich des Moduls `__main__`)
- der äußerste Scope ist der der `built-in` Namen
- `global` kann genutzt werden, um bestimmte Variablen in den globalen Scope zu verschieben
- `nonlocal` kann genutzt werden, um bestimmte Variabeln in den einschließenden Scope zu verschieben

Scope von Variablen

Typische Anwendung:

```
def scope_test():
    def do_local():
        spam = "local spam"
    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"
    def do_global():
        global spam
        spam = "global spam"
    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

Fehler, Ausnahmen, Ausnahmefehler

Grundsätzlich gibt es zwei Arten von Fehlern:

- Syntax-Fehler: diese gehen zurück auf ungültigen Code und sind allein durch den Programmierer verschuldet; sie gilt es auszumerzen, bevor ein Skript/Programm als korrekt bezeichnet werden darf
- Laufzeit-Fehler: hier funktioniert der Code im Prinzip, wenn im Skript alles wie vorhergesehen abläuft, durch (falsche) Interaktion, Systembedingungen etc. können dennoch Probleme auftreten; solche Probleme möchte man **abfangen**, um evtl. nachzusteuern oder wenigstens das Programm korrekt zu beenden

Versuch und Irrtum

- zum Abfangen von Fehlern gibt es die `try ...except` Anweisung; sie schließt kritischen Code in einem eigenen Block (Einrückung(!) ein; falls hier ein Fehler auftritt, wird das Programm nicht sofort beendet:

```
try:
    <some code>
except:
    pass
```

- Der Ausnahmfall, falls ein Fehler auftritt, muss angegeben werden. Die Anweisung `pass` führt dazu, dass das Skript normal weiter ausgeführt wird.

Versuch und Irrtum: Fehlertypen

- zwar kann man auftretende Fehler über `pass` einfach ignorieren, besser wäre aber, auf den Fehler durch weitere Anweisungen zu reagieren
- damit dies besser möglich ist, kennt Python mehrer Standardfehler, z.B.: `ZeroDivisionError, IOError, TypeError`

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except IOError as err:
    print("I/O error: {0}".format(err))
except ValueError:
    print("Could not convert data to an integer.")
except:
    print("Unexpected error:", sys.exc_info()[0])
    raise
```

Versuch und Irrtum: `raise`

- `raise` im vorherigen Beispiel führt dazu, dass der gerade abgefangene Fehler erneut geworfen wird (weil übergeordneter Code damit umgehen kann oder damit der Benutzer den Fehlertyp sehen kann)
- `raise` kann auch genutzt werden, um gezielt einen vordefinierten Fehler zu werfen:

```
import sys
name='tobias'
try:
  name2 = name
  if name2 == 'tobias':
    raise NameError
except NameError:
  sys.stderr.write(name + ' ist einmalig. Keine Duplikate!\n')
```

- über `sys.stderr.write()` kann die Fehlermeldung auf den Standard-Fehleroutput umgelenkt werden.

Versuch und Irrtum: weitere Verzweigung

- `try ...except ...else` Anweisung: Code im Zweig `else` wird dann ausgeführt, wenn die Anweisungen in `try` ohne Fehler ausgeführt wurden; sinnvoll, wenn die Fehlerbehandlung sich nur auf einen Teil des Zweigs `try` bezieht, die Anweisungen in `else` aber eben nur bei Erfolg ausgeführt werden sollen
- `try ...except ...finally` Anweisung: Code im Zweig `finally` wird immer ausgeführt, unabhängig davon, ob ein Fehler aufgetreten ist (*Clean-Up-Action*)
- `finally` folgt auf `else`, falls beides verwendet wird

```
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        sys.stderr.write("division by zero!\n")
    else:
        print("result is", result)
    finally:
        print("executing finally clause")
```

Dateien öffnen mit `with`

- um das Arbeiten mit Datein und häufige Konstrukte von `try ...finally` zu vermeiden, sollte man das `with`-Statement nutzen
- dieses sorgt dafür, dass ein Dateizugriff immer korrekt beendet und die Datei geschlossen wird, auch wenn **nicht** explizit `<datei>.close()` verwendet wurde, weil vielleicht ein Fehler aufgetreten ist

```
with open('foo.tmp','w') as f:
  f.write('')
try:
    with open('foo.tmp', 'r') as f:
        f.write('bar')  # should raise error: read only
except:
    pass
print( f.closed  )  # evaluates to True
# bar.tmp is missing, so g is never assigned to data object
try:
    with open('bar.tmp', 'r') as g:
        g.write('foo')
except:
    pass
print( g.closed )   # raises error: g is undefined
```

Objekt-orientierte Programmierung: Einführung

- objekt-orientierte Programmierung (OOP): (neueres) Entwicklungskonzept für Programme (im Gegensatz zu prozess-orientierter Programmierung)
- beim prozess-orientierten Ansatz steht der eigentliche Ablauf des Programms und dessen Resultat im Mittelpunkt, also Daten/Eingabe rein $\Rightarrow$ Rechnen $\Rightarrow$ Ergebnis erhalten
- beim objekt-orientierten Ansatz: die einzelnen Elemente eines Programms stehen im Mittelpunkt; sie geben vor, was mit ihnen gemacht werden kann und wie sie mit einander interagieren
- OOP vor allem eine Antwort für die Entwicklung von Programmen mit viel Nutzer-Interaktion, da hier Prozesse oft nicht strikt vorgegeben

Objekt-orientierte Programmierung: Einführung

- weitere Vorteile:
  - erleichtert Entwicklung großer Programmierprojekte (z.B. können einmal entwickelte Objekte in verschiedenen Programmen eingesetzt werden)
  - mehr Sicherheit/stabilerer Code
  - mehr Felxibilität
  - ...
- Nachteile:
  - Verwaltung von Objekte führt zu Mehraufwand in der Rechenzeit
  - Entwicklungsvorteil oft nur für große Vorhaben/langfristig

Objekt-orientierte Programmierung: Vokabular (Python)

- sich gleichende Gegenstände/Programmbestandteile gehören zu einer *Klasse* (etwa ein VW Polo und ein Opel Astra gehören beide zur Klasse PKW), die Gegenstände selbst sind eine *Instanz* der *Klasse* und werden *Objekt* genannt
- *Klassen* definieren *Eigenschaften* und *Methoden*, die allen *Objekten* gemein sind
- *Objekteigenschaften*: Werte, die ein *Objekt* charakterisieren
- *Objektmethoden*: Funktionen, die direkt das *Objekt* betreffen (z.B. *Eigenschaften* verändert) bzw. ein Ergebnis erzeugen, das vom *Objekt* bzw. seinen *Eigenschaften* abhängt
- z.B. könnte alle *Objekte* der Klasse PKW die *Eigenschaften* Motorleistung, Anzahl Türen, Neuwert, ... und die *Methoden* lackieren, Tür öffnen, beschleunigen, ... haben

OOP in Python

- Python ist eine (fast) durchgehend objekt-orientierte Sprache (alle Datentypen in Python sind z.B. *built-in* Klassen; die Zahl 4 ist daher ein *Objekt* der *Klasse* Integer usw.)
- in Python können auch neue *Klassen* definiert werden:

```
class eine_Klasse:
  name = ''
  wert = 23
```

- ein *Objekt* dieser *Klasse* wird wie folgt erzeugt: `obj = neue_Klasse()`
- auf eine *Eigenschaft* zugreifen: `print( 'Das Objekt hat den Wert: ', obj.wert )`
- *Eigenschaft* ändern: `obj.name = 'Objekt_1'`
- neues *Objekt*: `obj_2 = neue_Klasse()`
  `obj_2.name = 'Objekt_2'`

OOP in Python

- bei der Definition einer *Klasse* müssen den *Eigenschaften* stets Anfangswerte zugewiesen werden, um sie zu definieren
- die Behandlung von *Klassenmethoden* erfolgt nächste Stunde
- erzeugt man ein *Objekt* und verweist dieses *Objekt* einer neuen Variable zu, so sind diese beiden *Instanzen* (normaler Weise) komplett identisch

```
obj1 = neue_Klasse()
obj2 = obj1
obj1.wert = 42
if (obj2.wert == obj1.wert):
  print (' Die Werte sind gleich ')
```

- in obigem Beispiel erfolgt die Ausgabe, da `obj1` und `obj2` auf die **selbe** *Instanz* verweisen

Objekte und Methoden

- wie bereites erwähnt, können Objekte *Eigenschaften* und *Methoden* besitzen
- Methoden sind Funktionen, die zu einem Objekt gehören (und deren Ergebniss normaler Weise vom Objekt abhängen)
- die Definition erfolgt analog wie bei normalen Funktionen, jedoch innerhalb der Klassendefinition:

```
class eine_Person:
  name = ''
  alter = 23
  def geburtstag_feiern(self):
    self.alter = self.alter + 1

Mark=eine_Person()
Mark.name = 'Mark'
Mark.alter = 23
Mark.geburtstag_feiern()
print( Mark.alter )

>>> 24
```

- Besonderheit: selbst eine Funktion ohne Argument muss immer mit einem Verweis auf das Objekt selbst (`self`) definiert werden. Weitere Argumente folgen darauf wie gewohnt

Objekte und spezielle Methoden

- Objekte kenne einige ganz spezielle Methoden, die bestimmte Aufgaben erfüllen
- `__init__`: Methode zur Initialisierung eines Objekts

```
class eine_Person:
  name = ''
  alter = 0
  def __init__(self, Name, Alter):
    self.name  = Name
    self.alter = Alter

Mark=eine_Person('Mark', 23)
print( Mark.alter )

>>> 23
```

- die Initialisierungsmethode wird **nur** beim Erzeugen eines Objekts aufgerufen
- dadurch werden erste Eigenschaften definiert
- die Funktion darf **keinen** Rückgabewert haben (keine `return` Angabe)

Objekte und spezielle Methoden

- `__str__`: Methode zur Ausgabe eines Objekts

```
class eine_Person:
  name = ''
  alter = 0
  def __init__(self, Name, Alter):
    self.name  = Name
    self.alter = Alter
  def __str__(self):
    return ( self.name + ' (' + str(self.alter) + ')' )

Mark=eine_Person('Mark', 23)
print( Mark )

>>> Mark (23)
```

- die Methode muss einen String zurückgeben

Klassen und Vererbung

- bei der Definition von Klassen kann man sich das Leben leichter machen durch *Vererbung*
- die Idee: es gibt eine gemeinsame Oberklasse (Fahrzeug) und Unterklassen (PKW, SKW, LKW, Baufahrzeug). Alle gemeinsamen Eigenschaft und Methoden werden in der Oberklasse definiert, Spezialitäten in der jeweiligen Unterklassen
- die vererbende Klasse wird bei der Definition angegeben:

```
class LKW(Fahrzeug):
  ...
```

- bei der Definition der Unterklasse können Eigenschaften und Methoden neu belegt werden, ansonsten hat die Unterklasse implizit alle Eigenschaften und Methoden der Oberklasse
- dieser Ansatz kann viel Entwicklungs- und Programmierarbeit einsparen

Private Variablen in Objekten

- private Variablen (oder auch Methoden) werden nicht direkt von Python unterstützt, es gibt aber eine Konvention, dass Eigenschaften, die durch `__<varname>` definiert werden, privat behandelt werden.
- dies wird durch ein *name mangling* unterstützt, das diese Variablen durch `_classname__varname` ersetzt. Darüber sind sie aber immer noch erreichbar.

Zwei Mottos der Python-Community:

> *You don't enter the living room because you're not invited to—not because you couldn't.*
>
> *Python makes several assumptions. One is: You're not an idiot.*

```
class newob():
  nonprivvar= 42
  __privvar = 23
o = newob()
print( o.nonprivvar )
try:
  print( o.__privvar )
except:
  print( ' this is privat! ')
  pass
print( o._newob__privvar )
```

Das Modul `matplotlib`

(siehe: http://matplotlib.org/contents.html)

- erlaubt das Plotten von Graphen analog zu MATLAB
- ACHTUNG: einfacher in Skripten zu verwenden, da der Interpreter nicht (unbedingt) mit dem Backend (erzeugt die Ausgabe als Bild) kommuniziert
- wichtigstes Modul: `matplotlib.pyplot`, da hiermit die Graphen erstellt werden

Das Modul `matplotlib.pyplot`

Beispiel

```
import matplotlib.pyplot as plt

l1 = [0.1,0.2,0.3,0.4]
l2 = [0.01,0.04,0.09,0.16]
l3 = [0.02,0.08,0.18,0.32]

plt.plot(l1,l2,'r-o',l1,l3,'b-v', linewidth=3.0)
plt.title('A New Plot')
plt.grid(True)

plt.savefig('newplot.pdf')

# pdf is broken if show() is used first
plt.show()
```

Mehr zu Listen und Iteratoren

- *list comprehensions*: das Erzeugen von Listen durch Anweisungen wie:
  `newlist = [ pow(x,0.5) for x in xlist]`
- *generator expressions*: dabei wird nur bei Bedarf das nächste Objekt der Anweisung erzeugt ohne erst die ganze Liste zu erstellen (vgl. `range()`):
  `S = sum( ( pow(x,0.5) for x in xlist) )`
  (schont den Speicherbedarf bei langen Listen)
- `iter(seq)`: erzeugt ein Iteratorobjekt, wenn der Sequenzdatentyp von `seq` dies unterstützt, also z.B. bei Listen, umgekehrt erzeugt `list(it)` eine Liste mit allen Elementden der Iterators `it`
- Objekte können eine eigene Iteratormethode (`__iter__()`) zur Verfügung stellen:

```
class einob():
 def __init__(self):
    self.l1 = list(range(5))
 def __iter__(self):
    l2=[x*x for x in self.l1]
    return iter(l2)
o = einob()
for x in o:
  print( x, end=' ')

>>> 0 1 4 9 16
```

```
class einob():
  def __init__(self):
    self.l1 = list(range(5))
  def __iter__(self):
    l2=(x*x for x in self.l1)
    return l2
o = einob()
for x in o:
    print( x, end=' ')

>>> 0 1 4 9 16
```

Funktionale Programmierung

Die Umsetzung funktionaler Programmierung beruht im wesentlich auf den folgenden Funktionen:

- `lambda` Ausdrücke zur Definition anonymer Funktionen:

```
>>> lf = lambda x, y: x+y
>>> lf(20,3)
23
```

- `zip()`: das Zusammenführen von Listen
  `for x,y in zip(list1,list2):`
- `map(f,l)`: erzeugt einen Iterator mit der Funktion `f` für alle Werte in `l`
  `l_square = list( map(lambda x: x*x, l_x) )`
- `filter(l_logfilter,l)`: erzeugt einen Itertor mit den Elementen aus der Liste `l`, für die der Eintrag in der Liste `l_logfilter` wahr ist
  `>>> list( filter(lambda x: x%2==0, list(range(10))) )`
  `[0, 2, 4, 6, 8]`

Das Parsen von Argumenten

- häufig sollen einem (Python-)Skript bei Aufruf ein oder mehrere Kommandozeilenargumente übergeben werden (z.B. der Name einer Inputdatei)
- Python stellt zwei (komplementäre) Möglichkeiten zur Verfügung
- zunächst die direkte Version:

```
import sys
for i in sys.argv[1:]:
  try:
    finp = open( i )
    print( finp.read() )
 except:
   # extra ( ) to break long line
   sys.stderr.write((' Das ' + str(sys.argv.index(i)) +
     '. Argument kann nicht ge"offnet/gelesen werden'))
```

Die Liste `sys.argv`

- die Liste `sys.argv` enthält alle Kommandozeilenparamter als Strings
- `sys.argv[0]` enthält den Namen des Skripts
- die restliche Liste besteht aus weiteren Argumenten beim Aufruf, die Trennung erfolgt an den Leerstellen zwischen den Argumenten
- dies ist sehr nützlich, um eine (oder mehrere) Inputdateien an das Skript zu übergeben
- im Prinzip kann man darüber auch einen Parser für Optionen entwickeln – allerdings wurde diese Arbeit schon erledigt

Das Modul `argparse`

- mit dem Modul kann Objekt der Klasse `argparse.ArgumentParser` erstellt werden, welche das Handling der Argumente übernimmt
- damit kann der POSIX-Standard (aber auch flexiblere Handhabungen) umgesetzt werden (also: `-x` für Optionsnamen mit nur einem Zeichen, `-xyz`: Verknüpfung der Optionen x,y und z, `--xyz` für eine Option mit Namen `xyz`)
- darüber Hinaus können auch Argumente ohne Optionsbezeichnung behandelt werden

Das Modul argparse

```
import argparse
parser = argparse.ArgumentParser(prog='MyProg',
  usage='%(prog)s [options] <list of input files>')
parser.add_argument('-q','--quiet',default=False,
  action="store_true"
  help='turns quiet mode on ==> less output on screen')
parser.add_argument('-r','--repeats', nargs=1, type=int,
  action="store", help='no. of repeats in output')
parser.add_argument('inpfiles',nargs="*", action="store",
  help='list of input files (blank separated) to process')
args = parser.parse_args()
if not args.quiet:
  for file in args.inpfiles:
    print( args.repeats*file )
```

Das Modul `argparse`

das Wichtigste zu `.add_argument()`

- `name` oder `flag`: String, der die Option bezeichnet, mit `-` oder `--` werden Flags eingeleitet (sind optional, können vor oder hinter den festen Optionen erscheien)
- `help`: gibt String für die Ausgabe von `.print_help()` an
- `type`: gibt den Datentyp des zu speichernden Arguments an
- `action`: Default `store` speichert Argument unter `name` bzw. `flag`; `store_true` (`_false`) setzt `True` or `False` für das Arguemnt
- `nargs=[n]`: setzt eine Anzahl von Argumenten für die Option fest, `n = *` erlaubt beliebig viele Argumente, erzeugt eine Liste
- mehr unter:
  - https://docs.python.org/3.3/howto/argparse.html
  - https://docs.python.org/3.5/library/argparse.html

Das Modul `numpy` und seine Arrays

NumPy ist nicht Teil der Standardbibliothek und muss ggf. nachinstalliert werden
Siehe auch:

- http://www.numpy.org/
- http://www.python-kurs.eu/numpy.php
- http://wiki.scipy.org/Tentative_NumPy_Tutorial

- NumPy stellt viele Objekte und Methoden zur Verfügung, die auch im `math`-Modul enthalten sind: `.sin()`, `.Pi`, `.acos()` etc., aber auch viele darüber hinaus
- oft sind die NumPy-Methoden effizienter, z.B:
  `von0bis99999 = numpy.arange(100000)` vs.
  `von0bis99999 = list( range(100000) )`

`numpy` und seine Arrays

Arrays:

- NumPy erlaubt das Arbeiten mit Arrays (und Matrizen, s. unten)
- Arrays sind sequentielle Datentypen (Felder) des gleichen Elementardatentyps (normaler Weise int oder float) und könnenbeliebig-dimensional sein
- Benutzung: `feld = numpy.array([[11,12,13],[21,22,23]], int)` ergibt ein 2x3 Matrix
- Rang eines des Arrays: `print( feld.ndim ) >>> 2`
- Form des Arrays: `print( feld.shape ) >>> (2, 3)`
- Gesamtzahl der Elemente: `print( feld.size ) >>> 6`
- Zugriff auf einzelne Elemente: `print(feld(0,3)) >>> 13`
- iterieren über Arrays:

```
for zeile in feld:
  for wert in zeile.flatten:
    print( wert, end=',')
  print()
```

```
for zeile in feld:
  for i in range(len(zeile)):
    print( zeile[i], end=',')
  print()
```

`numpy` und seine Arrays

Rechnen mit Arrays:

- mathematische Operatoren `+-*/**%` werden unterstützt, wobei sie elementweise angewendet werden
- das Skalar- und das Kreuzprodukt kann berechnet werden:

```
import numpy
A = numpy.arange(10)
x = numpy.dot(A,A)
y = numpy.cross(A,A)
```

Das Modul `numpy`

- `numpy` kennt auch Matrizen (die immer 2-D sind), und viele Eigenschaften von Arrays erben, es gibt aber auch ein paar Unterschiede ...
- die Matrixmultiplikation wird direkt unterstützt:
  ```
  A = numpy.matrix([[1,-1],[-1,1])
  B = A*A
  ```
- Matrix-mal-Vektor wird aber nicht direkt umgesetzt, hier muss man das Skalarprodukt aufrufen ...
- Slicing: nur bestimme Zeilen/Spalten adressieren
  `B[:,1]` erzeugt eine Nx1 Matrix mit allen Elementen der 2. Spalte (Shape = (N 1))
  `B[0,:]` erzeugt eine 1xN Matrix mit allen Elementden der 1. Zeile (Shape = (1 N))
- wie bei allen sequenziellen Objekten, werden diese nicht direkt kopiert, sondern nur weitere Verweise auf das Objekt angelegt. Um dies zu umgehen gibt es:
  ```
  C = B.copy()
  ```

Das Submodul `numpy.linalg`

- NumPy stellt auch das Submodul `linalg` für simple(?) lineare Algebra zur Verfügung
- damit lassen sich z.B. die Inversen einer Matrix oder auch die Eigenwerte und Eigenvektoren einer Matrix bestimmen:

```
import numpy as np
import numpy.linalg as lina
A = np.matrix([[1,0,0],[0,2,0],[0,0,3]])
print( lina.inv(A) )
eigenwrt, eigenvek = lina.eig(A)
for w, vek in zip(eigenwrt, eigenvek):
  print(w, vek)
```

Das Modul `ctypes`

Python stellt ein Modul (`ctypes`) zur Verfügung, mit dessen Hilfe C-Funktion aus einem Pythonskript aufgerufen werden können.

Voraussetzung:

- die Funktion ist ein einer dynamisch gelinkten Bibliothek enthalten
- die Typenkonvertierung Python$\Longleftrightarrow$C gelingt

Benutzung:

- laden der Bibliothek:`clib = ctypes.CDLL('libc.so.6')`
- Zugriff: `clib.printf('Hello \n')`

Tuturial und Referenz:
https://docs.python.org/3.3/library/ctypes.html

Datenumwandlung mit `ctypes`-Methoden

ctypes type	C type	Python type
c_bool	_Bool	bool
c_char	char	1-character bytes object
c_wchar	wchar_t	1-character string
c_byte	char	int
c_ubyte	unsigned char	int
c_short	short	int
c_ushort	unsigned short	int
c_int	int	int
c_uint	unsigned int	int
c_long	long	int
c_ulong	unsigned long	int
c_longlong	__int64 or long long	int
c_ulonglong	unsigned __int64 or unsigned long long	int
c_size_t	size_t	int
c_ssize_t	ssize_t or Py_ssize_t	int
c_float	float	float
c_double	double	float
c_longdouble	long double	float
c_char_p	char * (NUL terminated)	bytes object or None
c_wchar_p	wchar_t * (NUL terminated)	string or None
c_void_p	void *	int or None

Datenumwandlung mit `ctypes`-Methoden

- Arrays werden ebenfalls unterstützt, indem erst der Array-Typ deklariert wird:
  ```
  ein_cint_feld = ctypes.c_int * 23
  ```
- dann wird ein Objekt dieses Typs erzeugt:
  ```
  intfeld = ein_cint_feld()
  ```
- Nutzung wie andere sequenzielle Datentypen in Python:
  ```
  for i in range(23):
    intfeld[i] = i
  ```
- das Arbeiten mit Zeigern ist auch möglich

Datenumwandlung mit `ctypes`-Methoden

- Grundsätzlich sind auch komplexere Datenstrukturen möglich (also das Definieren eigener `structures`), in dem eine Klasse erzeugt wird, die von `ctypes.structures` erbt:

```
import ctypes as ct
class myatom(ct.Structure):
      # this field is mandatory, needs at least
      # a tuple of name/type
      _fields_ = [("labl", ct.c_char*2),
                  ("chrg", ct.c_int)]
he_atm = myatom('He'.encode('ASCII'),2)
y_atm = myatom('Y '.encode('ASCII'),39)
for atm in he_atm, y_atm:
    print( 'Das '+atm.labl.decode('ASCII').strip()+
    '-Atom hast die Kernladung:', atm.chrg))
```

Beispiel: die C-Routine

```
#include<stdio.h>
#include<Python.h>

int hellofunc(char name[])
{
  printf("Hello %s\n",name);
  return 0;
}
```

Diese Routine muss in eine dynamisch gelinkte Bibliothek geladen werden (*shared object*, lib<name>.so; DLL unter Windows). Beispiel mit `gcc` unter Linux (ZBH):

```
> gcc -I /usr/include/python3.4m/
  -c -O3 -Wall -Werror -fpic -o hellofunc.o hellofunc.c

> gcc -shared -o libhello.so hellofunc.o
```

Wichtig: Die Objektdatei muss mit der Option *position independent code*,`-fpic`, erzeugt werden

Beispiel: das Python-Skript

```
import ctypes as ct

#so-library laden
cbib = ct.CDLL("./libhello.so")

name = input(' Who do you want to great? (Default: World)\n ')

if name == '':
    name = 'World'
N = len(name)

# C-"ahnlichen Array erzeugen
cstring = ct.c_char * N
# Objekt des Arrays erzeugen
cname = cstring()
# f"ur C-char, String neu kodieren
cname = name.encode('ASCII')

# alternativ: String-buffer, damit Pointer auf einen
# ver"anderbaren String "ubergeben werden kann
#cname = ct.create_string_buffer(name.encode('ASCII'),N)

# C-Funkiton aufrufen
cbib.hellofunc(cname)
```