# Programmierkurs Python I

Michaela Regneri

2009-11-26

(Folien basieren auf dem gemeinsamen Kurs mit Stefan Thater)

## Übersicht

- Mehr zu Strings
- Module
- Ausnahmen
- Ein- und Ausgabe in Python
- Encodings

# Strings: Methoden

http://docs.python.org/3.1/library/stdtypes.html#string-methods

- `s1.count(s2)`: Anzahl der Vorkommen von s2 in s1
- Index des ersten (letzten) Vorkommens von s2 in s1:
  - `s1.index(s2[, start[, end]])` (`rindex`)
  - `s1.find(s2[, start[, end]])` (`rfind`)
    (Fehler, wenn s2 nicht gefunden wird)
- Charakteristika von `s1` (`False` für leere `s1`):
  - Ziffern? `s1.isdigit()`
  - Buchstaben? `s1.isalpha()`
  - Ziffern oder Buchstaben (+ '_'): `s1.isalnum()`
  - Leer(e )zeichen: `s1.isspace()`



# Strings: Methoden

http://docs.python.org/3.1/library/stdtypes.html#string-methods

- Methoden für Groß-/Kleinschreibung:
  - `s1.isupper()` / `s1.islower()`: alles groß / alles klein?
    (`False` für Strings ohne Buchstaben)
  - `s1.upper()` / `s1.lower()`: Kopie von `s1`, alle Buchstaben groß bzw. klein
  - `s1.capitalize()`: Kopie von `s1` mit 1. Zeichen in Großbuchstaben
  - `s1.swapcase()`: Kopie von `s1`, Groß- und Kleinschreibung ausgetauscht
  - `s1.title()` (auch: `s1.istitle()`): Kopie von `s1`; jeder Buchstabe nach einem Leer- oder Satzzeichen ist groß

# Strings: Methoden

http://docs.python.org/3.1/library/stdtypes.html#string-methods

- Leerzeichen [Zeichen aus `s2`] am Rand entfernen: `s1.strip([s2])` (`lstrip`, `rstrip`)
- Strings zerteilen: `s1.split([‘str‘])`
  - Rückgabe: ein Array aus Strings, die übrig bleiben, wenn man `s1` an allen Vorkommen von `str` durchschneidet
  - Wenn `str` nicht spezifiziert ist, wird an Leerzeichen getrennt
  - aufeinanderfolgende Trennzeichen trennen den leeren String

```
> ‘aa,,a.b‘.split([‘,‘])
> [‘aa‘, ‘‘, a.b‘]
```



# Module

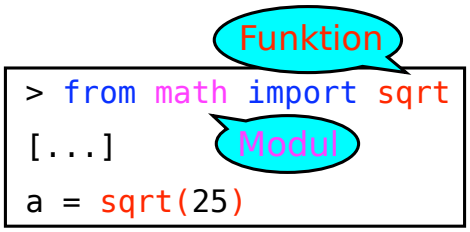
- Module sind Sammlungen von Klassen / Funktionen oder Code im allgemeinen (= *.py-Dateien)
- Module sind wiederverwertbar; man kann auf Code von anderen Modulen zugreifen
- Python hat (neben „builtins“) einige Standard-Module, auf die man bei Bedarf zurückgreifen kann (wie `sys`)
- Um die Module bzw. deren Elemente benutzen zu kennen, muss man sie importieren (mit `import <modulname>`)

```
> import sys
[...]
> a = sys.argv[0]
```

Modul-name

# Module

- Um die Datei foo.py als Modul zu benutzen, importiert man das Modul „foo“
- Man kann auch einzelne Klassen oder Funktionen eines Moduls importieren, mit `from`




- Python findet die Module ohne zusätzliche Angaben nur, wenn
  - sie im gleichen Ordner liegen wie das aktuelle Modul
  - sie im Python-Bibliotheksverzeichnis liegen (unter UNIX z.B. oft `/usr/local/lib/python/` )



# Module

- Man kann Module importieren, indem man den Pfad zu einem Unterverzeichnis explizit angibt:

```
import foo.bar.module
```

  wenn `module` im Unterordner `foo/baar` des aktuellen Verzeichnisses liegt

- mit dem Schlüsselwort as darf man Modulnamen an Variablen binden und später benutzen (praktisch für lange Namen)

```
import foo.bar.blah.blubb.module as fb
i = fb.methode()
```

# Ausnahmen (Exceptions)

- Ausnahmen sind Fehlermeldungen, die während des Programmablaufs auftreten
- Bisher haben wir versucht, Ausnahmen einfach zu vermeiden
- Es gibt Möglichkeiten, Ausnahmen zu behandeln, so dass das Programm nach der Ausnahme weiterläuft
- Außerdem kann es nützlich sein, Ausnahmen zu werfen (im Gegensatz zu leeren Rückgabewerten o.ä.)



# Ausnahmen (Exceptions)

- Es gibt eine Reihe von Exceptions, die in Pythons Standard-Modulen auftreten können (http://docs.python.org/3.1/library/exceptions.html)
- Ein Beispiel: Zugriff auf einen nicht vorhandenen Listen-Index

```
> l = [1,2]
> print(l[3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Name und Beschreibung der Ausnahme

Die Stelle, an der der Fehler auftrat

# Ausnahmen fangen

- Ausnahmen kann man fangen, mit „`try` ... `except`“
- Wenn in `block1` eine Ausnahme auftritt, wird der Code in `block1` an dieser Stelle abgebrochen und stattdessen `block2` ausgeführt
- danach läuft das Programm normal nach dem try-Konstrukt weiter
- Optional gibt es ein `else`- Statement nach `except`; `block3` wird ausgeführt, wenn in `block1` keine Ausnahme aufgetreten ist

```
try:
 block1
except:
 block2
```

```
try:
 block1
except:
 block2
else:
 block3
```



# Ausnahmen fangen

- `except:` fängt alles
- Um bestimmte Ausnahmen gesondert zu behandeln, schreibt man ihren Klassennamen in `except` (`except IndexError:` ...)
- Wenn man unterschiedliche Ausnahmen erwartet und auf jede anders reagieren will, kann man mehrere `except`-Blöcke definieren
- `else` kommt immer nach dem letzten `except`-Block

```
try:
 block1
except <Error1>:
 block2
except <Error2>:
 block3
[...]
else:
 blockx
```

# Ausnahmen: `finally`

- `finally` garantiert, dass der nachfolgende Code auf jeden (!) Fall ausgeführt wird
- Wird eine Ausnahme gefangen, wird zuerst `block2` ausgeführt, dann `block3`

```
try:
 block1
except <Exc>:
 block2
finally:
 block3
```

- Tritt eine unbehandelte Ausnahme auf, wird zuerst `block3` ausgeführt und dann die Ausnahme nochmals geworfen
- `else` kommt ggf. vor `finally` (in Notation und in der Ausführung)



# Ausnahmen als Klassen

- Alle in Python eingebauten Ausnahmen sind von `Exception` (bzw. `BaseException`) abgeleitet
- d.h. `except Exception` (`except BaseException`) fängt alle Ausnahmen (äquivalent zu `except` ohne Argument)
- Wenn man die konkrete Instanz einer Ausnahme zugreifen will, weist man sie im except-Statement mit `as` einer Variable zu

```
try:
 block1
except Exception as e:
 print(e)
```

# Ausnahmen definieren und werfen

- Man kann sich selbst Ausnahmen definieren
- Ausnahmen sollten von `Exception` erben (und müssen von `BaseException` erben)
- Die Standard-Nachricht wird in der `__str__`-Methode definiert

```
class MyIndexError(Exception):
  def __init__(self, length, index):
    self.length = length
    self.index = index

  def __str__(self):
    ret= 'Nur ' + str(self.length)
    ret+= ' Elemente in der Liste, '
    ret+= 'Index ' + str(self.index)
    ret+= ' ist nicht vergeben.'
    return ret
```



# Ausnahmen definieren und werfen

- Ausnahmen wirft man mit `raise` <Ausnahme>
- <Ausnahme> ist eine Instanz einer Exception-Klasse

```
> raise MyIndexError(2, 5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.MyIndexError: Nur 2 Elemente in
der Liste, Index 5 ist nicht vergeben.
```

- Wenn die `__init__`-Methode der Exception-Klasse keine zusätzlichen Argumente braucht, kann man einfach den Klassennamen hinschreiben

# Ausnahmen definieren und werfen

```
> raise Exception
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
Exception
```

- Die Basisklasse Exception hat ein *optionales* String-Argument

```
> raise Exception('Moep.')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
Exception: Moep.
```



# Ausnahmen definieren und werfen

- Wenn man eine Ausnahme ggf. werfen, aber davor noch etwas machen möchte, kann man `raise` ohne Parameter benutzen

```
try:
 block1
except:
 # mach irgendwas
 raise
```

- `raise` sucht nach gerade „aktiven“ Ausnahmen und wirft die neuste davon
- nach dem try-except-block ist die Ausnahme nicht mehr aktiv (auch nicht in finally)

# Ein- und Ausgabe: Konsole

- Ausgabe: Kennen wir schon (`print`)
- Kommandozeilenargumente: `sys.argv[i]`
- Interaktion während des Programmablaufs:
  `input([string])`
  - `string` wird vorm Lesen der Eingabe ausgegeben
  - Rückgabe enthält die Benutzer-Eingabe nach dem Methodenaufruf (mit Enter „abgeschickt“)
  - `input` gibt den eingegebenen String zurück



# Ein- und Ausgabe: Konsole

- ein Beispiel:

```
def trainMultiplication(x,y):
    i = input(str(x) + ' * ' + str(y) + '= ?\n')
    if int(i) == (x * y):
        print('Richtig!')
    else:
        print('Falsch.')
```

```
> trainMultiplication(15,7)
15 * 7 = ?
105
Richtig!
```

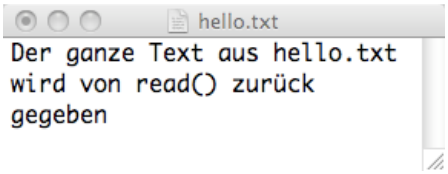
Benutzer-Eingabe

Ausgabe von `input`

Ausgabe von `print`

# Ein- und Ausgabe: Dateien

- Arbeit mit Dateien in Python funktioniert mit `file`-Objekten
- am einfachsten bekommt man die mit `open(string)`:




```
> f = open('hello.txt')
> f.read()
'Der ganze Text aus hello.txt\n wird von read()\n
zurück\n gegeben'
> f.close()
```



# Ein- und Ausgabe: Datei-Handhabung

- sämtliche Operationen auf Dateien starten bei der aktuellen „Position“ in der Datei
- Die Position ändert sich beim Lesen / Schreiben; direkt nach dem Öffnen ist sie 0
- Ausgabe der aktuellen Position: `f.tell()`
- Setzen der aktuellen Position: `f.seek(index)`
- Um Fehler zu vermeiden, muss man geöffnete Dateien wieder schließen, wenn sie nicht mehr benötigt werden: `f.close()`

# Kurzer Exkurs: das `with`-Statement

```
with foo as var:
  block
```

- `with` stellt u.a. sicher, dass die benutzen Objekte einen definierten „Lebenszyklus“ durchlaufen
- für Dateien heißt das konkret, dass sie nach dem `with`-Block geschlossen werden
- abstrakter: vor dem with-Block wird die Methode `__enter__` (von foo), und nach dem `with`-block die Methode `__exit__` aufgerufen

```
with open('hello.txt') as f:
   f.read()
```



# Ein- und Ausgabe: Dateien lesen

- `f.read()`: gibt den (Text-)Inhalt von `f` zurück
- `f.readline()`: gibt `f` Zeile für Zeile zurück (neuer Aufruf - nächste Zeile)
- `f.readlines()`: gibt eine Liste der Zeilen zurück
- über die Zeilen von `f` direkt iterieren:

```
with open(file) as f:
  int i = 1
  for l in f:
    print(i + '. Zeile: ' + l)
```

Die Position nach dem zuletzt gelesenen Zeichen in der Datei wird gespeichert; alle Lese-Methoden lesen ab der aktuellen Position!

# Ein- und Ausgabe: Dateien Schreiben

- Schreibzugriff auf Dateien bekommt man durch zusätzliche Parameter (*Flags*) in `open`:
  - `open(f,'w')`: Schreibzugriff
  - `open(f,'a')`: Schreibzugriff, Text wird angehängt
  - `open(f,'r+')`: Lese- und Schreibzugriff
  - `open(f,'r+a')`: Lese- und Schreibzugriff (Text wird angehängt)
  - `open(f, 'r')`: Lesezugriff
- Ohne zweiten Parameter: nur Lesezugriff
- Über die Variable `f.mode` kann dieser „Modus“ wieder abgerufen werden



# Ein- und Ausgabe: Dateien Schreiben

- `f.write(string)`: schreibt `string` in `f`
- `f.writelines(seq)`
  schreibt alle Elemente aus `seq` (ein Sammelobjekt) in `f` (keine automatische Zeilentrennung!)
- `f.flush()`:
  schreibt bisher mit write ausgegebenen Text tatsächlich in die Datei; wird mit `f.close()` (und durch einbetten in with) automatisch aufgerufen

# Ein- und Ausgabe: URLs

- `urllib.request` erlaubt das öffnen von URLs
- (den Quelltext von) Webseiten auslesen funktioniert ähnlich wie Dateien auslesen:

```
import urllib.request as url
hp = 'http://www.coli.uni-saarland.de'
for line in url.urlopen(hp):
    print(line)
```

- die von urlopen erzeugten Objekte unterstützen als Lesemethoden `read()` und `readlines()`
- Webseite in lokale Datei kopieren:

```
[...]
url.urlretrieve(hp, 'dateiname.html')
```



# Unicode-Strings vs. Byte-Strings

- Python kennt zwei Typen von Strings: Unicode- und Byte-Strings (`str` und `bytes`)
  - Standard-String-Literale (`"x"`,`'y'`) sind Unicode-Strings
  - `b“wort“` erzeugt einen Byte-String
- Bytestrings werden intern als Folge von Bytes kodiert (einschränkung auf maximal 255 verschiedene Zeichen)
- Unicode-Strings werden intern als Folge von 2 bzw. 4 Bytes dargestellt (decken praktisch alle Alphabete ab)

# Unicode-Strings vs. Byte-Strings

- Man darf die beiden nicht mischen (bei Konkatenation etc.), sondern muss konvertieren:
  - String → Byte: `str.encode(unicodeString)`
  - Byte → String: `bytes.decode(byteString)`
- `urlopen` gibt Byte-Strings zurück, open als Standard Unicode-Strings (!!!) - umgekehrt darf man dann auch nur die reinschreiben!
- Wenn keine explizite Kodierung angegeben ist, wird die ASCII-Kodierung angenommen



# Unicode-Strings vs. Byte-Strings

- wenn man nichts über die Datei weiß, kann es einfacher sein, nur mit Byte-String zu arbeiten (bis man eine lesbare Ausgabe braucht)

- lesen (und schreiben) einer Datei als Byte-String: `open(f,'br')`
  - **b** kann einfach vor die anderen *Flags* in open geschrieben werden
  - wenn man **b** als Flag angibt, braucht man immer einen zweiten Parameter der angibt, ob man die Datei lesen oder editieren (etc.) will

# Kodierungen (Encodings)

- Strings kann man sich als Folgen von Zeichen vorstellen
- Computer kennen aber keine Zeichen: Intern werden Strings als Folgen von Zahlen repräsentiert
- Wir brauchen also eine Abbildung, die Zahlen und Zeichen einander zuordnet
- Solche Abbildungen nennt man auch Kodierungen (Encodings)



# Kodierungen

- ASCII ist eine einfache (7-Bit) Kodierung, die den Zeichen der englischen Sprache Zahlen zwischen 32 und 127 zuweist (Zahlen ≤ 31 sind Steuerzeichen).

```
>>> for c in 'python':
...   print(ord(c), end=“ “)
112 121 116 104 111 110
```

- ASCII kennt keine Umlaute etc.
- Einige Erweiterungen von ASCII
  - ISO-8859-1 („latin1“) – westeuropäische Sprachen
  - ISO-8859-2 („latin2“) – osteuropäische Sprachen

# Kodierung festlegen

- Wenn der Quelltext nicht in ASCII vorliegt, muss die Kodierung für String-Literale explizit festgelegt werden:

```
# -*- coding: latin1 -*-
print("Hällo, Wörld!")
```

- Ohne explizite Angabe der Kodierung kompiliert obiges Beispiel nicht. Für die gleiche Funktionalität:

```
print("H\xe4llo, W\xf6rld!")
```



# Unicode

- Was machen wir, wenn wir Texte in verschiedenen Kodierungen verarbeiten wollen?
- Oder Sprachen mit mehr als 256 Zeichen?
- Unicode!
  - gibt die Einschränkung auf, dass Zeichen als ein Byte dargestellt werden müssen
  - umfasst alle (die allermeisten) Zeichen der meisten Sprachen

# Unicode und Kodierungen

- Unicode legt fest, wie Zeichen als Code-Points repräsentiert werden
  - Die Code-Points 0–256 sind mit Latin-1 identisch
- Code-Points sind Zahlen (hier representiert als Hex)

```
0061    'a'; LATIN SMALL LETTER A
0062    'b'; LATIN SMALL LETTER B
0063    'c'; LATIN SMALL LETTER C
...
007B    '{'; LEFT CURLY BRACKET
```



# Unicode und Kodierungen

- Eine Kodierung legt fest, wie Unicode-Zeichen im Speicher repräsentiert werden.
- Kodierungen können unvollständig sein (z.B., ASCII).
- Eine „naive“ vollständige Kodierung würde jedes Zeichen als Folge von 32-Bit Zahlen (4 Bytes) darstellen.
  - Aber: Plattformabhängig (Byte-Ordnung), hoher Speicherbedarf, Repräsentation enthält Nullen

# Unicode Transformation Format

- UTF-8 ist eine häufig verwendete, kompakte (8-Bit) Kodierung für Unicode:
  - kann alle Unicode Code-Points darstellen
  - die meisten Zeichen (ASCII) werden durch ein einzelnes Byte dargestellt.
- Kodierung:
  - Code-Point < 128 ⇒ 1 Byte
  - Code-Point ≥ 128 ⇒ 2–4 Bytes
- Anmerkung: UTF-8 ist nicht Unicode!



# Unicode & Dateien

- Stream-Objekte (Dateien, URLs) haben immer irgend eine Encodierung
- Wenn man sie nicht kennt, kann man einfach mit Byte-Strings arbeiten, so lange es geht
- Wenn man String braucht, muss man sie aber wieder decodieren, entweder so:

```
with open(file,encoding="UTF-8") as f:
```

...oder so:

```
with open(file,'br') as f:
  for line in f.readlines():
    astring = str(line, encoding="UTF-8")
```

# Zusammenfassung

- Mehr Basics: Module & Ausnahmen
- Input/Output: Konsole, Dateien, URLs
- String-Handling: Byte- vs. Unicdoe-Strings, Encodings

- nächste Woche: reguläre Ausdrücke, mehr zu Web-IO