Programmieren in Python

8. Funktionen, Ausnahmen und Module

Robert Mattmüller

Albert-Ludwigs-Universität Freiburg

Handlungsplanungs-Praktikum
Wintersemester 2010/2011

Funktionen, Ausnahmen und Module

In dieser Lektion vertiefen wir das Thema Funktionsdefinitionen und behandeln mit der Fehlerbehandlung und dem Erstellen und Verwenden von Modulen zwei weitere essentielle Themen.

Im Überblick:

- Mehr zu Funktionen
- Ausnahmen
- Module

Funktionen, Ausnahmen und Module

In dieser Lektion vertiefen wir das Thema Funktionsdefinitionen und behandeln mit der Fehlerbehandlung und dem Erstellen und Verwenden von Modulen zwei weitere essentielle Themen.

Im Überblick:

- Mehr zu Funktionen
- Ausnahmen
- Module

Mehr zu Funktionen

- Wir haben mittlerweile viele Funktionen gesehen, die eine variable Argumentzahl akzeptieren. Solche Funktionen können wir noch nicht selbst schreiben.
- Noch viel weniger können wir Funktionen wie den eigentümlichen `dict`-Konstruktor selbst schreiben, der Aufrufe wie `dict(spam=10, bar=20)` zulässt.
- Das macht uns neidisch.
- Daher werden solche Funktionsdefinitionen jetzt erklärt.
- Außerdem sprechen wir über lokale und globale Variablen.

Lokale und globale Variablen

- Python unterscheidet zwischen lokalen Variablen, die nur in der aktuellen Funktion gültig sind, und globalen Variablen, die in der ganzen Datei (im ganzen Modul) gültig sind.
- Da Python ja Variablen nicht deklariert, fragen wir uns: Wie wird zwischen lokalen und globalen Variablen unterschieden?

- Variablen außerhalb von Funktionen sind global (klar).
- Variablen innerhalb von Funktionen sind lokal, wenn ihnen innerhalb der Funktion ein Wert zugewiesen wird, und ansonsten global.
- Will man globalen Variablen innerhalb einer Funktion einen neuen Wert zuweisen, markiert man die Variable mit der Anweisung `global var1, var2, ...` (innerhalb der Funktion!) als global.

Lokale und globale Variablen: Beispiel

locals_and_globals.py

```
debug = False

def set_debug():
  global debug
  debug = True

def compute_sum(x, y):
  if debug:
    print("Computing sum of %s and %s" % (x, y))
  return x + y

print(compute_sum(10, 20))
set_debug()
print(compute_sum(20, 10))
```

Die Funktion `globals` (1)

- Python besitzt umfangreiche Möglichkeiten zur Introspektion (das, was man in Java als „Reflection“ bezeichnet).
- Ein (einfaches) Beispiel dafür sind die Builtins `globals` und `locals`.

- `globals()`:
  Liefert ein Dictionary, dessen Schlüssel und Werte die Namen und Inhalte der globalen Variablen im Modul (d.h. in dieser Datei) sind.
  - Das Dictionary kann mit den üblichen Methoden modifiziert werden. Solche Änderungen haben denselben Effekt wie tatsächliche Zuweisungen an globale Variablen.
  - Auf diese Weise können sogar Variablen mit ansonsten unzulässigen Namen wie `"10Euro50"` oder `"@!$%"` erzeugt werden, auch wenn dies nicht sonderlich nützlich ist.

Die Funktion `globals` (2)

modify_globals.py

```
globals()["egg"] = 2
print(egg)              # Ausgabe: 2
spam = 0
mydict = globals()
mydict["spam"] = 10
print(spam)             # Ausgabe: 10
y = 0
del mydict["y"]
print(y)                # NameError: 'y' is not defined
```

- Nebenbei bemerkt:
  Um wie im Beispiel die globale Variable `var` zu entfernen, kann man auch einfach die Anweisung `del var` verwenden.

Die Funktion `locals`

`locals()`:

- Außerhalb von Funktionen aufgerufen:
  Identisch mit `globals()`.
- Innerhalb einer Funktion aufgerufen:
  Liefert analog zu `globals()` ein Dictionary mit Namen und Werten der *lokalen* Variablen.
  Allerdings ist dies nur eine Kopie des „internen“ Dictionaries, so dass es nicht möglich ist, mithilfe von `locals()` lokale Variablen neu einzuführen, zu löschen oder neu zu binden.

locals und globals: Beispiel

vars.py

```
def f(bo, moe):
  print(locals())  # {'bo': 'parrot', 'moe': 'fjord'}
  jim = "egg"
  locals()["jim"] = 111
  print(jim)       # egg
  del moe
  print(locals())  # {'jim': 'egg', 'bo': 'parrot'}
  print(globals()) # {'f': <function f at 0x402f0844>,
                   #  'bo': 'ham', 'joe': 'spam', ...}

bo = "ham"
joe = "spam"
f("parrot", "fjord")
```

Erweiterte Argumentlisten: Einführung

- Funktionen wie `min` und `max` akzeptieren eine variable Anzahl an Argumenten.
- Funktionen wie der `dict`-Konstruktor oder die `sort`-Methode von Listen akzeptieren sogenannte *benannte Argumente*.
- Beides können wir auch in selbst definierten Funktionen verwenden.
- Bevor wir dazu kommen, wollen wir erst einmal beschreiben, was benannte Argumente sind.

Benannte Argumente (1)

- Betrachten wir folgende Funktion:

```
def power(base, exponent):
  return base ** exponent
```

- Bisher haben wir solche Funktionen immer so aufgerufen:

```
print(power(2, 10))                          # 1024.
```

- Tatsächlich geht es aber auch anders:

```
print(power(base=2, exponent=10))            # 1024.
print(power(2, exponent=10))                 # 1024.
print(power(exponent=10, base=2))            # 1024.
```

Benannte Argumente (2)

- Zusätzlich zu „normalen“ (sog. *positionalen*) Argumenten können beim Funktionsaufruf auch *benannte* Argumente mit der Notation `var=wert` übergeben werden.
- `var` muss dabei der Name eines Parameters der aufgerufenen Funktion sein:

Python-Interpreter

```
>>> def power(base, exponent):
...   return base ** exponent
...
>>> print(power(x=2, y=10))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: power() got an unexpected keyword argument 'x'
```

Benannte Argumente (3)

- Benannte Argumente müssen am Ende der Argumentliste (also nach positionalen Argumenten) stehen:

Python-Interpreter

```
>>> def power(base, exponent):
...   return base ** exponent
...
>>> print(power(base=2, 10))
SyntaxError: non-keyword arg after keyword arg
```

Benannte Argumente (4)

- Ansonsten dürfen benannte Argumente beliebig verwendet werden.
- Insbesondere ist ihre Reihenfolge vollkommen beliebig.

- Konvention:
  Während man bei Zuweisungen allgemein Leerzeichen vor und nach das Gleichheitszeichen setzt, tut man dies bei benannten Argumenten nicht — auch um deutlich zu machen, dass hier *keine Zuweisung* im normalen Sinne stattfindet, sondern nur eine ähnliche Syntax benutzt wird.

Default-Argumente

- Besonders interessant sind benannte Argumente in Zusammenhang mit *Default-Argumenten*, die wir jetzt kennenlernen werden:

```
def power(base, exponent=2, debug=False):
  if debug:
    print(base, exponent)
  return base ** exponent
```

- Default-Argumente können beim Aufruf weggelassen werden und bekommen dann einen bestimmten Wert zugewiesen.
- Zusammen mit benannten Argumenten können wir schreiben:

```
print(power(10))                    # 100.
print(power(10, 3, False))          # 1000.
print(power(10, debug=True))        # 10 2; 1000.
print(power(debug=True, base=4))    # 4 2; 16.
```

Achtung bei veränderlichen Default-Argumenten (1)

- Veränderliche Default-Argumente werden nur einmal ausgewertet (zum Zeitpunkt der Funktionsdefinition), nicht bei jedem Aufruf.
- Mutiert man daher ein Default-Argument, hat das Auswirkungen auf spätere Funktionsaufrufe:

mutable_default_arg.py

```
def test(spam, egg=[]):
  egg.append(spam)
  print(egg)

test("parrot")   # Ausgabe: ['parrot']
test("fjord")    # Ausgabe: ['parrot', 'fjord']
```

Achtung bei veränderlichen Default-Argumenten (2)

- Aus diesem Grund sollte man in der Regel keine veränderlichen Default-Argumente verwenden.
  Das übliche Idiom ist das Folgende:

mutable_default_arg_corrected.py

```
def test(spam, egg=None):
  if egg is None:
    egg = []
  egg.append(spam)
  print(egg)

test("parrot")    # Ausgabe: ['parrot']
test("fjord")     # Ausgabe: ['fjord']
```

- Manchmal sind veränderliche Default-Argumente allerdings gewollt, etwa zur Implementation von *memoization*.

Variable Argumentlisten

- Das letzte fehlende Puzzlestück sind *variable Argumentlisten*. Mit diesen kann man Funktionen definieren, die beliebig viele positionale Argumente, beliebig viele benannte Argumente, oder beides unterstützen.
- Die Idee ist ganz einfach: Alle „überzähligen“ positionalen Parameter werden in ein Tupel, alle überzähligen benannten Argumente in ein Dictionary gepackt.
- Notation:
  - `def f(x, xy, *spam):`
    `f` benötigt mindestens zwei Argumente. Weitere positionale Argumente werden im Tupel `spam` übergeben.
  - `def f(x, **egg):`
    `f` benötigt mindestens ein Argument. Weitere benannte Argumente werden im Dictionary `egg` übergeben.
- „Gesternte“ Parameter müssen am Ende der Argumentliste stehen. Werden beide verwendet, dann in der Reihenfolge `*spam` vor `**egg`.

Variable Argumentlisten: Beispiel (1)

varargs.py

```
def v(spam, *argtuple, **argdict):
  print(spam, argtuple, argdict)

v(0)                          # 0 () {}
v(1, 2, 3)                    # 1 (2, 3) {}
v(1, ham=10)                  # 1 () {'ham': 10}
v(ham=1, jam=2, spam=3) # 3 () {'jam': 2, 'ham': 1}
v(1, 2, ham=3, jam=4)    # 1 (2,) {'jam': 4, 'ham': 3}
```

Variable Argumentlisten: Beispiel (2)

vararg_examples.py

```
def product(*numbers):
  result = 1
  for num in numbers:
    result *= num
  return result

def make_pairs(**argdict):
  return list(argdict.items())

print(product(5, 6, 7))
# Ausgabe: 210

print(make_pairs(spam="nice", egg="ok"))
# Ausgabe: [('egg', 'ok'), ('spam', 'nice')]
```

`list.sort` und `sorted`

Zur `sort`-Methode der Klasse `list` und dem Builtin `sorted` mussten wir früher sagen, dass wir einige Optionen noch nicht erklären konnten. Jetzt können wir es:

- `l.sort(key=None, reverse=False)`:
  Sortiert die Liste.
  Die optionalen Argumente haben folgende Bedeutung:
  - `key` sollte entweder `None` oder eine Funktion sein, die ein einzelnes Argument akzeptiert. In letzterem Fall werden zum Anordnen der Elemente `x` und `y` nicht die Elemente selbst, sondern `key(x)` und `key(y)` verglichen.
  - Wenn `reverse` wahr ist, dann wird absteigend sortiert.
- `sorted(iterable, key=None, reverse=False)`:
  Erstellt eine Liste aus `iterable`, sortiert diese und liefert sie zurück.
  Die Argumente `key` und `reverse` haben dieselbe Bedeutung wie bei `list.sort`.

Bemerkung: `min` und `max` unterstützen auch `key`.

Sortieren mit `key`: Beispiel

sort_example.py

```
food = ["Spam", "egg", "Ham", "spam", "baked beans"]
print(sorted(food))
# ['Ham', 'Spam', 'baked beans', 'egg', 'spam']

def lower(astring):
  return astring.lower()
print(sorted(food, key=lower))
# ['baked beans', 'egg', 'Ham', 'Spam', 'spam']
# sort() bzw. sorted() ist stabil!

mynumbers = [10, 2.4, -20, 100, 5j, -7]
mynumbers.sort(key=abs, reverse=True)
print(mynumbers)
# [100, -20, 10, -7, 5j, 2.4]
```

Erweiterte Aufrufsyntax

- Die Notationen `*argtuple` und `**argdict` können nicht nur in Funktionsdefinitionen verwendet werden, sondern auch in *Funktionsaufrufen*.
- Dabei bedeutet beispielsweise

  ```
  f(1, x=2, *argtuple, **argdict),
  ```

  dass als positionale Parameter eine `1` gefolgt von den Elementen aus `argtuple` und als benannte Parameter `x=2` sowie die Paare aus `argdict` übergeben werden.
- Man nennt dies die *erweiterte Aufrufsyntax*.

Erweiterte Aufrufsyntax: Beispiel

- Eine nützliche Anwendung der erweiterten Aufrufsyntax besteht darin, die eigenen Argumente an eine andere Funktion weiterzureichen, ohne deren genaue Signatur zu kennen. Beispiel:

```
def my_function(*argtuple, **argdict):
  print("Arguments for other_function:", end=' ')
  print(argtuple, argdict)
  result = other_function(*argtuple, **argdict)
  print("other_function returns:", result)
  return result
```

- In etwas verfeinerter Form wird diese Idee häufig bei sogenannten *Dekoratoren* verwendet, die wir hier aus Zeitgründen aber nicht diskutieren wollen.

Funktionen, Ausnahmen und Module

In dieser Lektion vertiefen wir das Thema Funktionsdefinitionen und behandeln mit der Fehlerbehandlung und dem Erstellen und Verwenden von Modulen zwei weitere essentielle Themen.

Im Überblick:

- Mehr zu Funktionen
- Ausnahmen
- Module

Ausnahmen (1)

- In vielen unserer Beispiele sind uns *Tracebacks* wie der folgende begegnet:

Python-Interpreter

```
>>> print({"spam": "egg"}["parrot"])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'parrot'
```

- Bisher konnten wir solchen Fehlern weder abfangen noch selbst entsprechende Fehler melden. Das wollen wir jetzt ändern.

Ausnahmen (2)

- Ebenso wie Java und C++ kennt Python das Konzept der *Ausnahmebehandlung* (*exception handling*).
- Wird eine Funktion mit einer Situation konfrontiert, mit der sie nichts anfangen kann, kann sie eine Ausnahme signalisieren.
- Die Funktion wird dann beendet und es wird solange zur jeweils aufrufenden Funktion zurückgekehrt, bis sich eine Funktion findet, die mit der Ausnahmesituation umgehen kann.

- Zur Ausnahmebehandlung dienen in Python die Anweisungen `raise`, `try`, `except`, `finally` und `else`.

Die `raise`-Anweisung

- Mit der `raise`-Anweisung kann eine Ausnahme signalisiert werden (entsprechend `throw` in Java und C++).
- Dazu verwendet man `raise` zusammen mit einer Instanz einer speziellen Ausnahmeklasse (beispielsweise `IndexError` oder `NameError`). Üblicherweise erzeugt man die Ausnahme in derselben Anweisung, in der man sie auch signalisiert:

  ```
  raise KeyError("Fehlerbeschreibung")
  ```
  .
- Die Beschreibung kann auch weggelassen werden; die Form `raise KeyError()` ist also auch zulässig.
- Auch die Notation `raise KeyError` ist erlaubt; in diesem Fall wird automatisch eine Instanz erzeugt.

try-except-Blöcke

- Funktionen, die Ausnahmen behandeln wollen, verwenden dafür try-except-Blöcke, die wie in folgendem Beispiel aufgebaut sind:

```
try:
    call_critical_code()
except NameError as e:
    print("Sieh mal einer an:", e)
except KeyError:
    print("Oops! Ein KeyError!")
except (IOError, OSError):
    print("Na sowas!")
except:
    print("Ich verschwinde lieber!")
    raise
```

`except`-Spezifikationen (1)

- Das Beispiel zeigt, dass es verschiedene Arten gibt, `except`-Spezifikationen zu schreiben:
  - Die normale Form ist `except XYError as e`.
    Ein solcher Block wird ausgeführt, wenn innerhalb des `try`-Blocks eine Ausnahme der Klasse `XYError` (oder einer abgeleiteten Klasse) auftritt und weist der Variablen `e` die Ausnahme zu.
  - Interessiert die Ausnahme nicht im Detail, kann die Variable auch weggelassen werden, also die Notation `except XYError` verwendet werden.
  - Bei beiden Formen kann man auch mehrere Klassen gemeinsam behandeln, indem man die Klassen in ein Tupel schreibt, also z.B. `except (XYError, YZError) as e`.
  - Schließlich gibt es noch die Form `except` ohne weitere Angaben, die beliebige Ausnahmen behandelt (nicht empfohlen, da damit z. B. auch `Ctrl+C` abgefangen wird).

`except`-Spezifikationen (2)

- `except`-Blöcke werden wie in anderen Programmiersprachen der Reihe nach abgearbeitet, bis der erste passende Block gefunden wird (falls überhaupt einer passt).
- Die Reihenfolge ist also wichtig; unspezifische `except`-Blöcke sind nur als letzter Test sinnvoll.
- Um beliebige Ausnahmen abzufangen *und* die Ausnahme an eine Variable zu binden, ist die Spezifikation `except Exception as e` geeignet, da alle üblicherweise abzufangenden Ausnahmeklassen von `Exception` abgeleitet sind.
- Stellt sich innerhalb eines `except`-Blocks heraus, dass die Ausnahme nicht vernünftig behandelt werden kann, kann sie mit einer `raise`-Anweisung ohne Argument weitergereicht werden.

try – except – else

- Ein try-except-Block kann mit einem else-Block abgeschlossen werden, der ausgeführt wird, falls im try-Block keine Ausnahme ausgelöst wurde:

```
try:
    call_critical_code()
except IOError:
    print("IOError!")
else:
    print("Keine Ausnahme")
```

`try-finally`-Blöcke

- Manchmal kann man Ausnahmen nicht behandeln, möchte aber darauf reagieren – etwa um Netzwerkverbindungen zu schließen oder andere Ressourcen freizugeben.
- Dazu dient der `try-finally`-Block:

```
try:
    call_critical_code()
finally:
    print("Das letzte Wort habe ich!")
```

- Der `finally`-Block wird *auf jeden Fall* ausgeführt, wenn der `try`-Block betreten wird, egal ob Ausnahmen auftreten oder nicht. Auch bei einem `return` im `try`-Block wird der `finally`-Block vor Rückgabe des Resultats ausgeführt.
- Wurde eine Ausnahme signalisiert, wird sie nach Behandlung des `finally`-Blocks weitergegeben.

Verwendung von Ausnahmen in Python

- Ausnahmen sind in Python allgegenwärtig. Da Ausnahmebehandlung im Vergleich zu anderen Programmiersprachen einen relativ geringen Overhead erzeugt, wird sie oft in Situationen eingesetzt, in denen man sie durch zusätzliche Tests vermeiden könnte.
- Man spricht vom EAFP-Prinzip:
  „**It's easier to ask forgiveness than permission.**“
- Beispielsweise würde man in Python nicht testen, ob eine Datei existiert, bevor man sie zum Lesen öffnet, sondern das Öffnen versuchen und gegebenfalls die Ausnahme abfangen, die das Fehlen der Datei signalisiert. Zugleich robusteres Vorgehen z. B. bei mehreren Threads.

Ausnahmeklassen in den Builtins

- Pythons Builtins enthalten eine große Zahl an Ausnahmeklassen, die hier aufzuzählen nicht sehr sinnvoll wäre. Ein Überblick befindet sich auf der Seite:
  `http://docs.python.org/py3k/library/exceptions.html`

Eigene Ausnahmeklassen

Als kleiner Vorgriff auf die Diskussion von Klassen hier das Kochrezept zum Definieren eigener Ausnahmeklassen:

```
class MyException(BaseClass):
    pass
```

- `MyException` kann dann genauso verwendet werden wie eingebaute Ausnahmeklassen wie `IndexError`.
- Für `BaseClass` wird man oft `Exception` wählen, aber natürlich eignen sich auch andere Ausnahmeklassen.

- Nebenbemerkung: `pass` ist die Python-Anweisung für „tue nichts“.

Funktionen, Ausnahmen und Module

In dieser Lektion vertiefen wir das Thema Funktionsdefinitionen und behandeln mit der Fehlerbehandlung und dem Erstellen und Verwenden von Modulen zwei weitere essentielle Themen.

- Mehr zu Funktionen
- Ausnahmen
- Module

Module

- Bisher waren unsere Programme auf einzelnen Dateien beschränkt und haben außer Builtins auch keine Funktionen oder Klassen benutzt, die außerhalb der Datei definiert sind.
- Das ist bei richtigen Programmen natürlich fast nie der Fall: Zum einen will man fast immer externe Bibliotheken benutzen, zum anderen möchte man umfangreichere Programme auf mehrere Sinneinheiten verteilen.
- Solche Sinneinheiten sind in Python typischerweise *Module*.
- Neben Modulen gibt es noch *Pakete*, Sammlungen von Modulen unter einem gemeinsamen Namen, aber mit denen werden wir uns nicht beschäftigen – man kommt meist auch gut ohne aus.

Module erstellen und verwenden

- Wir haben schon die ganze Zeit über Module erstellt: Jede `*.py`-Datei ist ein Modul, und jedes Modul ist eine `*.py`-Datei.
- Noch nicht besprochen haben wir, wie man auf externe Module zugreifen kann. Dafür gibt es zwei Anweisungen, das (häufigere)

  ```
  import mymodule
  ```

  und das seltener verwendete

  ```
  from mymodule import X.
  ```

Die `import`-Anweisung

Die Anweisung `import mymodule` tut drei Dinge:

- Zunächst wird überprüft, ob das Modul `mymodule` bereits früher von unserem Programm geladen wurde (z.B. von einem anderen Modul aus). Falls ja, wird das damals erzeugte Modulobjekt verwendet und die folgenden Schritte übersprungen. Module werden also nicht mehrfach geladen.
- Dann wird die Datei `mymodule.py` an verschiedenen Stellen (aktuelles Verzeichnis, Standardbibliothek) gesucht und ausgeführt, als würde man sie mit `python mymodule.py` aufrufen. Die meisten Module enthalten nur (Funktions- und Klassen-) Definitionen, aber es ist durchaus möglich, in diesem Initialisierungsschritt beliebigen Code auszuführen.
- Schließlich wird ein neues *Modul-Objekt* (vom Typ `module`) erzeugt und der Variable `mymodule` zugewiesen. Die globalen Variablen von `mymodule.py` sind in dem importierenden Modul als Attribute von `mymodule` verfügbar.

import: Beispiel

helpers.py

```
def product(x, y, z):
  return x * y * z
food = "spam"
print("helpers ist dran.")
```

import_example.py

```
print("Start")       # Ausgabe: Start
import helpers       # Ausgabe: helpers ist dran.
print(helpers.food) # Ausgabe: spam
print(helpers)       # Ausgabe: <module 'helpers' from
                     # '/home/robert/.../helpers.pyc'>
del helpers
import helpers       # keine Ausgabe
print(helpers.product(2, 3, 4))  # Ausgabe: 24
```

Bytecode und `*.pyc`-Dateien

- Wir stellen fest, dass Python bei der Ausführung unseres Testbeispiels eine Datei namens `helpers.pyc` angelegt hat.
- Dabei handelt es sich um *Bytecode* vergleichbar `class`-Dateien in Java.
- Python-Code wird vor der Ausführung immer in Bytecode für die Python Virtual Machine übersetzt. Normalerweise merkt man davon nichts, aber beim Importieren von Modulen wird der Bytecode für die Module gespeichert, damit er bei späteren Programmaufrufen sofort verfügbar ist.
- Ändert sich eine `*.py`-Datei, wird die zugehörige `*.pyc`-Datei automatisch neu generiert — man kann also nichts falsch machen. Es schadet aber auch nie, `*.pyc`-Dateien von Hand zu löschen, um ein Verzeichnis aufzuräumen.

Die Anweisung `from mymodule import x`

Die Anweisung `from mymodule import x` verhält sich ähnlich wie `import mymodule`. Die ersten zwei Schritte (Überprüfen ob `mymodule` schon geladen ist und ggf. Ausführen der Datei) sind identisch.
Der Unterschied liegt im dritten Schritt:

- Bei `from mymodule import x` wird *keine* Variable namens `mymodule` angelegt, sondern die Variable `x`, die der globalen Variable `x` aus `mymodule` entspricht. Es wird also nicht das ganze Modul, sondern nur ein Objekt verfügbar, und dieses kann ohne Punkt-Notation verwendet werden.

from ... import: Beispiel

helpers.py

```
def product(x, y, z):
  return x * y * z
food = "spam"
print("helpers ist dran.")
```

from_import_example.py

```
print("Start")                    # Start
from helpers import food          # helpers ist dran.
print(food)                       # spam
from helpers import product       # <keine Ausgabe>
print(product(8, 8, 8))           # 512
```

Allgemeinere Formen von `import` und `from ... import`

Es ist möglich, mehrere Module bzw. Objekte gleichzeitig zu importieren und Module bzw. Objekte unter einem anderen Namen zu importieren:

- `import spammodule, eggmodule`
- `import mymodule as mymod`
- `from monty import spam, egg`
- `from monty import spam as food`
- `from mymodule import *`

Die Standardbibliothek

- Python ist von Haus aus mit einer Vielzahl von Modulen ausgestattet.
- Die mitgelieferten Module bezeichnet man als Pythons *Standardbibliothek*.
- Es wäre unsinnig zu versuchen, hier einen Überblick über die Standardbibliothek zu geben; dafür ist sie viel zu umfangreich.
- Stattdessen ein Link zum Inhaltsverzeichnis: `http://docs.python.org/py3k/library/`