Informatik I: Einführung in die Programmierung

14. Funktionsaufrufe & Ausnahmebehandlung

Albert-Ludwigs-Universität Freiburg

Bernhard Nebel

24. November 2013

1 PEP8: Der Stil-Checker

Stil-Konventionen

- Wir haben immer mal wieder gesehen, dass es Stil-Vorgaben für Python gibt: PEP8, siehe `http://www.python.org/dev/peps/pep-0008/`
  - Sehr wichtige, wie keine Mischung von Tabs und Leerzeichen
  - Ästhetische, wie die Platzierung von Leerzeichen
  - Vereinheitlichende, wie die Schreibweise von Variablen, Funktionen usw.
- Benutzen Sie einen Stil-Checker!
- Online: Z.B. `http://pep8online.com`
- Offline:
  1. Installieren Sie den Python-Package-Manager `pip`: `http://www.pip-installer.org/en/latest/installing.html` (sollte aber bereits da sein!)
  2. dann das Paket `pep8`: `pip install pep8`

2 Funktionsaufrufe

- Benannte Argumente
- Default-Argumente
- Variable Argumentenliste
- Erweiterte Aufrufsyntax

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:

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

- Tatsächlich geht es aber auch anders:

```
power(base=2, exponent=10)      # 1024.
power(2, exponent=10)           # 1024.
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
...
>>> power(x=2, y=10)
Traceback (most recent call last): ...
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
...
>>> 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:

```
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:

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


Achtung bei veränderlichen Default-Argumenten (1)

- 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) # enspricht egg += [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 und beliebig viele benannte Argumente unterstützen.
- Die Idee: 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, wobei `*spam` vor `**egg` stehen

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')]
```


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 Argumente eine `1` gefolgt von den Elementen aus `argtuple` und als benannte Argumente `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 Aufrufkonvention 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, end=' ')
  return result
```

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

3 Ausnahmebehandlung

- Ausnahmen
- `try-except`-Blöcke
- `try-except-else`-Blöcke
- `try-finally`-Blöcke
- Verwendung von Ausnahmen
- Ausnahmehierarchie
- `raise`-Anweisung
- `assert`-Anweisung

Ausnahmen (1)

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

Python-Interpreter

```
>>> print({"spam": "egg"}["parrot"])
Traceback (most recent call last): ...
KeyError: 'parrot'
```

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

Ausnahmen (2)

- Ebenso wie viele andere moderne Sprachen 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`.

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 `XYError` 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 Ausnahmetypen gemeinsam behandeln, indem man diese 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. Vorsicht: Es werden dann auch CTRL-C-Ausnahmen abgefangen! Besser ist, den Ausnahmetyp `Exception` in dem Fall zu benutzen.

except-Spezifikationen (2)

- `except`-Blöcke werden 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.
- 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 (kommt gleich).

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 die try-finally-Konstruktion:

```
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.

Behandlung des finally-Blocks weitergegeben.

Beispiel

kaboom.py

```
def kaboom(x, y):
    print(x + y)

def tryout():
    kaboom("abc", [1, 2])

try:
    tryout()
except TypeError as e:
    print("Hello world", e)
else:
    print("All OK")
finally:
    print("Cleaning up")
print("Resuming ...")
```


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 for forgiveness than permission.**‘
- Der Gegensatz ist das LBYL-Prinzip: *Look before you leap*, d.h. teste Vorbedingung, bevor eine Operation durchgeführt wird (in Sprachen wir C).

Beispiele: EAFP und LBYL:

EAFP

```
try:
    x = my_dict["key"]
except KeyError:
    # handle missing key
```

LBYL

```
if "key" in my_dict:
    x = my_dict["key"]
else:
    # handle missing key
```

Ausnahmehierarchie

- Pythons enthält eine große Zahl an Ausnahmetypen. Ein Überblick findet sich hier: `http://docs.python.org/3.4/library/exceptions.html`

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      .
      .
```


Eigene Ausnahmen

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

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

- `MyException` kann dann genauso verwendet werden wie eingebaute Ausnahmen, z.B. `IndexError`.
- Für `BaseClass` wird man meist `Exception` wählen, aber natürlich eignen sich auch andere Ausnahmetypen.
- Nebenbemerkung: `pass` ist die Python-Anweisung für ‚tue nichts'.

Die `raise`-Anweisung

- Mit der `raise`-Anweisung kann eine Ausnahme signalisiert (ausgelöst, geschmissen) werden.
- Dazu verwendet man `raise` zusammen mit der Angabe einer Ausnahme (beispielsweise `IndexError` oder `NameError`):

```
raise KeyError("Fehlerbeschreibung")
```

- Die Beschreibung kann auch weggelassen werden; die Form `raise KeyError()` ist also auch zulässig.
- Auch die Notation `raise KeyError` ist erlaubt.
- `raise` alleine benutzt man, wenn man in einer Ausnahme „weiter reichen" möchte.
- Mit `raise` *Exception* `from` *e* kann man eine eigene Ausnahme innerhalb einer Ausnahme signalisieren, die dann auch extra angezeigt wird.

Die `assert`-Anweisung

- Mit der `assert`-Anweisung macht man eine Zusicherung:

```
assert test [, data]
```

- Dies ist nichts anderes als eine konditionale `raise`-Anweisung:

```
if __debug__:
    if not test:
        raise AssertionError(data)
```

- `__debug__` ist eine globale Variable, die normalerweise `True` ist.
- Wird Python mit der Option `-O` gestartet, wird `__debug__` auf `False` gesetzt.

Zusammenfassung

UNI FREIBURG

- Es ist möglich, benannte Argumente beim Aufruf einer Funktion anzugeben.
- Speziell Default-Atgumenten erhalten so einen Wert.
- Variable Argumentenlisten (mit * und **) erlauben einen weiteren Freiheitsgrad bei der Angabe der Argumente.
- Auch beim Aufruf kann die * und **-Notation benutzt werden.
- Ausnahmen sind in Python allgegebenwärtig.
- Diese können mit `try`, `except`, `else` und `finally` abgefangen und behandelt werden.
- In Python verfolgt man die die EAFP-Strategie (statt LBYL), und behandelt lieber Ausnahmen als sie zu vermeiden.
- Mit `raise` und `assert` kann man eigene Ausnahmen auslösen.