Robustere Python-Programme

Chemnitzer Linuxtage 2010

Stefan Schwarzer, SSchwarzer.com
info@sschwarzer.com

Chemnitz, Germany, 2010-03-13

Überblick

- Einführung
- Einrückungen
- Objekte und Namen
- Funktionen und Methoden
- Exceptions (Ausnahmen)
- `exec` und `eval`
- `subprocess`-Modul
- `for`-Schleifen
- Zeichenketten
- Optimierung
- Werkzeuge zur Code-Analyse
- Zusammenfassung

Einführung

- Python ist eine vielseitige Sprache
- Konzentration aufs Problem statt auf die Sprache
- kompakte Problemlösungen
- einige Fehler finden sich immer wieder in Python-Programmen
- vor allem bei Einsteigern und Gelegenheitsprogrammierern
- Vortrag beschreibt (hoffentlich) die wichtigsten Konzepte, die häufigsten Fehler und deren Vermeidung
- in diesem Vortrag Python 2.x, da meist vorinstalliert

Einführung

Arbeitserleichterungen und Robustheit

- manche Hinweise fallen auf den ersten Blick eher unter „Arbeitserleichterung“ als unter Fehlervermeidung
- Arbeitserleichterungen ersparen aber oft aufwändigeren Code
- Code, der weniger aufwändig ist, ist leichter zu schreiben und zu lesen (wichtig für spätere Änderungen)
- Arbeitserleichterungen können also indirekt ebenfalls zu robusterem Code führen
- aber nur dann, wenn der Code einfacher zu verstehen und nicht nur kürzer ist

Einrückungen

Probleme vermeiden

- Anweisungsblöcke entstehen durch gleiche Einrückung der enthaltenen Anweisungen
- Einrückung besteht aus „horizontalem Leerraum“ (Leerzeichen, Tabulatorzeichen)
- theoretisch auch gemischt
- Empfehlung: genau vier Leerzeichen pro Einrückungsebene
- siehe PEP 8, `http://www.python.org/dev/peps/pep-0008/`
- wird oft von Programmier-Editoren automatisch verwendet, wenn Endung `.py` vorhanden
- falls nicht, so konfigurieren, dass bei Drücken der Tabulatortaste vier Leerzeichen eingefügt werden

Einrückungen

Mögliche Probleme

- Was kann bei Mischung von Leer- und Tabulatorzeichen schiefgehen?
- Programmfehler sind möglich
- fast immer jedoch Syntaxfehler aufgrund inkonsistenter Einrückung
- zum Beispiel muss auf ein `if` eine Einrückung folgen und vor einem `except` eine „Ausrückung“

Einrückungen

Probleme finden

- Sichtbarmachen von Leerzeichen und Tabulatorzeichen im Editor, zum Beispiel in Vim `:set list`
- `find` und `grep` verwenden:
  `find . -name "*.py" -exec grep -EnH "\t" {} \;`
- `python -tt` . . . benutzen

Identitäts-Operator

- ermittelt, ob zwei Objekte identisch sind
- mit anderen Worten, ob es sich bei beiden um ein und dasselbe Objekt handelt
- liefert in dem Fall `True` zurück, sonst `False`
- der Operator ist das Schlüsselwort `is`
- Identität ist nicht dasselbe wie Gleichheit!

```
>>> 1 == 1.0
True
>>> 1 is 1.0
False
>>> [1] == [1]
True
>>> [1] is [1]
False
```

Namen und Zuweisungen

Allgemeines

- Namen („Variablen“) enthalten in Python keine Objekte
- sie verweisen auf Objekte
- `x = 1.0` bindet den Namen `x` an das Objekt `1.0`
- in einem Ausdruck (zum Beispiel auf der rechten Seite einer Zuweisung) steht ein Name für das Objekt, auf das er verweist
- ```
  >>> x = 1.0
  >>> y = x
  >>> x is y
  True
  ```
- `x` und `y` verweisen nun auf dasselbe Objekt
- in der zweiten Zuweisung wird das Objekt `1.0` nicht kopiert (sonst hätten wir zwei Objekte)
  ```

Namen und Zuweisungen

Unveränderbare und veränderbare Objekte

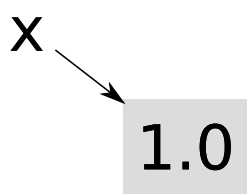
- unveränderbare Objekte haben meist einfache Datentypen; Beispiele: `7.0`, `"abc"`, `True`
- bei veränderbaren Objekten handelt es sich um zusammengesetzte Daten, zum Beispiel Listen oder Dictionaries

```
>>> L = []
>>> L.append(2)
>>> L
[2]
>>> L[0] = 3
>>> L
[3]
```

Namen und Zuweisungen

Unveränderbare Objekte

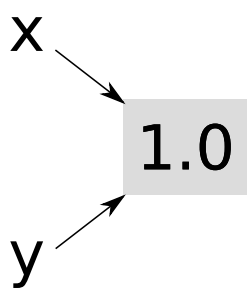
```
>>> x = 1.0
>>> y = x
>>> x is y
True
>>> y = 1.0
>>> x is y
False
```



Namen und Zuweisungen

Unveränderbare Objekte

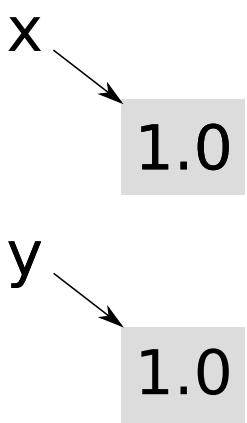
```
>>> x = 1.0
>>> y = x
>>> x is y
True
>>> y = 1.0
>>> x is y
False
```



Namen und Zuweisungen

Unveränderbare Objekte

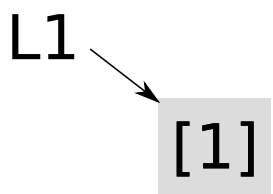
```
>>> x = 1.0
>>> y = x
>>> x is y
True
>>> y = 1.0
>>> x is y
False
```



Namen und Zuweisungen

Veränderbare Objekte

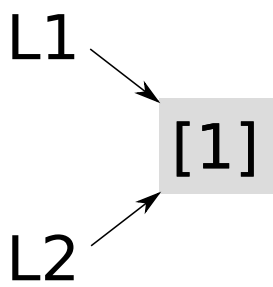
```
>>> L1 = [1]
>>> L2 = L1
>>> L1.append(2)
>>> L1
[1, 2]
>>> L2
[1, 2]
>>> L2 = [5, 6]
>>> L1.append(3)
>>> L1
[1, 2, 3]
>>> L2
[5, 6]
```



Namen und Zuweisungen

Veränderbare Objekte

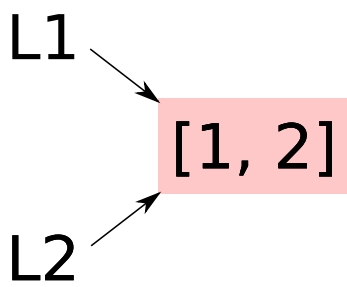
```
>>> L1 = [1]
>>> L2 = L1
>>> L1.append(2)
>>> L1
[1, 2]
>>> L2
[1, 2]
>>> L2 = [5, 6]
>>> L1.append(3)
>>> L1
[1, 2, 3]
>>> L2
[5, 6]
```



Namen und Zuweisungen

Veränderbare Objekte

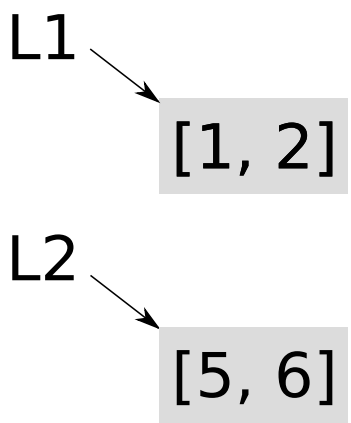
```
>>> L1 = [1]
>>> L2 = L1
>>> L1.append(2)
>>> L1
[1, 2]
>>> L2
[1, 2]
>>> L2 = [5, 6]
>>> L1.append(3)
>>> L1
[1, 2, 3]
>>> L2
[5, 6]
```



Namen und Zuweisungen

Veränderbare Objekte

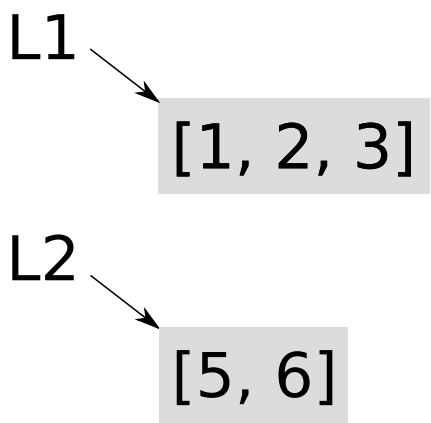
```
>>> L1 = [1]
>>> L2 = L1
>>> L1.append(2)
>>> L1
[1, 2]
>>> L2
[1, 2]
>>> L2 = [5, 6]
>>> L1.append(3)
>>> L1
[1, 2, 3]
>>> L2
[5, 6]
```



Namen und Zuweisungen

Veränderbare Objekte

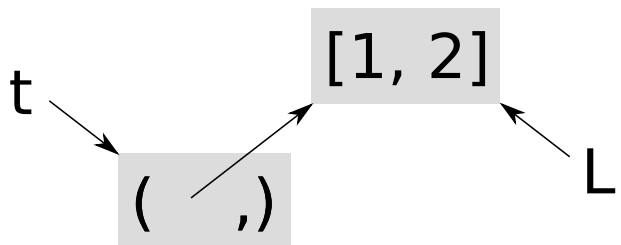
```
>>> L1 = [1]
>>> L2 = L1
>>> L1.append(2)
>>> L1
[1, 2]
>>> L2
[1, 2]
>>> L2 = [5, 6]
>>> L1.append(3)
>>> L1
[1, 2, 3]
>>> L2
[5, 6]
```



Namen und Zuweisungen

Kombination unveränderbarer und veränderbarer Objekte

```
>>> L = [1]
>>> t = (L,)
>>> t.append(2)
Traceback (most recent call last):
  File "<ipython console>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
>>> L.append(2)
>>> t
([1, 2],)
```



Dynamische Typisierung/Bindung

- Python wird oft als Sprache mit dynamischer Typisierung bezeichnet
- Objekte ändern aber in Python nicht ihren Typ
- jedoch kann ein Name auf Objekte unterschiedlicher Typen verweisen
- „dynamische Bindung“ statt „dynamische Typisierung“
- `a = "abc"; a = []` ist gültiger Python-Code
- relativ selten verwendet, am ehesten noch Objekte eines einzigen Typs vs. `None` (um anzuzeigen, dass kein Wert vorliegt, zum Beispiel bei Default-Argumenten)
- kaum automatische Typ-Umwandlungen für bestimmte Operationen (vor allem numerische Berechnungen; `1 + 1.0` ist gültiger Code, `1 + "2"` erzeugt einen `TypeError`)

Vergleiche

`is None` vs. `== None`

- `is` ermittelt Identität, `==` Wert-Gleichheit
- Empfehlung: `wert is None`
- Grund: Klassen können das Ergebnis von Vergleichen ändern

```
>>> class ImmerGleich(object):
...     def __eq__(self, operand2):
...         return True
>>> ig = ImmerGleich()
>>> ig == None
True
>>> None == ig
True
>>> ig is None
False
```

Vergleiche

„Wahrheit“ und „Falschheit“

- für die eingebauten Datentypen gilt: falsch sind numerische Null-Werte (bspw. `0.0`), leere Strings (`""`, `u""`) und leere Container (`[]`, `()`, `{}`, `set()`, `frozenset()`), sowie `None` und `False`, andere eingebaute Objekte sind wahr
- folglich können viele `if`-Bedingungen vereinfacht werden:

```
if wert == True       →  if wert
if liste != []        →  if liste
if liste == []        →  if not liste
if len(liste) == 0    →  if not liste
if string == u""      →  if not string
```

und so weiter ...

Vergleiche

`if liste` etc.

- Was ist so toll an `if liste` etc.? ;-)
- kürzer
- aber auch verständlicher (robuster)?
- ja – mit anderer Lesweise
- nicht „sind Werte in der Liste?“, sondern „gibt's . . . ?“
- Beispiel:

```
def zeige_namensliste(namen):
    if namen:
        print "\n".join(namen)
    else:
        print "keine Namen in der Liste"
```

Funktionen und Methoden

Funktion vs. Aufruf

- die Verwendung einer Funktion (oder Methode) ohne Klammern (Aufruf) liefert nur das Funktionsobjekt
- ```
  fobj = open(dateiname, 'rb')
  # erste 100 Bytes lesen
  data = fobj.read(100)
  fobj.close()  # aufrufen!
  ```
  ```

Funktionen und Methoden

Default-Argumente

- Default-Argumente werden während der Definition ausgewertet
- nicht während jeder Ausführung
-
```
>>> def anhaengen(obj, L=[]):
...     L.append(obj)
...     return L
...
>>> anhaengen(2)
[2]
>>> anhaengen(5)
[2, 5]
```

Funktionen und Methoden

Namen im Aufruf

- beim Aufruf von Funktionen und Methoden können die Argument-Namen explizit angegeben werden
- damit andere Reihenfolge als in der Definition möglich
- folgende Aufrufe sind gleichwertig:

```
>>> def f(a, b, c):
...     return [a, b, c]
...
>>> f(1, 2, 3)
[1, 2, 3]
>>> f(a=1, b=2, c=3)
[1, 2, 3]
>>> f(b=2, c=3, a=1)
[1, 2, 3]
```

Funktionen und Methoden

Argumente „durchreichen“

- „Durchreichen“ von Argumenten von einer Funktion zur anderen ist manchmal nützlich
-
```
>>> def f(a, b, c):
...     print a, b, c
...
>>> def g(*args, **kwargs):
...     print "Positions-Argumente:", args
...     print "Schluesselwort-Argumente:", kwargs
...     f(*args, **kwargs)
...
>>> g(1, c=3, b=2)
Positions-Argumente: (1,)
Schluesselwort-Argumente: {'c': 3, 'b': 2}
1 2 3
```

Funktionen und Methoden

Argumentübergabe durch Namensbindung

- Argumentübergabe funktioniert wie Zuweisung
- Name wird mit Objekt verknüpft
-
```
>>> def liste_loeschen(liste):
...     "Alle Elemente aus der Liste entfernen."
...     liste = []  # neuer lokaler Name
...
>>> eine_liste = [1, 2, 3]
>>> liste_loeschen(eine_liste)
>>> eine_liste
[1, 2, 3]  # keine Aenderung!
```

Funktionen und Methoden

Argumentübergabe durch Namensbindung

- Argumentübergabe funktioniert wie Zuweisung
- Name wird mit Objekt verknüpft
-
```
>>> def liste_loeschen(liste):
...     "Alle Elemente aus der Liste entfernen."
...     liste[:] = []  # Aenderung des Arguments
...
>>> eine_liste = [1, 2, 3]
>>> liste_loeschen(eine_liste)
>>> eine_liste
[]  # geaendert
```

Exceptions

Warum Exceptions?

Fehlerbehandlung in manchen Sprachen (Shell, C, . . .)
mit Fehlercodes

Mögliche Probleme bei Verwendung von Fehlercodes:

- Fehlerbehandlung macht Rückgabewerte und damit deren Handhabung komplexer (bspw. Tupel statt Skalar)
- Kontrolle der Fehlercodes muss unter Umständen durch eine umfangreiche Aufrufhierarchie hinweg erfolgen
- wird eine Kontrolle vergessen, gibt es undefinierte (meist zunächst „unsichtbare“) Folgen

Exceptions

Leerer except-Zweig

- ```
  try:
      # tu was ...
  except:
      # Fehlerbehandlung
  ```
- Problem: manche Exceptions werden ungewollt abgefangen (`NameError`, `AttributeError`, `IndexError`, ...)
- damit werden leicht eigene Programmierfehler verschleiert
- ```
  try:
      fobj = opne("evtl_nicht_da")
      ...
  except:
      print "Datei nicht vorhanden!"
  ```
- Liste von Exception-Klassen unter `http://docs.python.org/library/exceptions.html`

Exceptions

Leerer `except`-Zweig

- es gibt Fälle, in denen ein leerer `except`-Zweig in Ordnung ist
- Beispiel: Serverprozesse, die „fremden“ Code ausführen
- Überlegung: ein fehlerhaft laufendes Servlet ist besser als ein komplett gestoppter Server
-
```
def fuehre_servlet_aus(servlet):
    try:
        # Server-fremder Code
        servlet(...)
    except:
        # logge Fehler und mach weiter
```

Exceptions

Zu viel Code im `try`-Zweig

```
def alter_aus_db(name):
    ...

try:
    person[name][alter] = alter_aus_db(name)
except KeyError:
    print 'Kein Datensatz für Person "%s"' % name
```

Exceptions

Zu viel Code im `try`-Zweig

```
def alter_aus_db(name):
    return cache[name]

try:
    person[name][alter] = alter_aus_db(name)
except KeyError:
    print 'Kein Datensatz für Person "%s"' % name
```

Exceptions

Zu viel Code im `try`-Zweig

```
def alter_aus_db(name):
    return cache[name]

# moegliche Exception nicht "verstecken"
db_alter = alter_aus_db(name)
try:
    person[name][alter] = db_alter
except KeyError:
    print 'Kein Datensatz für Person "%s"' % name
```

Empfehlung: Problem von vornherein vermeiden, indem man mit abstrakteren Schnittstellen und Exceptions arbeitet (bspw. `CacheFehler` statt `KeyError` erzeugen)

Exceptions

Ressourcen freigeben

- sicherstellen, dass keine Ressourcen-Lecks auftreten:

```
db_conn = connect(datenbank)
try:
    # Datenbank-Operationen
    ...
finally:
    db_conn.rollback()
    db_conn.close()
```

- ab Python 2.5 kann für Dateien und Sockets auch die `with`-Anweisung verwendet werden

```
from __future__ import with_statement # fuer Py 2.5
with open(dateiname) as fobj:
    data = fobj.read()
# Datei nach with-Anweisung automatisch geschlossen
```

Exceptions

Mehrere Ausnahmen pro `except`-Zweig

```
try:
    # kann ValueError oder IndexError ausloesen
    ...
except (ValueError, IndexError):
    # Fehlerbehandlung fuer ValueError und IndexError
    ...
```

Problem: ohne Verwendung von Klammern ist `IndexError` im Fehlerfall ein `ValueError`-Objekt

Anmerkung: in Python 3.x ist die Syntax ohne Klammern nicht mehr möglich :-)

Exceptions

Sonstiges

- `except`-Zweige immer von den spezielleren zu den allgemeineren Exception-Klassen ordnen, da der erste passende Zweig von oben „drankommt“
- keine String-Exceptions verwenden: `raise "So nicht!"`
  Python-Versionen ab 2.6 betrachten String-Exceptions zudem als Syntaxfehler

Exec und eval

Probleme

`exec` und `eval` interpretieren eine Zeichenkette als Pythoncode und führen ihn aus.

Probleme:

- unübersichtlicher Code
- Einrückungs-Fehler leichter möglich
- Syntaxkontrolle erst zur Laufzeit
- Sicherheitslücken
- eingeschränkte Code-Analyse durch Programme

Exec und eval

Unübersichtlicher Code

- ```
  def baue_addierer(offset):
      # fuer konsistente Einrueckungen sorgen
      code = """
  def addierer(n):
      return n + %s
  """ % offset
      exec code
      return addierer

  neuer_addierer = baue_addierer(3)
  print neuer_addierer(2)  # 3 + 2 = 5
  ```
- ```
  def wert_n(n):
      return eval("obj.wert%d" % n)
  ```

Exec und eval

Unübersichtlichen Code vermeiden

- Funktionen, Klassen etc. in andere Funktionen und Methoden aufnehmen

```
def baue_addierer(offset):
    def addierer(n):
        return n + offset
    return addierer

neuer_addierer = baue_addierer(3)
print neuer_addierer(2)  # 3 + 2 = 5
```

- `getattr`, `setattr` und `delattr` verwenden

```
def wert_n(n):
    return getattr(obj, "wert%d" % n)
```

(noch besser: mit Dictionary mit Schlüssel `n` arbeiten)

Exec und eval

Sicherheitslücken

- Beispiel: Funktionsplotter auf einer Internet-Seite

  Funktionsplotter

  f(x) = 2*x + 3_ | Anzeigen

- ```
  def auswertung(funktion):
      for i in xrange(-100, 101):
          x = 0.1 * i
          y = eval(funktion)
          neuer_punkt(x, y)
      zeige_funktion()
  ```

- Was passiert, wenn die eingegebene Zeichenkette
  `os.system("rm -rf *")`
  lautet?
  ```

Exec und eval

Sicherheitslücken vermeiden

- gegen gültige Werte prüfen

```
if eingabe in gueltige_werte:
    # ok
else:
    # Fehler (abweisen oder Default verwenden)
```

  wobei `gueltige_werte` bspw. eine Liste oder ein Set ist
- für Ausdrücke (siehe Beispiel Funktionsplotter) Parser verwenden
- kann sehr aufwändig sein
- fertige Parser im PyPI (Python Package Index) oder Python Recipes (ActiveState)
- diverse Bibliotheken, um eigene Parser leichter zu entwickeln (pyparsing, SimpleParse, PLY etc.); siehe `http://nedbatchelder.com/text/python-parsers.html`

Das subprocess-Modul

- das subprocess-Modul ersetzt einige Befehle aus dem os-Modul durch sicherere Varianten
- ```
  import os
  def verzeichnis(name):
      return os.system("ls -l %s" % name)
  ```
- ok für `name == "/home/schwa"`
- nicht ok für `name == "/home/schwa ; rm -rf *"`
- „Entschärfung“ solcher Zeichenketten aufwändig und fehleranfällig
- besser:

  ```
  import subprocess
  def verzeichnis(name):
      return subprocess.call(["ls", "-l", name])
  ```
- ebenfalls Ersatz für `os.popen` etc.
  ```

Schleifen

`for`-Schleifen

- ist die Sequenz in der `for`-Schleife leer, wird der Schleifenrumpf nicht durchlaufen
- direkt über Sequenzen iterieren; keine Indizes nötig

```
sprachen = (u"Python", u"Ruby", u"Perl")
for i in xrange(len(sprachen)):
    print sprachen[i]
```

Schleifen

`for`-Schleifen

- ist die Sequenz in der `for`-Schleife leer, wird der Schleifenrumpf nicht durchlaufen
- direkt über Sequenzen iterieren; keine Indizes nötig

```
sprachen = (u"Python", u"Ruby", u"Perl")
for sprache in sprachen:
    print sprache
```

Schleifen

`for`-Schleifen

- ist die Sequenz in der `for`-Schleife leer, wird der Schleifenrumpf nicht durchlaufen
- direkt über Sequenzen iterieren; keine Indizes nötig

```
sprachen = (u"Python", u"Ruby", u"Perl")
for sprache in sprachen:
    print sprache
```

- werden auch Indizes benötigt, `enumerate` verwenden

```
sprachen = (u"Python", u"Ruby", u"Perl")
for index, sprache in enumerate(sprachen):
    print u"%d: %s" % (index+1, sprache)
```

Zeichenketten

Allgemeine Hinweise

- Zeichenketten (sowohl Bytestrings als auch Unicode) sind unveränderbar
- `s.startswith(anfang)` prüft, ob der String `s` mit dem String `anfang` beginnt
- `s.endswith(ende)` prüft, ob der String `s` mit dem String `ende` endet
- `teilstring in s` prüft, ob `teilstring` im String `s` enthalten ist; `index` und erst recht `find` sind nicht nötig
- negative Indizes zählen vom Ende des Strings, Beispiel: `u"Python-Vortrag"[-4:] == u"trag"`
- hier nicht behandelt: Bytestrings vs. Unicode; Encodings (wichtige Themen, die einen eigenen Vortrag wert sind) `http://www.p-nand-q.com/python/unicode_faq.html`

Optimierung

- nicht beim Schreiben des Codes optimieren
- führt im Allgemeinen nicht zu einem schnelleren Programm
- aber zu schwerer wartbarem Code
- erst sauber entwickeln
- falls zu langsam, Profiler verwenden (`cProfile`/`profile`-Module); gezielt optimieren
- „Optimierungen“, die keine Geschwindigkeitssteigerungen bringen, zurücknehmen
- Vorgehen ausführlich beschrieben unter `http://www.linux-magazin.de/Heft-Abo/Ausgaben/2006/12/Gut-gezielt`

Werkzeuge zur Code-Analyse

- erkennen viele der hier besprochenen Probleme
- nicht narrensicher, aber sehr hilfreich :-)
- PyLint

  `http://pypi.python.org/pypi/pylint`
  `http://www.logilab.org/project/pylint`
- PyChecker

  `http://pypi.python.org/pypi/PyChecker`
  `http://pychecker.sourceforge.net/`

Zusammenfassung, Teil 1/2

- Lesbarkeit ist wichtiger als Kürze
- inkonsistente Einrückungen lassen sich leicht vermeiden
- Gleichheit ist nicht dasselbe wie Identität
- in Bedingungsausdrücken nicht explizit mit leeren Listen, Tupeln etc. vergleichen
- Default-Argumente in Funktionen werden während der Definition ausgewertet
- bei benannten Argumenten ist die Reihenfolge flexibel
- Argumente können mit `*args` und `**kwargs` gesammelt durchgereicht werden
- sollen außerhalb einer Funktion Änderungen von Argumenten sichtbar werden, müssen sie selbst verändert werden, nicht nur die Namensbindung

Zusammenfassung, Teil 2/2

- „leere“ `except`-Zweige nur in Sonderfällen verwenden
- aufpassen, was in einen `try`-Zweig soll und was nicht
- Ressourcen mit `try...finally` oder `with` freigeben
- mehrere Ausnahmen in einem `except`-Zweig einklammern
- `except`-Zweige von abgeleiteten zu deren Basisklassen ordnen
- `exec` und `eval` sollten wegen ihrer Anfälligkeit für Sicherheitslücken und andere Fehler vermieden werden
- Shell-Aufrufe nicht mit den Funktionen im `os`-, sondern mit denen im `subprocess`-Modul durchführen
- `for`-Schleifen benötigen meist keinen expliziten Zugriff auf einen Feld-Index
- nur mit Hilfe eines Profilers optimieren – und falls überhaupt, erst wenn der Code funktioniert
- PyLint und PyChecker helfen beim Erstellen von sauberem Python-Code

Danke für Ihre Aufmerksamkeit! :-)

Fragen?

Anmerkungen?

Diskussion?