Effiziente interaktive Lernmethoden: IPython & Debugger

Carsten Knoll

TU Dresden, Institut für Regelungs- und Steuerungstheorie

Interdisziplinärer Python "'Sommerkurs"'

11. April 2011

Vorbemerkungen

- Warum Vortrag über „effiziente Methoden zum Lernen von Python“?

- Möglichkeiten aufzeigen Probleme selbständig zu lösen
  - Spart Zeit und Nerven
  - Erhöht Motivation
- Aufbau:
  - Exceptions und Docstrings
  - IPython
  - Debugger
  - Häufige Fallen
  - Übungsaufgaben

Vorbemerkungen

- Warum Vortrag über „effiziente Methoden zum Lernen von Python“?

- Möglichkeiten aufzeigen Probleme selbständig zu lösen
  - Spart Zeit und Nerven
  - Erhöht Motivation
- Aufbau:
  - Exceptions und Docstrings
  - IPython
  - Debugger
  - Häufige Fallen
  - Übungsaufgaben

Exceptions

- Programmierfehler unvermeidbar
- $\exists$ Syntaxfehler, **semantische Fehler**
- Fehler in der Programmlogik führen häufig zu ungültigen Programmzuständen
- Es tritt eine Ausnahme ein

Exception (de.wikipedia, gekürzt):

Eine Ausnahme oder (engl. exception) bezeichnet ein Verfahren, Informationen über bestimmte Programmzustände – meistens Fehlerzustände – an andere Programmebenen weiterzureichen.

- Wozu ist das gut?
- Kann „weiter oben“ abgefangen werden (`try...except`)
- Liefert Hinweise **wo** ein Fehler aufgetreten ist, und **welcher Art**

Exceptions

- Programmierfehler unvermeidbar
- $\exists$ Syntaxfehler, **semantische Fehler**
- Fehler in der Programmlogik führen häufig zu ungültigen Programmzuständen
- Es tritt eine Ausnahme ein

Exception (de.wikipedia, gekürzt):

Eine Ausnahme oder (engl. exception) bezeichnet ein Verfahren, Informationen über bestimmte Programmzustände – meistens Fehlerzustände – an andere Programmebenen weiterzureichen.

- Wozu ist das gut?
- Kann „weiter oben“ abgefangen werden (`try...except`)
- Liefert Hinweise **wo** ein Fehler aufgetreten ist, und **welcher Art**

Exceptions II

Häufige Exeptions:

- `NameError` (unbekannte Variablenname)
- `ValueError`(int('text'))
- `TypeError` ('2'+5)
- `IndexError` (L = [1, 2, 3]; b = L[5])
- `ZeroDivisionError`, ..., eigene Exception-Typen

- Man kann (und sollte) eigene Exceptions „werfen“
  → `raise`, `assert`

```
def quadratwurzel(x):
    assert(x>=0)
    return x**0.5
```

Exceptions II

Häufige Exeptions:

- `NameError` (unbekannte Variablenname)
- `ValueError`(int('text'))
- `TypeError` ('2'+5)
- `IndexError` (L = [1, 2, 3]; b = L[5])
- `ZeroDivisionError`, ..., eigene Exception-Typen

- Man kann (und sollte) eigene Exceptions „werfen“
  → `raise`, `assert`

```
def quadratwurzel(x):
    assert(x>=0)
    return x**0.5
```

Docstrings

- Gute Dokumentation von Programmen sehr wichtig
- Aussagekräftige Namen für Funktionen, Variablen, ...
- Komentare schreiben und aktuell halten
- Docstrings: ähnlich wie Kommentare, aber zur Laufzeit im Programm verfügbar
- Extrem nützlich für interaktives Arbeiten

```
def quadratwurzel(x):
    """berechnet die Wurzel einer nicht negativen Zahl"""
    assert(x>=0)
    return x**0.5
```

- Können auch für automatisches Generieren von Dokumentation genutzt werden
- Außerdem: ∃ Doctests: Mischung aus Beispiel-Code und automatischen Tests

IPython (I)

- Erweiterte interaktive python Eingabeaufforderung („enhanced interactive python shell“)
- Zusatzfeatures:
  - Auto-Vervollständigung (mit TAB)
  - History (mit Suche; bleibt über mehrere Sessions erhalten)
  - Formatierung („pretty printing“)
  - Farbige Hervorhebung (bei Exceptions, ...)
  - Dynamische Objektinformationen mit `?, ??`
    Typ, Wert, ggf. Signatur, **Docstring**, mit `??`: Quellcode
  - „Magische Kommandos:“
    `%macro, %bookmark, %store`,... Hilfe: `%magic`
  - Speichern von Ein- und Ausgabe (`_, __, ___`)
  - Ausführern von Systemkommandos:
    Bsp: `!cp $fname1 $fname2`
    `cp, rm, ls`,... schon eingebaut

IPython (II)

- Wozu das alles??
- Weniger Zeit für Tippen und Suchen

$\Rightarrow$ Mehr Zeit für eigentliches Problem

- `TAB` u. `?`: sehr gut zum Erforschen unbekannter Module
- Meist schneller, als richtige Doku zu lesen
- Sehr gut auch als „Taschenrechner“ oder
  um schnell mal was auszuprobieren
- Kann in eigene Programme eingebunden werden.

Debuggen

- Detailliertes Verstehen eines Programmteils
- $\exists$ Grafischer Debugger `winpdb` (plattformunabh. trotz „win“)
- Kommandozeilendebugger `pdb` standardmäßig mit dabei
- IPyton: `pdb` += TAB-Vervollständigung + Farben
- `pdb` kann bei unbehandeldet Ausnahmen „aufwachen“
- Kommandos:

Kurz	**Lang**	**Bedeutung**	**Kurz**	**Lang**	**Bedeutung**
h	help	Hilfe anzeigen	w	where	zeigen wo man ist
n	next	nächste Anweisung	a	args	Argumente der
s	step	ggf. in eine Funktion			aktuellen Funktion
		hineinspringen	u	up	im Stack aufwärts
r	return	aus Funktion	d	down	im Stack abwärts
		zurückspringen	c	continue	Programm fortführen
p	print	Variable ausgeben	q	quit	Abbruch

Das Modul ipHelp

- Persönliches Hilfsmodul (subjektiv: sehr nützlich)

```
from ipHelp import IPS, ST, ip_syshook
```

- `IPS`: startet IPython Shell im aktuellen Namensraum verlassen mit `STRG+D`, Programm läuft weiter
- `ST` startet den pdb an der Stelle
- `ip_syshook(1)` startet pdb bei unbehandelter Ausnahme

- außerdem: `dirsearch(word, obj)`
- um Objekte mit vielen Attributen zu durchsuchen, z.B. Module

```
dirsearch('prime', sympy)
```

Das Modul ipHelp

- Persönliches Hilfsmodul (subjektiv: sehr nützlich)

```
from ipHelp import IPS, ST, ip_syshook
```

- `IPS`: startet IPython Shell im aktuellen Namensraum verlassen mit `STRG+D`, Programm läuft weiter
- `ST` startet den pdb an der Stelle
- `ip_syshook(1)` startet pdb bei unbehandelter Ausnahme

- außerdem: `dirsearch(word, obj)`
- um Objekte mit vielen Attributen zu durchsuchen, z.B. Module

```
dirsearch('prime', sympy)
```

Häufige Fallen

- Ganzzahl Division

```
for k in [4,5,6]:
    print 5 / k
```

```
> 1
> 1
> 0
```

- TABs und Leerzeichen vermischt → Konvention: 4 Leerzeichen
- Zirkuläre imports
- Verwendung von Umlauten (bzw. Nicht-ASCII-Zeichen) ohne Kodierungs-Kommentar am Dateianfang, z. B.: `# -*- coding: utf-8 -*-`
- Manche Funktionen geben nichts zurück („in place“)

```
L = [3,2,8,4,7]
print L.sort() # > None
```

- Sinnlose Vergleiche erzeugen keine Ausnahme

```
print float('inf') < [] # > True
print 4 > {} # > False
1+ 5j > 3 - 7j # > TypeError (So muesste es immer sein)
```

- Vergessenes `self` bei Methoden-Deklaration

Häufige Fallen

- Ganzzahl Division

```
for k in [4,5,6]:
    print 5 / k
```

```
> 1
> 1
> 0
```

- TABs und Leerzeichen vermischt → Konvention: 4 Leerzeichen
- Zirkuläre imports
- Verwendung von Umlauten (bzw. Nicht-ASCII-Zeichen) ohne Kodierungs-Kommentar am Dateianfang, z. B.: `# -*- coding: utf-8 -*-`
- Manche Funktionen geben nichts zurück („in place“)

```
L = [3,2,8,4,7]
print L.sort() # > None
```

- Sinnlose Vergleiche erzeugen keine Ausnahme

```
print float('inf') < [] # > True
print 4 > {} # > False
1+ 5j > 3 - 7j # > TypeError (So muesste es immer sein)
```

- Vergessenes `self` bei Methoden-Deklaration

Häufige Fallen

- Ganzzahl Division

```
for k in [4,5,6]:
    print 5 / k
```

```
> 1
> 1
> 0
```

- TABs und Leerzeichen vermischt → Konvention: 4 Leerzeichen
- Zirkuläre imports
- Verwendung von Umlauten (bzw. Nicht-ASCII-Zeichen) ohne Kodierungs-Kommentar am Dateianfang, z. B.: `# -*- coding: utf-8 -*-`
- Manche Funktionen geben nichts zurück („in place“)

```
L = [3,2,8,4,7]
print L.sort() # > None
```

- Sinnlose Vergleiche erzeugen keine Ausnahme

```
print float('inf') < [] # > True
print 4 > {} # > False
1+ 5j > 3 - 7j # > TypeError (So muesste es immer sein)
```

- Vergessenes `self` bei Methoden-Deklaration

Häufige Fallen

- Ganzzahl Division

```
for k in [4,5,6]:
    print 5 / k
```

```
> 1
> 1
> 0
```

- TABs und Leerzeichen vermischt → Konvention: 4 Leerzeichen
- Zirkuläre imports
- Verwendung von Umlauten (bzw. Nicht-ASCII-Zeichen) ohne Kodierungs-Kommentar am Dateianfang, z. B.: `# -*- coding: utf-8 -*-`
- Manche Funktionen geben nichts zurück („in place“)

```
L = [3,2,8,4,7]
print L.sort() # > None
```

- Sinnlose Vergleiche erzeugen keine Ausnahme

```
print float('inf') < [] # > True
print 4 > {} # > False
1+ 5j > 3 - 7j # > TypeError (So muesste es immer sein)
```

- Vergessenes `self` bei Methoden-Deklaration

Häufige Fallen II

- globale und lokale Variablen (Namensräume)

```
txt = 'globaler Text'
def func(x):
    print txt, x # geht

func(5)
```

```
txt = 'globaler Text'
def func(x):
  txt = 'lokaler Text' # geht
  print txt, x
func(5)
```

```
txt = 'globaler Text'
def func(x):
  print txt, x
  txt = 'lokaler Text' # geht nicht

func(5)
```

```
txt = 'globaler Text'
def func(x):
  global txt
  print txt, x
  txt = 'lokaler Text' #geht
func(5)
```

- Mehrfachreferenzierung von Listen

```
a = [1, 2, 3]
b = a # besser: b = a[:]
a.append(0)
print b # > [1,2,3,0]
```

Häufige Fallen II

- globale und lokale Variablen (Namensräume)

```
txt = 'globaler Text'
def func(x):
    print txt, x # geht

func(5)
```

```
txt = 'globaler Text'
def func(x):
  txt = 'lokaler Text' # geht
  print txt, x
func(5)
```

```
txt = 'globaler Text'
def func(x):
  print txt, x
  txt = 'lokaler Text' # geht nicht

func(5)
```

```
txt = 'globaler Text'
def func(x):
  global txt
  print txt, x
  txt = 'lokaler Text' #geht
func(5)
```

- Mehrfachreferenzierung von Listen

```
a = [1, 2, 3]
b = a # besser: b = a[:]
a.append(0)
print b # > [1,2,3,0]
```

Gliederung

Abschließende Tipps

- Mischung aus interaktivem und Datei-basiertem Arbeiten
- Immer eine laufende IPython session
  (Taschenrechner, Docstrings lesen, kurz was ausprobieren)
- Skript aus einem Konsole-Fenster aufrufen
  (um Fehlermeldungen nicht zu verpassen)
- Lesen von fremden Code ($\rightarrow$ google codesearch)
- Das Rad nicht zu oft neu erfinden

Vorschläge für Übungsaufgaben

Ziele:

- Umgang mit den Datentypen `str, list, dict, ...`
- Nutzen von Docstrings und interaktivem Ausprobieren

1. Worte in einem Text zählen. (Tipp_1: Text: Multiline-String mit beliebigem Inhalt aus www, Tipp_2: `str.split?`)
2. Die 10 häufigsten Worte aus einem Text herausfinden (Tipp: ein `dict` hilft)
3. Die Summe aller Zahlen in einer Liste bilden
   (schwerer, wenn nicht ausschließlich Zahlen in der Liste sind)
4. In einem Dictionary Schlüssel und Werte vertauschen (Tipp: `dict.<TAB>`)
5. Duplikate in einer Liste finden
6. Gemeinsame Elemente in zwei Listen finden
7. Eine Liste von Worten alphabetisch sortieren, aber mit dem letzten Buchstaben beginnend (Tipp: `list.sort?`)
8. Strings Ver- und Entschlüsseln durch Buchstaben weiterschieben.
   Bsp: 'hallo welt' -> 'ibmmp xfmu' (Tipp: `ord`?, `chr`?)