Debugging mit Python

PyCon DE 2012

Stefan Schwarzer, SSchwarzer.com
info@sschwarzer.com

Leipzig, Deutschland, 2012-10-31

Überblick

- Print-Anweisungen und Logging
- Debugger-Konzepte
- Umsetzung mit `pdb`
- Weitere Python-Debugger
- Zusammenfassung

Print-Anweisungen

- Oft ausreichend
- Einfach einsetzbar
- Vorteile gegenüber Debugger:
  - kann einfacher bei Code mit Timeouts eingesetzt werden
  - praktisch, um zu zeigen, bei welchen Werten (zum Beispiel in einer Schleife) ein Problem auftritt

Print-Anweisungen

Wie man es eher nicht machen sollte

```
print "====================="
print self.employer.name()
...
print "*********************"
print self.employee.name()
print self.employee.address()
```

- Man kann leicht vergessen, welche Markierung zu welchem Wert gehört.
- Entfernung von Debugging-Ausgaben ist fehleranfällig, sobald einer Markierung mehr als eine Ausgabe folgt. Man kann allzu leicht etwas entfernen, was nicht für das Debugging gedacht war.

Print-Anweisungen

Empfehlung

```
print "=== employer name:", self.employer.name()
...
print "=== employee name:", self.employee.name()
print "=== employee address:", self.employee.address()
```

- Man sieht in der Ausgabe, was ausgegeben wird.
- Debug-Ausgaben sind leicht zu entfernen
  (im Editor nach „`===` “ suchen und Zeile löschen).
- Abtrennung mit Komma vermeidet Tupel-Anomalie:
  `"=== Wert: %s" % value` versagt bei Tupel-Werten.
- Schreibarbeit kann durch Editor-Makro reduziert werden.
  Beispiel für Vim:

  ```
  inoremap <leader>p print "=== :", <ESC><BS><BS><BS>i
  ```

  `\p` gibt `print "=== :",` aus und setzt den Cursor vor den Doppelpunkt.

Logging

```
import logging
logging.basicConfig()

log = logging.getLogger("debugging")
log.setLevel(logging.DEBUG)

log.debug("Here's a tuple: %s", (1, 2))
# DEBUG:debugging:Here's a tuple: (1, 2)
```

- Aufwändiger „einzurichten“ als Print-Anweisungen.
- „Geschwätzigkeit“ kann durch Log-Level gesteuert werden.
- Log-Ausgaben können gegebenenfalls im Code bleiben, wenn Log-Level und/oder Ausgabeziel angepasst werden.
- Ausgabe sehr flexibel konfigurierbar
- Threadsichere Ausgabe

Debugger

Konzepte

- Zwei Modi
  - Normaler Programmablauf
  - Eingabe von Debugging-Kommandos
- Programmzeiger
  Dieser markiert die als nächstes auszuführende Anweisung.
- Ausführung
  - ohne sichtbare Verzweigung in Aufrufe („next“)
  - mit sichtbarer Verzweigung in Aufrufe („step“)
- Unterbrechungspunkte/Breakpoints
  An diesen Anweisungen hält die Code-Ausführung an und der Debugger-Prompt erscheint.

Debugger pdb (Python Debugger)

- In der Standardbibliothek enthalten
- Nur Kommandozeilen-Modus (ähnlich gdb)
- Für viele Zwecke ausreichend
- `help` liefert eine Befehlsliste,
  `help` *`befehl`* einen Hilfetext zum Befehl *befehl*.
- Lesetipp: http://www.doughellmann.com/PyMOTW/pdb/

Debugger pdb

Aktivierung

- Innerhalb des Programmcodes

  ```
  import pdb; pdb.set_trace()
  ```

- In der Shell

  ```
  $ python -m pdb programm argumente
  ```

  Danach Breakpoint setzen und fortsetzen (kommt später).

- Post-mortem (im Vortrag nicht weiter behandelt)

Code-Beispiel

```
import os

if __name__ == "__main__":
    import pdb; pdb.set_trace()
    items = os.listdir(os.curdir)
    path = os.path.join(os.getcwd(), items[0])
    print path
```

Debugger pdb

Anzeigen von Code (list)

- `list`, `l`
  Code um den Programmzeiger herum auflisten;
  nach einem vorherigen `list`-Befehl das Listing fortsetzen
- `list` *`zeile`*
  Code um die genannte Programmzeile listen
- `list` *`anfangszeile endzeile`*
  Zeilenbereich listen

Debugger pdb

Anzeigen von Code

```
$ python debug_example.py
> /home/someone/pycon_de/debug_example.py(5)<module>()
-> items = os.listdir(os.curdir)
(Pdb) l
  1     import os
  2
  3     if __name__ == "__main__":
  4         import pdb; pdb.set_trace()
  5  ->     items = os.listdir(os.curdir)
  6         path = os.path.join(os.getcwd(), items[0])
  7         print path
[EOF]
```

Debugger pdb

Ausführen von Anweisungen

- `next`, `n`
  Anweisung einschließlich enthaltener Aufrufe ohne zwischenzeitliche Debugger-Kontrolle ausführen
- `step`, `s`
  Anweisung ausführen, dabei auch enthaltene Aufrufe unter Debugger-Kontrolle ausführen.
  In C-Funktionen – wie `os.getcwd()` im Beispiel – kann mit `s` nicht hineingesprungen werden.

Debugger pdb

Ausführen von Anweisungen

Nach dem ersten `l` von oben:

```
(Pdb) n
> /home/someone/pycon_de/debug_example.py(6)<module>()
-> path = os.path.join(os.getcwd(), items[0])
(Pdb) l
  1     import os
  2
  3     if __name__ == "__main__":
  4         import pdb; pdb.set_trace()
  5         items = os.listdir(os.curdir)
  6  ->     path = os.path.join(os.getcwd(), items[0])
  7         print path
[EOF]
```

Debugger pdb

Ausführen von Anweisungen

```
(Pdb) s
--Call--
> /usr/lib/python2.7/posixpath.py(60)join()
-> def join(a, *p):
(Pdb) l
 55
 56     # Join pathnames.
 57     # Ignore the previous parts if a part is absolute.
 58     # Insert a '/' unless the first part is empty or already
 59
 60  -> def join(a, *p):
 61         "Join two or more pathname components, inserting '/'
 62         If any component is an absolute path, all previous p
 63         will be discarded."
 64         path = a
 65         for b in p:
```

Debugger pdb

Werte anzeigen

- `p` *`ausdruck`*
  Ausdruck evaluiert anzeigen
- `pp` *`ausdruck`*
  Dito, aber übersichtlicher formatiert
  (wie von `pprint.pprint`)
- `args`
  Aktuelle Werte der Argumente der Funktion/Methode anzeigen

Debugger pdb

Aus Funktion zurückkehren

- `return`, `r`
  So lange nicht-interaktiv Code ausführen, bis die aktuelle Funktion/Methode verlassen wird und dann wieder in den interaktiven Modus schalten.

Debugger pdb

Unterbrechungspunkte/Breakpoints

- Markierung, an der der Code zur Inspektion im Debugger angehalten werden kann
- Ob das tatsächlich passiert, kann an eine Bedingung geknüpft werden.
- Ein Breakpoint kann aus- und eingeschaltet werden. Dabei bleiben die Eigenschaften (Position, Bedingungen) erhalten.
- Es lassen sich beliebig viele Unterbrechungspunkte definieren.

Debugger pdb

Unterbrechungspunkte/Breakpoints

- `break`, `b`
  Alle Breakpoints auflisten
- `break` *`dateipfad:zeilennummer, bedingung`*
  Breakpoint setzen, an dem nur unter der Bedingung *bedingung* angehalten wird. Die Bedingung wird nicht in Anführungszeichen eingeschlossen.
- Die Pfadangabe ist relativ zu `sys.path`. Obwohl der Dateipfad wie eine Modul-Angabe wirkt, darf kein Punkt als Trennzeichen verwendet werden. Stattdessen muss es ein Pfadtrenner sein, zum Beispiel `break paket/modul.py:23`
- Dateipfad und Bedingung sind optional.

Debugger pdb

Unterbrechungspunkte/Breakpoints

- `condition` *`breakpointnr bedingung`*
  Setze oder ändere die Bedingung des Breakpoints mit der Nummer *breakpointnr*.
- `continue`, `cont`, `c`
  Setze den Programmablauf fort. Dabei wird der interaktive Debugger-Modus bis zum nächsten aktiven Breakpoint oder bis zum Programmende verlassen.

Alternative Debugger für Python

- Liste (Suche im Python Package Index)
  http://pypi.python.org/pypi?%3Aaction=search&term=debugger
- Textmodus
  - pudb (nur Posix, einschließlich Cygwin)
    http://pypi.python.org/pypi/pudb
  - pdb++
    http://pypi.python.org/pypi/pdbpp
  - ipdb
    http://pypi.python.org/pypi/ipdb
- GUI
  - IDLE
    Einfache IDE; gehört zur Standardbibliothek, muss aber unter Unix/Linux eventuell zusätzlich installiert werden
  - WinPdb – nicht nur für Windows!
    http://winpdb.org/
  - Debugger in diversen Python-IDEs

Zusammenfassung

- Print-Anweisungen können hilfreich sein. Man muss nicht unbedingt gleich zum Debugger greifen.
- Das `logging`-Modul ist flexibler als `print`, aber für einfache Aufgaben etwas umständlicher.
- Debugging-Konzepte: Programmzeiger; Ausführung ohne/mit Unterprogrammaufrufen; Unterbrechungspunkte
- Code kann mit `l` gelistet und mit `n`, `s` und `r` abgearbeitet werden. `p`, `pp` und `args` zeigen Objekte an.
- Unterbrechungspunkte lassen sich mit `b` setzen; nach deren Auslösung führt `c` den Code weiter aus.
- Es gibt diverse weitere Textmodus- und GUI-Debugger für Python. Die meisten davon finden sich im Python Package Index (PyPI).

Danke für die Aufmerksamkeit! :-)

Fragen?

Anmerkungen?

Diskussion?

sschwarzer@sschwarzer.com

http://sschwarzer.com