

2 Doctest

Mache die Dinge so einfach
wie möglich - aber nicht
einfacher.

(Albert Einstein)

In diesem Kapitel wird die Testmethode *doctest* beschrieben. Für die Anregungen zu den Beispielen möchte ich mich bei Doug Hellmann [Hel15] für sein Blog „Python-Module of the Week“ bedanken.

2.1 Definition: Docstring

In der Sprache Python sind verschiedene Arten bekannt, Zeichenketten (Strings) zu bilden: Mit einfachen, mit doppelten und mit dreifach einfachen oder doppelten Anführungszeichen:

```
'Zeichenkette'  
"Zeichenkette"  
'''Zeichenkette'''  
"""Zeichenkette"""
```

Abb. 2.1: Zeichenketten in Varianten

In Python3 sind diese Strings immer als utf-8 Strings zu verstehen, die Schreibweise mit dem vorangestellten „u“ (u“string“) ist nicht länger notwendig. In der dritten und vierten Variante können Zeilenumbrüche enthalten sein, diese wird als Docstring bezeichnet:

```
"""  
Hier ist ein Beispiel  
für eine mehrzeilige Zeichenkette, diese hat  
vier Zeilenumbrüche und drei Textzeilen.  
"""
```

Abb. 2.2: Zeichenketten mit mehreren Zeilen

2 Doctest

Python Enhancement Proposal PEP 257 von 2001 beschreibt die korrekte Formatierung. Danach ist die zuletzt gezeigte Schreibweise nicht erwünscht, Die erste Zeile sollte eine Zusammenfassung enthalten und eine Leerzeile sollte folgen. Also besser so:

```
"""Hier ist ein PEP 257-konformes Beispiel.  
  
Dies ist eine längere Zeichenkette  
mit vier Zeilenumbrüchen und drei Textzeilen.  
"""
```

Abb. 2.3: Zeichenketten mit mehreren Zeilen nach PEP 257

2.2 Einfaches Beispiel

Jedes Python-Programm sollte mit Dokumentation versehen sein, diese sollte die Funktion erklären, also weitere Informationen dem Quelltext hinzufügen. Es können auch gleich einfache Tests für Funktionen enthalten sein, um den Gebrauch des Codes zu erläutern. Jeder Dokumentationsstring, der typischerweise durch drei doppelte Anführungszeichen eingerahmt ist, kann erläuternder Text, aber genauso gut auch ausführbare Programmzeilen enthalten. So kann in der Dokumentation gleich an der richtigen Stelle ein Test hinterlegt werden. Er ist als eingerückter Text geschrieben, welcher syntaktisch korrekt in Python ausführbar ist.

Der nachfolgende Text wird in der Datei bsp1.txt abgelegt.

```
# ein Objekt wird als Dictionary definiert, ein Wert zugewiesen  
>>> a = {}  
>>> a['color'] = 'blue'  
  
# Zeige das Objekt  
>>> a  
{'color': 'blue'}
```

Abb. 2.4: Datei mit doctest Zeilen

Ein paar Worte zur Erläuterung: Der Text wird dem Python-Interpreter übergeben, dieser arbeitet Zeile für Zeile ab. Die Einrückung und die einleitenden Zeichen bestimmen, wie der Interpreter mit dem Rest der Zeile umgeht.

- Nicht eingerückte Zeile wird als Kommentar überlesen [1, 5,]
- Zeile mit `____>>>` wird als Statement ausgeführt [2, 3, 6,]

- Zeile mit `____` . . . wird als Folgestatement angehängt []
- Zeile mit `____` wird mit dem letzten Ergebnis verglichen [7,]
- Leerzeile initiiert die Ausführung vorheriger Zeile(n) [4,]

Das Zeichen `_` steht in diesem erläuternden Text jeweils anstelle eines Leerzeichens. In Listings wird es nicht dargestellt und wird nur durch die Einrückung sichtbar.

Ein erster Versuch ohne weitere Argumente ist wenig spektakulär. Aus dem Fehlen jeglicher Ausgabe ist auf Fehlerfreiheit zu schließen.

```
$ python -m doctest bsp1.txt
$
```

Abb. 2.5: Doctest ohne Fehler

Wollen wir den Interpreter bei der Arbeit beobachten, empfiehlt es sich, den Plaudermodus mit `-v` auf der Kommandozeile einzuschalten. Dann lässt der Interpreter zeilenweise den Inhalt der Textdatei nachvollziehen und hilft damit, Fehler zu analysieren.

```
$ python -m doctest -v bsp1.txt
Trying:
    a = {}
Expecting nothing
ok
Trying:
    a['color'] = 'blue'
Expecting nothing
ok
Trying:
    a
Expecting:
    {'color': 'blue'}
ok
1 items passed all tests:
  3 tests in README.txt
3 tests in 1 items.
3 passed and 0 failed.
Test passed.
$
```

Abb. 2.6: Doctest geschwätzig ohne Fehler

2.3 Der Interpreter

Getreu dem Motto *batteries included* bringt jede Python-Installation ein Modul namens `doctest` mit. Um seine Funktionsweise zu verstehen, reicht zunächst ein Blick auf den Python-Interpreter.

2 Doctest

```
$ python3
Python 3.4.2 (default, Oct 8 2014, 10:45:20)
[GCC 4.9.1] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Abb. 2.7: Der Interpreter startet

So aufgerufen arbeitet der Interpreter interaktiv. Nach seinem Aufruf, hier mit `python3` in einer Linux-Kommandozeile, erscheint Version und Kompilierungsdatum, ein Hinweis zu Hilfestellung, Urheberrechten, Urhebern und Lizenz. Danach erscheint das typische Python-Prompt `>>>` als Aufforderung, etwas über die Tastatur einzugeben. Ein Zeilenende signalisiert dem Interpreter, den bis dahin erkannten Text auszuwerten, evtl. aber auch weitere Zeilen anzunehmen.

```
>>> a = 23
>>> a
23
>>> b = 2 * a
>>> b
46
```

1
2
3
4
5
6

Abb. 2.8: Der Interpreter verarbeitet sofort

Dem Objekt `'a'` wird ein Wert zugewiesen, hier die Zahl `'23'`. Bei dieser Operation gibt der Interpreter nur ein neues Python-Prompt aus, an dem dann der Wert des Objektes erfragt wird durch Eingabe seines Namens, hier `'a'`. Die Antwort erfolgt unmittelbar: `'23'`. Dieser interaktive Modus dient häufig dazu, einfach und schnell Dinge auszuprobieren oder nur eine Formulierung zu finden.

2.4 Eine Python-Datei

Es ist stets eine gute Idee, auch kurze, nur zu Testzwecken eingegebene Programmzeilen jeweils in Dateien abzulegen. Denn auch während des Schaffensprozesses ist Dokumentation wichtig. Jahre später weiß niemand mehr, wie ein Ergebnis zustande kam, wenn nur noch das Ergebnis vorhanden ist. Und es schafft Reproduzierbarkeit, die auch später noch nachvollzogen und wichtig werden kann.

Eine erste Python-Datei `bsp1.py` kann beispielsweise wie soeben gezeigt aussehen. Die Dateierendung `'py'` ist üblich, der Python-Interpreter versteht

damit auch die Zeichenketten, hier den Docstring. Eine andere Dateieindung führt dazu, die Datei nicht als Python-Code zu interpretieren, so dass die dreifachen doppelten Anführungszeichen nicht als Beginn und Ende von Docstrings erkannt werden. Fehlen diese, kann der Test dennoch korrekt ausgeführt werden. Daher können derartige Tests in einer Datei `Readme.txt` ebenso funktionieren. Hier ist aber nur die Rede von Python-Dateien, in denen Doctests an beliebiger Stelle in Docstrings auftauchen können.

```
$ cat bsp1.py
"""doctest Datei bsp1.py
    >>> a = 23
    >>> a
    23

    >>> b = 2 * a
    >>> b
    46
"""
$
```

Abb. 2.9: Eine Datei mit Doctests

Eine sinnvolle Funktion ist mit dieser Datei nicht zu erreichen, aber immerhin meckert Python beim Aufruf mit dem Modul `doctest` nichts an.

```
$ python3 -m doctest bsp1.py
$
```

Abb. 2.10: Der Interpreter verarbeitet eine Datei mit Doctests

Hier ist `python3` der Name des Interpreters, `-m doctest` fordert auf, das `doctest` Modul zu laden. Der Aufruf in der Kommandozeile zeigt keine erkennbare Wirkung. Dies ist jedoch keine schlechte Botschaft, alles ist gut verlaufen. Möchte man Informationen zu den Abläufen innerhalb des Interpreters, so ist wieder zusätzliche Parameter `-v` nützlich:

```
$ python3 -m doctest bsp1.py -v
Trying:
  a = 23
Expecting nothing
ok
Trying:
  a
Expecting:
  23
ok
Trying:
  b = 2 * a
Expecting nothing
ok
Trying:
```

2 Doctest

```
b
Expecting:
  46
ok
1 items passed all tests:
  4 tests in bsp1
4 tests in 1 items.
4 passed and 0 failed.
Test passed.
$
```

Abb. 2.11: Der Interpreter zeigt, was er macht

Was gibt es zu sehen? Jede Quellcodezeile mit eingerücktem Python-Prompt wird unter einem Trying: eingerückt ausgegeben. Falls an das aktuelle Statement keine Erwartung geknüpft ist, wird jeweils Expecting nothing angezeigt, ansonsten nach dem Expecting: eingerückt die Erwartung und in der nächsten Zeile ok. Abschließend folgt ein Absatz mit statistischen Angaben zu den erfolgten Tests. Und wenn die Erwartung nicht erfüllt wird? Auch das ist mit einer winzigen Änderung einfach provoziert durch Änderung des erwarteten Wertes in der Datei.

```
"""doctest datei bsp1_fail.py
    >>> a = 24
    >>> a
    23
"""
```

1
2
3
4
5
6

Abb. 2.12: Ein absichtlicher Fehler

Derartige Fehler werden stets berichtet.

```
$ python -m doctest bsp1_fail.py -v
Trying:
  a = 24
Expecting nothing
ok
Trying:
  a
Expecting:
  23
*****
File "/home/hans/py/bsp1_fail.py", line 3, in bsp1_fail
Failed example:
  a
Expected:
  23
Got:
  24
*****
1 items had failures:
  1 of 2 in bsp1_fail
```

```

2 tests in 1 items.
1 passed and 1 failed.
***Test Failed*** 1 failures.
$

```

Abb. 2.13: Die Reaktion auf den Fehler mit Absicht

Ohne den Zusatz `-v` beschränkt sich die Anzeige auf die fehlerhaften Tests, mit dem Zusatz erscheint die ganze Statistik. Die Zeilennummern, bei denen Fehler erkannt werden, sind bei langen Dateien sehr hilfreich. Der Unterschied zwischen Expected und Got wird klar gegenübergestellt. Wo der verursachende Fehler zu suchen ist, findet sich nur im Quelltext. So dient auch der Testcode neben den Anwendungsquellen der Dokumentation des Projektes. Es wird nicht nur gezeigt, was durch Testfälle zuverlässig und jederzeit überprüft werden kann, sondern Aufrufe einzelner Teile des Anwendungsprogramms zeigen auch die mögliche Nutzung. Dies ist während der Entwicklung sehr nützlich. Insbesondere hilft es im Team, da sich der zweite Entwickler leichter in die Gedanken des Ersten hinein versetzen kann.

2.5 Dokumentierte Python-Datei

Python kennt die bereits erwähnten Zeichenketten zur Dokumentation, es ist guter Brauch, diese am Anfang von Dateien, Funktionen und Klassen zu verwenden. So ist erklärt, wozu der jeweilige Abschnitt nützlich sein kann. Das Werkzeug `pydoc` extrahiert aus Dateien diese Docstrings und zeigt sie schön formatiert an. Zuerst folgt beispielhaft ein sehr frühes Stadium eines Python-Moduls für eine neue Anwendung.

```

"""Dieses Modul definiert zu Demonstrationszwecken unnötigerweise
nochmals einige mathematischen Funktionen, die bereits in der
Standardbibliothek enthalten sind.
1
2
3
4
Author: Emil Mustermann, Musterstadt
5
Lizenz: t.b.d.
6
Datum: 1. 4. 2015
7
8
Funktionen: demo_add, demo_mul
9
"""
10
11
12
def demo_add(a, b):
13
    """demo_add addiert zwei gegebene ganze Zahlen und gibt das
14
    Ergebnis als ganze Zahl oder im Fehlerfall None zurück.
15
    >>> c = demo_add(17, 19)
16
    >>> type(c) == int
17
    True
18

```

2 Doctest

```
>>> c == 36
True
"""
pass
```

19
20
21
22

Abb. 2.14: Demonstration: `demo_add()`

Der Aufruf von `pydoc ./bsp2.py` gibt genau den Inhalt der Docstrings wieder.

Help on module bsp2:

NAME
bsp2

FILE
/home/hans/py/bsp2.py

DESCRIPTION
Dieses Modul definiert zu Demonstrationszwecken unnötigerweise nochmals einige mathematischen Funktionen, die bereits in der Standardbibliothek enthalten sind.

Author: Emil Mustermann, Musterstadt
Lizenz: t.b.d.
Datum: 1. 4. 2015

Funktionen: `demo_add`, `demo_mul`

FUNCTIONS
`demo_add(a, b)`
`demo_add` addiert zwei gegebene ganze Zahlen und gibt das Ergebnis als ganze Zahl oder im Fehlerfall `None` zurück.
`>>> c = demo_add(17, 19)`
`>>> type(c) == int`
`True`
`>>> c == 36`
`True`

Abb. 2.15: Extrahierte Dokumentation aus `demo_add()`

Der erste Docstring wird unverändert als Beschreibung des Moduls ausgegeben. Anschließend folgen die Docstrings der gefundenen Klassen und Funktionen. Durch die Tests kann darin beispielhaft der Aufruf dargestellt werden und auf mögliche Fallstricke hingewiesen werden.

Auch die Ausgabe einer HTML-Datei ist mit `pydoc` zu bewerkstelligen.

```
(wb) $ pydoc -w ./bsp2.py
wrote bsp2.html
(wb) $
```

Abb. 2.16: Extraktion der Dokumentation als html Datei

Der Inhalt ist der Gleiche, er sieht nur hübscher aus.



Abb. 2.17: Extrahierte Dokumentation aus `demo_add()` als Webseite

Selbstverständlich kann aufgrund der bis jetzt vorhandenen Implementierung kein Test gelingen, die Ausgabe kann dennoch nützlich sein.

```
$ python -m doctest bsp2.py
*****
File "/home/hans/py/bsp2.py", line 21, in bsp2.demo_add
Failed example:
    type(c) == int
Expected:
    True
Got:
    False
*****
File "/home/hans/py/bsp2.py", line 23, in bsp2.demo_add
Failed example:
    c == 36
Expected:
    True
Got:
    False
*****
1 items had failures:
  2 of  3 in bsp2.demo_add
```

2 Doctest

```
***Test Failed*** 2 failures.  
$
```

Abb. 2.18: Erster Testlauf mit `demo_add()`

Beide Fehler sind durch die noch nicht ganz perfekte Implementierung erklärt. „pass“ reicht nicht aus, jedoch können beide Tests dazu genutzt werden, die Funktion robust zu implementieren. Evtl. kommen während des Schreibens noch weitere Tests dazu. Im Docstring der Funktion ist schon ziemlich exakt beschrieben, wie eine Implementierung zu funktionieren hat. Es ist die Rede davon, ganze Zahlen zu addieren. Also muss zuerst geprüft werden, ob ganze Zahlen als Argumente übergeben wurden. Falls nicht, soll `None` zurückgegeben werden, sonst die Summe.

```
def demo_add(a, b):  
    """demo_add addiert zwei gegebene ganze Zahlen und gibt das  
    Ergebnis als ganze Zahl oder im Fehlerfall None zurück.  
    >>> c = demo_add(17, 19)  
    >>> type(c) == int  
    True  
    >>> c == 36  
    True  
    """  
    if type(a) == int and type(b) == int:  
        return a + b  
    else:  
        return None
```

Abb. 2.19: Testergebnisse verarbeitet in `demo_add()`

Die Prüfung `type(a) == int` prüft streng auf die Eigenschaft Ganzzahl. Eine etwas schwächere Prüfung könnte stattfinden mit `isinstance(a, int)` falls auch davon abgeleitetet Objekte eine Summe bilden sollen.

Die beiden Tests sind nun erfüllt. Reicht das schon als Zuverlässigkeitsprüfung? Nein, denn es ist ja nur einer von zwei möglichen Fällen getestet worden. Also wird mindestens ein weiterer Test benötigt, der `None` als Rückgabewert zur Folge hat. Eine Fließkommazahl ist (nicht nur) in Python keine ganze Zahl, ergo reicht ein einziger Test aus. Da beide Eingabewerte in der Funktion mit logischem und verknüpft sind, reicht ein Wert mit Dezimaltrennzeichen aus, um als Ergebnis `False` zu liefern.

```
def demo_add(a, b):  
    """demo_add addiert zwei gegebene ganze Zahlen und gibt das  
    Ergebnis als ganze Zahl oder im Fehlerfall None zurück.  
    >>> c = demo_add(17, 19)  
    >>> type(c) == int  
    True  
    >>> c == 36
```



<http://www.springer.com/978-3-662-48602-3>

Softwaretests mit Python

Hubertz, J.

2016, IX, 254 S. 150 Abb., Hardcover

ISBN: 978-3-662-48602-3