

Mithilfe dieser Einführung in die Techniken des Debuggens und mit ein wenig Experimentierfreude dürfte es für Sie kein Problem darstellen, den Debugger Ihrer favorisierten IDE in den Griff zu bekommen.

Abgesehen von dem eigentlichen Debugger umfasst die Standardbibliothek von Python noch einige Module, die speziell im Kontext des Debuggens von Bedeutung sind – sei es innerhalb der interaktiven Python-Shell eines Debuggers oder völlig losgelöst vom Debugger. Diese Module werden in den folgenden Abschnitten besprochen.

36.2 Automatisiertes Testen

Pythons Standardbibliothek stellt zwei Module zur *testgetriebenen Entwicklung* (engl. *test-driven development*) bereit. Unter testgetriebener Entwicklung versteht man eine Art der Programmierung, bei der viele kleine Abschnitte des Programms, sogenannte *Units*, durch automatisierte Testdurchläufe auf Fehler geprüft werden. Bei der testgetriebenen Entwicklung wird das Programm nach kleineren, in sich geschlossenen Arbeitsschritten so lange verbessert, bis es wieder alle bisherigen und alle hinzugekommenen Tests besteht. Auf diese Weise können sich durch das Hinzufügen von neuem Code keine Fehler in alten, bereits getesteten Code einschleichen.

In Python ist das Ihnen möglicherweise bekannte Konzept der *Unit Tests* im Modul `unittest` implementiert. Das Modul `doctest` ermöglicht es, Testfälle innerhalb eines Docstrings, beispielsweise einer Funktion, unterzubringen. Im Folgenden werden wir uns zunächst mit dem Modul `doctest` beschäftigen, um danach zum Modul `unittest` voranzuschreiten.

36.2.1 Testfälle in Docstrings – `doctest`

Das Modul `doctest` erlaubt es, Testfälle innerhalb des Docstrings einer Funktion, Methode, Klasse oder eines Moduls zu erstellen, die beim Aufruf der im Modul `doctest` enthaltenen Funktion `testmod` getestet werden. Die Testfälle innerhalb eines Docstrings werden dabei nicht in einer speziellen Sprache verfasst, sondern können direkt aus einer Sitzung im interaktiven Modus in den Docstring kopiert werden.

Hinweis

Docstrings sind auch bzw. hauptsächlich für die Dokumentation beispielsweise einer Funktion gedacht. Aus diesem Grund sollten Sie die Testfälle im Docstring möglichst einfach und lehrreich halten, sodass der resultierende Docstring auch in Dokumentationen Ihres Programms verwendet werden kann.

Das folgende Beispiel erläutert die Verwendung des Moduls doctest anhand der Funktion fak, die die Fakultät einer ganzen Zahl berechnen und zurückgeben soll:

```
import doctest
def fak(n):
    """
    Berechnet die Fakultät einer ganzen Zahl.

    >>> fak(5)
    120
    >>> fak(10)
    3628800
    >>> fak(20)
    2432902008176640000
```

36

Es muss eine positive ganze Zahl übergeben werden.

```
>>> fak(-1)
Traceback (most recent call last):
...
ValueError: Keine negativen Zahlen!
"""

res = 1
for i in range(2, n+1):
    res *= i
return res
if __name__ == "__main__":
    doctest.testmod()
```

Im Docstring der Funktion fak steht zunächst ein erklärender Text. Dann folgt, durch eine leere Zeile davon abgetrennt, ein Auszug aus dem interaktiven Modus von Python, in dem Funktionsaufrufe von fak mit ihren Rückgabewerten stehen. Diese Testfälle werden beim Ausführen des Tests nachvollzogen und entweder für wahr oder für falsch befunden.

Auf diese einfachen Fälle folgen, jeweils durch eine Leerzeile eingeleitet, ein weiterer erklärender Text sowie ein Ausnahmefall, in dem eine negative Zahl übergeben wurde. Beachten Sie, dass Sie den Stacktrace eines auftretenden Tracebacks im Docstring weglassen können. Auch die im Beispiel stattdessen geschriebenen Auslassungszeichen sind optional.

Der letzte Testfall wurde in der Funktion noch nicht berücksichtigt, sodass er im Test fehlschlagen wird. Um den Test zu starten, muss die Funktion testmod des Moduls doctest aufgerufen werden. Aufgrund der if-Abfrage

```
if __name__ == "__main__":
    doctest.testmod()
```

wird diese Funktion immer dann aufgerufen, wenn die Programmdatei direkt ausgeführt wird. Der Test wird hingegen nicht durchgeführt, wenn die Programmdatei von einem anderen Python-Programm als Modul eingebunden wird. Im provozierten Fehlerfall lautet das Testresultat folgendermaßen:

```
*****
File "fak.py", line 17, in __main__.fak
Failed example:
    fak(-1)
Expected:
    Traceback (most recent call last):
    ...
    ValueError: Keine negativen Zahlen!
Got:
    1
*****
1 items had failures:
 1 of  4 in __main__.fak
***Test Failed*** 1 failures.
```

Jetzt erweitern wir die Funktion `fak` dahingehend, dass sie im Falle eines negativen Parameters die gewünschte Exception wirft:

```
def fak(n):
    """
    [...]
    """
    if n < 0:
        raise ValueError("Keine negativen Zahlen!")
    res = 1
    for i in range(2, n+1):
        res *= i
    return res
```

Durch diese Änderung werden bei erneutem Durchführen des Tests keine Fehler mehr angezeigt. Um genau zu sein: Es wird überhaupt nichts angezeigt. Das liegt daran, dass generell nur fehlgeschlagene Testfälle auf dem Bildschirm ausgegeben werden. Sollten Sie auch auf der Ausgabe geglückter Testfälle bestehen, starten Sie die Programmdatei mit der Option `-v` (für `verbose`).

Beachten Sie bei den Vorgaben verglichen werden bestimmter Datengänge einer Menge in tionen, die vom In entspricht die Ide wegen natürlich be Eine weitere Beson warteten Ausgabe den muss, da eine fungiert:

```
def f(a, b):
    """
    ...
    >>> f(3, 4)
    7
    <BLANKLINE>
    12
    ...
    print(a + b)
    print()
    print(a * b)
```

Flags

Um einen Testfall Das sind Einstellun Form eines Komma von einem Plus (+) wir zu einem kon Tabelle 36.1) ke

Flag

EINPUNKT

Tabelle 36.1 Dode

Beachten Sie bei der Verwendung von Doctests, dass die in den Docstrings geschriebenen Vorgaben Zeichen für Zeichen mit den Ausgaben der ausgeführten Testfälle verglichen werden. Dabei sollten Sie stets im Hinterkopf behalten, dass die Ausgaben bestimmter Datentypen nicht immer gleich sind. So stehen beispielsweise die Einträge einer Menge in keiner garantierten Reihenfolge. Darüber hinaus gibt es Informationen, die vom Interpreter oder anderen Gegebenheiten abhängen; beispielsweise entspricht die Identität einer Instanz intern ihrer Speicheradresse und wird sich deswegen natürlich beim Neustart des Programms ändern.

Eine weitere Besonderheit, auf die Sie achten müssen, ist, dass eine Leerzeile in der erwarteten Ausgabe einer Funktion durch den String <BLANKLINE> gekennzeichnet werden muss, da eine Leerzeile als Trennung zwischen Testfällen und Dokumentation fungiert:

```
def f(a, b):
    """
    >>> f(3, 4)
    7
    <BLANKLINE>
    12
    """
    print(a + b)
    print()
    print(a * b)
```

36

Flags

Um einen Testfall genau an Ihre Bedürfnisse anzupassen, können Sie *Flags* vorgeben. Das sind Einstellungen, die Sie aktivieren oder deaktivieren können. Ein Flag wird in Form eines Kommentars hinter den Testfall im Docstring geschrieben. Wird das Flag von einem Plus (+) eingeleitet, wird es aktiviert, bei einem Minus (-) deaktiviert. Bevor wir zu einem konkreten Beispiel kommen, lernen Sie die drei wichtigsten Flags (siehe Tabelle 36.1) kennen.

Flag	Bedeutung
ELLIPSIS	Wenn dieses Flag gesetzt ist, kann die Angabe ... für eine beliebige Ausgabe einer Funktion verwendet werden. So können veränderliche Angaben wie Speicheradressen oder Ähnliches in größeren Ausgaben überlesen werden.

Tabelle 36.1 Doctest-Flags

Flag	Bedeutung
NORMALIZE_WHITESPACES	Wenn dieses Flag gesetzt ist, werden Whitespace-Zeichen nicht in den Ergebnisvergleich einbezogen. Das ist besonders dann interessant, wenn Sie ein langes Ergebnis auf mehrere Zeilen umbrechen möchten.
SKIP	Dieses Flag veranlasst das Überspringen des Tests. Das ist beispielsweise dann nützlich, wenn Sie im Docstring zu Dokumentationszwecken eine Reihe von Beispielen liefern, aber nur wenige davon bei einem Testlauf berücksichtigt werden sollen.

Tabelle 36.1 Doctest-Flags (Forts.)

In einem einfachen Beispiel erweitern wir den Doctest der bereits bekannten Fakultätsfunktion um die Berechnung der Fakultät einer relativ großen Zahl. Da es müßig wäre, alle Stellen des Ergebnisses im Doctest anzugeben, soll die Zahl mithilfe des Flags ELLIPSIS gekürzt angegeben werden:

```
import doctest
def fak(n):
    """
        Berechnet die Fakultät einer ganzen Zahl.

    >>> fak(1000) # doctest: +ELLIPSIS
    402387260077093773543702...000
    >>> fak("Bla") # doctest: +SKIP
    'BlubbBlubb'
    """

    res = 1
    for i in range(2, n+1):
        res *= i
    return res
if __name__ == "__main__":
    doctest.testmod()
```

Das Setzen der Flags wurde fett hervorgehoben. Wie Sie sehen, umfasst das Beispiel einen zweiten – offensichtlich fehlschlagenden – Test, bei dem aber das SKIP-Flag gesetzt wurde. Deshalb wird ein Testlauf hier keinen Fehler feststellen.

Bleibt noch zu sagen, dass insbesondere die Funktion testmod eine Fülle von Möglichkeiten bietet, die Testergebnisse im Programm zu verwenden oder den Prozess des Testens an Ihre Bedürfnisse anzupassen. Sollten Sie daran interessiert sein, bietet sich die Python-Dokumentation an, in der die Funktion besprochen wird.

36.2.2 Unit Tests – unittest

Das zweite Modul zur testgetriebenen Entwicklung heißt `unittest` und ist ebenfalls in der Standardbibliothek enthalten. Das Modul `unittest` implementiert die Funktionalität des aus Java bekannten Moduls JUnit, das den De-facto-Standard zur testgetriebenen Entwicklung in Java darstellt.

Der Unterschied zum Modul `doctest` besteht darin, dass die Testfälle bei `unittest` außerhalb des eigentlichen Programmcodes in einer eigenen Programmdatei in Form von regulärem Python-Code definiert werden. Das vereinfacht die Ausführung der Tests und hält die Programmdokumentation sauber. Umgekehrt ist mit dem Erstellen der Testfälle allerdings mehr Aufwand verbunden.

Um einen neuen Testfall mit `unittest` zu erstellen, müssen Sie eine von der Basisklasse `unittest.TestCase` abgeleitete Klasse erstellen, in der einzelne Testfälle als Methoden implementiert sind. Die folgende Klasse implementiert die gleichen Testfälle, die wir im vorangegangenen Abschnitt mit dem Modul `doctest` durchgeführt haben. Dabei muss die zu testende Funktion `fak` in der Programmdatei `fak.py` implementiert sein, die von unserer Test-Programmdatei als Modul eingebunden wird.

```
import unittest
import fak
class MeinTest(unittest.TestCase):
    def testBerechnung(self):
        self.assertEqual(fak.fak(5), 120)
        self.assertEqual(fak.fak(10), 3628800)
        self.assertEqual(fak.fak(20), 2432902008176640000)
    def testAusnahmen(self):
        self.assertRaises(ValueError, fak.fak, -1)
if __name__ == "__main__":
    unittest.main()
```

Es wurde eine Klasse namens `MeinTest` erzeugt, die von der Basisklasse `unittest.TestCase` erbt. In der Klasse `MeinTest` wurden zwei Testmethoden namens `testBerechnung` und `testAusnahmen` implementiert. Beachten Sie, dass der Name solcher Testmethoden mit `test` beginnen muss, damit sie später auch tatsächlich zum Testen gefunden und ausgeführt werden.

Innerhalb der Testmethoden werden die Methoden `assertEqual` bzw. `assertRaises` verwendet, die den Test fehlschlagen lassen, wenn die beiden angegebenen Werte nicht gleich sind bzw. wenn die angegebene Exception nicht geworfen wurde.

Um den Testlauf zu starten, wird die Funktion `unittest.main` aufgerufen. Die Fallunterscheidung

```
if __name__ == "__main__":
    unittest.main()
```

bewirkt, dass der Unit Test nur durchgeführt wird, wenn die Programmdatei direkt ausgeführt wird, und ausdrücklich nicht, wenn die Programmdatei als Modul in ein anderes Python-Programm importiert wurde. Die Funktion `unittest.main` erzeugt, um den Test durchzuführen, Instanzen aller Klassen, die im aktuellen Namensraum existieren und von `unittest.TestCase` erben. Dann werden alle Methoden dieser Instanzen aufgerufen, deren Namen mit `test` beginnen.

Die Ausgabe des Beispiels lautet im Erfolgsfall:

```
...
-----
Ran 2 tests in 0.000s
OK
```

Dabei stehen die beiden Punkte zu Beginn für zwei erfolgreich durchgeführte Tests. Ein fehlgeschlagener Test würde durch ein F gekennzeichnet.

Im Fehlerfall wird die genaue Bedingung angegeben, die zum Fehler geführt hat:

```
.F
=====
FAIL: testBerechnung (__main__.MeinTest)
-----
Traceback (most recent call last):
  File "testen.py", line 7, in testBerechnung
    self.assertEqual(fak.fak(5), 12)
AssertionError: 120 != 12
```

```
-----
Ran 2 tests in 0.001s
```

```
FAILED (failures=1)
```

Die Klasse `TestCase` erlaubt es zusätzlich, die parameterlosen Methoden `setUp` und `tearDown` zu überschreiben, die vor bzw. nach den Aufrufen der einzelnen Testmethoden ausgeführt werden. In diesen Methoden können also Initialisierungs- und Deinitialisierungsoperationen implementiert werden. Exceptions, die in `setUp` oder `tearDown` geworfen werden, lassen den jeweils aktuellen Test fehlschlagen.

Grundlegende Testmethoden
Aus dem vorangegangenen Abschnitt ist ersichtlich, dass die Methoden <code>assertEqual</code> , <code>assertNotEqual</code> , <code>assertTrue</code> , <code>assertFalse</code> , <code>assertIs</code> , <code>assertIsNot</code> , <code>assertIsNone</code> , <code>assertIsNotNone</code> , <code>assertIn</code> , <code>assertNotIn</code> , <code>assertIsInstance</code> , <code>assertNotIsInstance</code> , <code>assertGreater</code> , <code>assertGreaterEqual</code> , <code>assertLess</code> und <code>assertLessEqual</code> implementiert sind. Diese Methoden sind in Tabelle 36.2 zusammengefasst.
im Folgenden wird die Verwendung von <code>assertEqual</code> und <code>assertNotEqual</code> erläutert.
Die Methoden <code>assertEqual</code> und <code>assertNotEqual</code> sind die Basis für die übrigen assert-Methode.
ausgegeben werden. Tabelle 36.2 zeigt die Methoden, die von <code>assertEqual</code> abgeleitet sind.

Tabelle 36.2 Methoden, die von `assertEqual` abgeleitet sind

Grundlegende Testmethoden

Aus den vorangegangenen Beispielen kennen Sie bereits die Methoden `assertEqual` und `assertRaises`, mithilfe derer der einem Test zugrunde liegende Vergleich implementiert wird. Die Klasse `TestCase` definiert eine ganze Reihe solcher Methoden, die im Folgenden zusammengefasst werden.

Die Methoden verfügen alle über den optionalen Parameter `msg`, für den eine Fehlerbeschreibung angegeben werden kann, die im Falle eines fehlschlagenden Tests ausgegeben wird. Dieser Parameter wurde aus Gründen der Übersichtlichkeit in Tabelle 36.2 ausgelassen.

36

Methode	Testet auf
<code>assertEqual(first, second)</code>	<code>first == second</code>
<code>assertNotEqual(first, second)</code>	<code>first != second</code>
<code>assertTrue(expr)</code>	<code>bool(expr) is True</code>
<code>assertFalse(expr)</code>	<code>bool(expr) is False</code>
<code>assertIs(first, second)</code>	<code>first is second</code>
<code>assert IsNot(first, second)</code>	<code>first is not second</code>
<code>assertIsNone(expr)</code>	<code>expr is None</code>
<code>assert IsNotNone(expr)</code>	<code>expr is not None</code>
<code>assertIn(first, second)</code>	<code>first in second</code>
<code>assertNotIn(first, second)</code>	<code>first not in second</code>
<code>assertIsInstance(obj, cls)</code>	<code>isinstance(obj, cls)</code>
<code>assertNotIsInstance(obj, cls)</code>	<code>not isinstance(obj, cls)</code>
<code>assertGreater(first, second)</code>	<code>first > second</code>
<code>assertGreaterEqual(first, second)</code>	<code>first >= second</code>
<code>assertLess(first, second)</code>	<code>first < second</code>
<code>assertLessEqual(first, second)</code>	<code>first <= second</code>

Tabelle 36.2 Methoden der Klasse `TestCase`

Testen auf Exceptions

Die Klasse `TestCase` enthält die Methoden `assertRaises` und `assertWarns`, die verwendet werden können, um zu testen, ob Funktionen Exceptions bzw. Warnungen werfen. Sie können mit einer funktionalen Schnittstelle verwendet werden:

```
assertRaises(exc, fun, *args, **kwds)
```

Dabei wird getestet, ob das Funktionsobjekt `fun` bei der Ausführung mit den Parametern `args` und `kwargs` eine Exception vom Typ `exc` wirft.

Alternativ können sowohl `assertRaises` als auch `assertWarns` ein Kontextobjekt erzeugen:

```
with self.assertRaises(TypeError):
    pass
```

Der Vorteil dieser Schreibweise ist, dass der zu testende Code nicht extra in eine Funktion gekapselt werden muss.

Testen auf reguläre Ausdrücke

Zum Prüfen von Strings existieren die Methoden `assertRegex` und `assertNotRegex`, denen die Parameter `text` und `regex` übergeben werden. Ein Aufruf einer dieser Funktionen prüft, ob `text` auf den regulären Ausdruck `regex` passt bzw. nicht passt. Der reguläre Ausdruck `regex` kann sowohl als String als auch als RE-Objekt übergeben werden.

```
self.assertRegex("Test", r"Te.t")
```

Analog dazu existieren die Methoden `assertWarnsRegex` und `assertRaisesRegex`, die wie ihre Pendants aus dem vorangegangenen Abschnitt funktionieren, aber den Text der geworfenen Exception gegen einen regulären Ausdruck prüfen:

```
with self.assertRaises(TypeError, r"."):
    pass
```

36.3 Analyse des Laufzeitverhaltens

Die Optimierung eines Programms kann viel Zeit in Anspruch nehmen. In der Regel wird zunächst ein lauffähiges Programm erstellt, das alle gewünschten Anforderungen erfüllt, bei dem jedoch noch nicht unbedingt Wert auf die Optimierung der Algorithmitk gelegt wird. Das liegt vor allem daran, dass man oftmals erst beim fertigen Programm die tatsächlichen Engpässe erkennt und im frühen Stadium somit eventuell viel Zeit in die Optimierung unkritischer Bereiche investiert hätte.

Um das Laufzeitverhalten zu analysieren, existieren die drei Methoden `t.timeit`, `t.Timer` und `t.profile` in Python. Diese Methoden werden in den folgenden Abschnitten detailliert beschrieben.

36.3.1 Laufzeitmessung

Das Modul `timeit` enthält die Funktion `timeit`, die ein Python-Programm ausführt und dessen Laufzeit in Mikrosekunden zurückgibt.

Sie erinnern sich sicherlich an den Algorithmus zur Berechnung der Fibonacci-Zahlen, der rekursiv implementiert ist. Eine iterative Variante ist deutlich schneller.

Um die Laufzeit einer Funktion zu messen, kann die Klasse `Timer` instanziiert werden. Sie unterstützt die beschriebene Syntax.

`Timer([stmt, setup])`

Der zu analysierende Code wird in `stmt` als String übergeben. Ein String übergeben ist optional. Der Parameter `setup` ist ebenfalls optional. Der Parameter `globals` kann eine Liste von globalen Variablen angeben, in dessen Kontext die Funktion ausgeführt wird.

Nachdem eine Instanz von `Timer` erstellt wurde, kann sie im Folgenden benutzt werden.

`t.timeit([number])`

Diese Methode führt die Funktion `stmt` `number` Mal auf und gibt die Summe der Laufzeiten zurück. Der Parameter `number` ist optional und hat den Wert 1000000000.

Die Funktion gibt die gesamte Laufzeit aller Wiederholungen zurück, die sie ausgeführt hat. Der Wert wird in Nanosekunden zurückgegeben.