



Hochschule für angewandte Wissenschaften Augsburg
Fakultät für Informatik

Bachelorarbeit

Python Test-Tools für Test-driven development im Vergleich

zur Erlangung des akademischen Grades
Bachelor of Science

Thema:	Python Test-Tools für Test-driven development im Vergleich
Autor:	Maximilian Konter maximilian.konter@hs-augsburg.de MatNr. 951004
Version vom:	9. April 2019
1. Betreuer:	Dipl.-Inf. (FH), Dipl.-De Erich Seifert, MA
2. BetreuerIn:	Prof. Dr. X

Diese Arbeit befasst sich mit den aktuellen Tools zum testen von Python Software im Aspekt Test-driven development.

Dabei wird auf verschiedene Test-Tools eingegangen um diese dann zu vergleichen um so einen Leitfaden für das richtige Test-Tool zu bieten. Der Vergleich behandelt die Anwendbarkeit, Effizienz, Komplexität sowie die Erweiterbarkeit der Tools im Bezug auf Test-driven development.

Am Ende dieser Arbeit wird über die allgemeine Anwendbarkeit von Test-driven development diskutiert.

Inhaltsverzeichnis

Abbildungsverzeichnis	5
Listingverzeichnis	6
Glossar	7
Abkürzungsverzeichnis	8
1 Einleitung	8
1.1 Die Programmiersprache Python	8
1.2 Test-driven development	9
2 Python Test-Tools	9
2.1 Tools der Standard Bibliothek	10
2.1.1 unittest	10
2.1.2 doctest	12
2.2 Tools abseits der Standard Bibliothek	13
2.2.0.1 pytest	16
2.2.1 Mocking Tools	18
2.2.1.1 stubble	19
2.2.1.2 mocktest	19
2.2.1.3 flexmock	19
2.2.1.4 python-doublex	19
2.2.1.5 python-aspectlib	19
2.2.2 Fuzz-testing Tools	19
2.2.2.1 hypothesis	19
3 Zusammenfassung	19
4 Vergleich der Tools	19
5 Kombinierung von Tools	19
6 Diskussion: Test-driven development in der Praxis	20
6.1 Stärken von Test-driven development	20
6.2 Schwächen von Test-driven development	20
6.2.1 Vermeidbare Schwächen	20
6.2.2 Unumgängliche Schwächen	20
6.3 Wirtschaftlichen Aspekte von Test-driven development	20
6.4 Zusammenfassung	20
7 Fazit	20
8 Nachwort	21

Inhaltsverzeichnis	5
Literaturverzeichnis	22
Listings	22
Eidesstattliche Erklärung	30

Abbildungsverzeichnis

Listingverzeichnis

1	Basis Modul zum testen	22
2	unittest einfaches Beispiel	23
3	unittest einfaches Beispiel: Output erfolgreich	23
4	unittest einfaches Beispiel: Output misslungen	23
5	unittest my_module	24
6	doctest unterscheidet Typen	25
7	doctest unterscheidet Typen: Output	25
8	doctest verbose Output	25
9	doctest: my_module	26
10	doctest my_module: Textdatei	27
11	pytest Abhängigkeiten	28
12	pytest my_module	28
13	pytest my_module: Output	28

Glossar

bug Ein Bug(zu deutsch Käfer) ist ein Fehler in einem Programm oder in einer Software. 14

commit Ein commit ist die Sammlung von Änderungen an Dateien, welche mithilfe eines VCS verwaltet werden. 14

contextmanager Ein Contextmanager in Python ist ein Objekt, das einen Kontext bietet, in dem gearbeitet werden kann. Dieser Kontext wird mit `with contextmanager as c` geöffnet und schließt sich selbst beim verlassen des Kontextes. Dadurch stehen dem Entwickler der nutzbare Kontext als Variable `c` zur Verfügung. 16

decorator Ein Decorator ist eine Funktion, die eine andere Funktion oder Methode umschließt, wodurch der Code im Decorator vor und/oder nach der Funktion oder Methode ausgeführt werden kann. Ein Decorator wird einer anderen Funktion mit `@decorator-funktion` über der Funktions- oder Methoden-Definition übergeben. 17

docstring Ein Docstring ist ein Block von Text, der sich zwischen jeweils drei Anführungsstrichen befindet. Mit diesem Text wird ein Objekt in Python dokumentiert. 13

fixture Beschreibt die Präparation von Tests mit festen (fix) Werten, so wird zum Beispiel vor jedem Test eine Variable gesetzt. Dies dient der Übersichtlichkeit, da dies dann nicht in jedem Test vorgenommen werden muss und um Varianz zwischen Tests und Testläufen zu vermeiden. 9–13, 16–18

fuzz-testing Beschreibt eine Art von Test, bei dem das zu testende Modul oder Programm mit zufälligen Werten aufgerufen wird. Dabei soll jede mögliche Anwendung des Moduls oder Programms dargestellt werden. 9, 16

mock Etwas mocken bedeutet, ein Objekt durch ein falsches Objekt, den Mock, zu ersetzen, das genau so aussieht, wie das erwartete Objekt, aber nicht das Gleiche ist. So ist es möglich eine Funktion zu mocken, wodurch nicht die Funktion, sondern der Mock aufgerufen wird und das eingestellte Ergebnis liefert. 9, 11–13, 15, 17, 18

Paket-Manager Ein Paket-Manager ist ein Tool zum Verwalten von externen Python Bibliotheken, mit dessen Hilfe Pakete (Bibliotheken) installiert und versioniert werden können. 16

StackTrace Der StackTrace ist ein Protokoll der aufgerufenen Funktionen und Methoden, die ineinander verschachtelt sind, bis hin zu der tiefsten Funktion/Methode, die einen Fehler geworfen hat. 12

Abkürzungsverzeichnis

CLI	Command-line interface
GNU	GNU is not Unix
GPL	GNU General Public License
GUI	Graphical User Interface
LGPL	GNU Lesser General Public License
STDLIB	Python Standard Bibliothek
TDD	Test-driven development
VCS	Versions Control System

1 Einleitung

TDD wird in der heutigen Softwareentwicklung immer verbreiteter und beliebter. Die Ansprüche an Software sind in den letzten Jahren immer weiter gestiegen. Dies liegt vor allem an der Reichweite, die Software heute hat. So besitzen nach Schobelt, 2017 im Jahr 2018 bereits circa 66% aller Menschen ein Smartphone. Im Arbeitsleben ist ein PC meist gar nicht mehr weg zu denken. Doch mit den steigenden Nutzerzahlen steigen auch die Anforderungen, welche die Nutzer an die Software stellen. Somit wird die Anzahl der gefundenen Bugs dementsprechend größer.

Im weltweiten Markt gibt es viele große Unternehmen, die gegenseitig um die Nutzer kämpfen. Selbstverständlich präferieren die Nutzer diejenigen Anbieter, welcher die bessere Software bietet. Dies kann sich heute jedoch stetig ändern. Mit der steigenden Anzahl an Bugs, die gefunden werden, steigt auch die Anzahl der Nutzer, die von diesen Bugs betroffen sind. Diese Bugs sollen natürlich schnellstmöglich gefixt werden um so zu verhindern, dass die Nutzer die Software wechseln.

So schwer es ist seine Nutzer zu halten, umso schwerer ist es, zum Start einer Software Nutzer zu akquirieren. Es gibt bereits Software, die ähnliche Services anbieten. So ist es noch schwerer dem Markt beizutreten. Die Anforderungen werden durch die bereits am Markt tätigen Firmen gesetzt, damit sollten Fehler, die bereits gelöst wurden nicht mehr auftauchen.

Für Unternehmen sind diese Anforderungen meist schwer zu meistern, weshalb Software meist mit Fehlern released wird, um diese dann von den Nutzern aufdecken zu lassen und zu fixen.

Sowohl als Entwickler als auch als Arbeitgeber muss man sich bei der Wahl der Programmiersprache Gedanken darüber ob und wie einfach eine Sprache für Test-driven development ein zu setzen ist. Der wichtigste Aspekt bei diesem Prozess ist die Auswahl und die Qualität der von der Sprache zur Verfügung gestellten Tools.

1.1 Die Programmiersprache Python

In diesem Kapitel wird die Programmiersprache Python beschrieben, um einen genauen Überblick über die Sprache zu bekommen mit dem diese Arbeit besser verstanden werden kann.

1.2 Test-driven development

In diesem Kapitel wird das Thema testen und Test-driven development behandelt um ein Grundlegendes Verständnis von TDD zu schaffen mit welchem diese Arbeit besser zu verstehen ist.

2 Python Test-Tools

Dieses Kapitel befasst sich mit den von der STDLIB bereitgestellten Test-Tools sowie denen aus externen Paketen. Diese werden unter [2.1](#) und [2.2](#) zusammengefasst, wobei diese unterteilt sind in unit-testing -, mock-testing - und fuzz-testing Tools.

Jedes Tool wird anhand folgender Aspekte untersucht:

- Anwendbarkeit:

Bietet das Tool alles, um TDD betreiben zu können? (Fixtures und Mocks) Mit wie vielen Paketen muss das Tool betrieben werden? (Mehr Abhängigkeiten führen zu mehr externe Entwicklern auf die man sich verlassen muss.)

- Effizienz:

Wie viel lässt sich mit diesem Tool möglichst einfach und schnell erreichen? Ist besonders viel Vorarbeit notwendig um die Tests auf zu setzen oder kann sofort mit dem Schreiben der Tests begonnen werden?

Genauso stellt sich die Frage, wie Effizient der Entwickler die Tests auswerten kann.

- Komplexität:

Wie komplex ist das Tool? Das heißt, wie viel Funktionalität bietet das Tool dem Entwickler von Haus aus, aber auch wie schwer ist es einen Code zu schreiben oder wie schnell wird ein Code unübersichtlich, da das Tool viel Code abseits der Tests benötigt.

- Erweiterbarkeit:

Wie leicht lässt sich das Tool mit anderen Tools erweitern? Gibt es vielleicht Erweiterungen der Community für dieses Tool, die sehr hilfreich sind?

Diese Aspekte eines jeden Tools, werden mithilfe von dem in Listing 1 abgebildeten Code auf gezeigt. Dieser Code dient als das zu testende Modul.

Das Modul enthält eine selbst geschriebene Implementierung der Funktion `pow()`, welche eine Zahl `a` mit einer Zahl `b` quadriert. Um die Komplexität zu erhöhen wurde eine in-memory Datenbank implementiert, welche Items enthält. Jedes Item hat eine ID (`id`), einen Namen (`name`), einen Lager Platz (`storage.location`) und eine Anzahl der vorrätigen Items `amount`. Jedes Item besitzt zudem eine Methode `do_something()` welche eine externe Funktion/Methode aufruft, die jedoch noch nicht existiert `do_something_which_does_not_exist()`.

2.1 Tools der Standard Bibliothek

Die Standard Bibliothek von Python bietet zwei verschiedene Test-Tools(Muthukadan et al., 2011). Zum einen ist dies `unittest`¹ und zum anderen `doctest`². Diese beiden Tools reichen sind in ihrem Umfang bereits so vielseitig dass, es einfach ist eine hohe Test-Abdeckung eines Programms oder eine Bibliothek zu erreichen.

Beide Tools zählen zu den „Unit Testing Tools“(Muthukadan et al., 2011) - auf Deutsch Modul Test-Tools - mit deren Hilfe die einzelnen Module eines Programms getestet werden können. In einem Programm oder einer Bibliothek wären dies die einzelnen Funktionen und Methoden.

2.1.1 unittest

Das von JUnit inspirierte (Brandl, van Rossum, Heimes, Peterson, Pitrou et al., 2007) Tool `unittest`, ist bestand der Python Standardbibliothek und bietet seit jeher seinen Nutzern ein umfangreiches Repertoire an Funktionen zum testen von Python Code. Die Funktionen von `unittest` lassen sich unter folgenden Punkten beschreiben:

- Fixture, zum präparieren der Tests.
- Test Fälle, zum gliedern einzelner Tests.
- Testumgebungen, zum gliedern von zusammengehörigen Tests.
- Test runner, zum ausführen von Testumgebungen oder Test Fällen.

Mithilfe der genannten Punkte ist es dem Entwickler möglich eine Stabile Test Umgebung auf zu bauen. Jedoch bietet `unittest` alleine nicht alles um TDD betreiben zu können.

¹<http://pyunit.sourceforge.net/pyunit.html>

²<https://docs.python.org/3/library/doctest.html>

Bei TDD werden zuerst die Tests und dann die Funktionalitäten geschrieben, daher muss es möglich sein andere Module(units) zu mocken auf denen ein Test basiert. Durch die Fixtures ist es bereits möglich den Test oder die Tests so vor zu bereiten, dass diese funktionieren, jedoch bieten Mocks einfachere und schnellere Möglichkeiten Funktionen, Methoden, Klassen usw. zu imitieren.

Jedoch gibt es in der STDLIB eine Erweiterung zu unittest mit dem Namen `unittest.mock` welche unter eben diesem importiert werden kann um die mocking Funktionalität zu bekommen. Diese Erweiterung, auch als submodule bezeichnet ist Teil der STDLIB seit Python 3.3 wie in PEP417 von Foord et al., 2012 definiert wird.

Das Tool bietet des weiteren einen CLI, mit welchem es dem Benutzer möglich ist seine Tests gebündelt aus zu führen und aus zu werten. Mit dem CLI ist es auch möglich automatisch Tests in einem Ordner zu „entdecken“(discover) und aus zu führen. Dadurch ist es sehr leicht neue Tests in ein bestehendes Test System ein zu führen und diese ohne Veränderungen am bestehenden System aus zu führen.

Im Aspekt Anwendbarkeit bietet `unittest` alles um als Tool für TDD in Frage zu kommen, jedoch nur unter Einbezug der Erweiterung `unittest.mock`.

Mit Hilfe von `unittest` lässt sich sehr einfach und schnell ein Test schreiben, so würde das der Code aus Listing 2 bereits unsere selbst geschriebene `quadrat`-Funktion aus Listing 1 testen, einmal mit unserem selbst berechneten Wert und einmal gegen den wert der `quadrat`-Funktion aus der STDLIB.

Mit `self.assertEqual` wird überprüft ob der erste Wert dem zweiten Wert gleicht. Im ersten `assert` wird auf einen im Kopf ausgerechneten Wert geprüft und im zweiten wird die von Python gegebene Methode zum überprüfen verwendet.

Die basis-Funktionalität von `unittest` ist schnell verstanden und setzt sich eigentlich nur aus `self.assert{irgendwas}(...)` zusammen. Hat man die richtige `assert` Funktion gefunden lässt sich eigentlich jede unabhängige Funktion testen.

Möchte man allerdings fortgeschrittenere Tests schreiben so muss man sich der Dokumentation bedienen, welche unter <https://docs.python.org/3/library/unittest.html> zu finden ist. Würde man die Seite als PDF herunterladen so wären dies 58 Seiten Fließtext. Möchte man nun zum Beispiel vor den Tests etwas vorbereiten oder präparieren so lässt sich mit `setUp()` und `tearDown()` dies realisieren, diese zwei Methoden überschreiben die Methoden aus `unittest.TestCase` und werden vor jeder Funktion ausgeführt. Das Gleiche gibt es auch für den Test Fall, bei dem `setUp()` und `tearDown()` allerdings nur beim eintritt

der Klasse und beim Austritt ausgeführt werden. Diese Methoden sind die sogenannten Fixtures.

Die folgenden Listings 3 und 4 zeigen wie ein erfolgreicher Test und wie ein misslungener Test aussehen.

In Beiden Listings ist gut zu erkennen ob und was schief gegangen ist bei einem Test. Der erfolgreiche Test zeigt dem Nutzer sofort wie viele Tests in wie vielen Sekunden gelaufen sind, und ist dies erwünscht kann der Nutzer mit `--verbose` sich noch mehr Informationen anzeigen lassen.

Bei misslungenen Tests ist im Output immer der StackTrace abgebildet um so den Fehler bis zur Wurzel zurück verfolgen zu können. Auch der Fehler selbst wird in den Output geschrieben, wie in Zeile 8 in Listing 4 zu erkennen ist. Am Ende wird auch noch einmal angezeigt wie viele Tests schief gelaufen sind.

Die Effizienz Tests auszuwerten ist also sehr hoch, jedoch wird der Output bei mehreren Fehlern und langem StackTrace sehr schnell sehr unübersichtlich für das Terminal. Ein farbiger Output würde hier Abhilfe schaffen.

Um die Komplexität von `unittest` darzustellen wurde mithilfe des in Listing 1 definierten Codes ein fortgeschrittener Test geschrieben, welcher die Basis Features von `unittest` umfasst. Dieser Code befindet sich in Listing 5.

2.1.2 doctest

„Das `doctest` Modul sucht nach Textstücken, die wie interaktive Python-Sitzungen aussehen, und führt diese Sitzungen dann aus, um sicherzustellen, dass sie genau wie gezeigt funktionieren.“ (Brandl, van Rossum, Heimes, Peterson, Melotti et al., 2007). Diese Textstücke müssen sich in Kommentaren befinden, da sie sonst von Python als Code interpretiert werden.

Wie `unittest`, ist auch `doctest` in der `STDLIB`, wodurch keine externen Abhängigkeiten geladen werden müssen.

`Doctest` bietet dem Nutzer eine Möglichkeit mithilfe von einem Test den Code zu dokumentieren. Da `doctest` lediglich Code ausführt wie in einer interaktiven Python-Sitzung, ist auch nur das möglich auszuführen was im Code geschrieben ist. Testfälle oder gar Mocks sind hierbei nicht möglich, Fixtures hingegen sind teilweise realisierbar in Form von Code der vor dem Test ausgeführt wird.

`Doctest` selbst nutzt keine Assert-Funktionen oder Methoden, stattdessen wird der Output eines Befehls überprüft. Dabei spielt der Typ auch eine Rolle, so ergibt eine Funktion `return_3(s=None, i=None)` bei `s=True` eine `'3'` und bei `i=True` eine `3`. Dieses Beispiel

ist in Listing 6 zu sehen, um einen Output zu erzeugen wurde noch ein fehlerhafter Test hinzu gefügt. Dieser Output ist in Listing 7 zu sehen.

Möchte der Entwickler einen etwas größeren Test schreiben, so kann er sich einer Funktionalität von doctest bedienen, die es Ihm ermöglicht Tests in eine externe Datei zu verlagern. Dabei wird die Datei als ein Docstring behandelt, wodurch sie keine """ benötigt. Um jedoch Code in diese Datei ausführen zu können muss das zu testende Modul importiert werden.

Durch das schreiben der Tests in den Docstrings werden die Module in denen Tests geschrieben wurden schnell sehr unübersichtlich und lang, wenn der Entwickler große Tests schreibt. Zwar wird durch externe Textdateien Abhilfe geschaffen, dennoch sind doctests wenn sie länger werden schwer zu lesen und nach zu vollziehen.

Die Komplexität als auch die Anforderungen für doctest sind sehr gering. Das testen geschieht mithilfe der interaktiven Python-Sitzung, welche jedem Python Entwickler bekannt ist. Die Tests selbst sind aufrufe der geschriebenen Funktionen und Methoden und ein Abgleich des Outputs mit dem Erwarteten Wert.

Das Listing 9 zeigt den Code aus 1 mit Docstrings versehen. Dieser Test wurde mithilfe der main Funktion von Python ausgeführt. Der in Listing 10 ausgelagerte Test, wurde mithilfe des CLIs von doctest ausgeführt. Da beide Test keine Fehler werfen existiert auch kein Output für diese Tests.

Die einzige Möglichkeit hier einen Output zu bekommen wäre mit `-v` im CLI oder `verbose=True` im Funktionsaufruf. Die dort dargestellten Informationen sind lediglich welche Kommandos ausgeführt wurden, was erwartet wurde und das, dass erwartete eingetroffen ist. Ein kleines Beispiel ist in Listing 8 zu sehen.

Da doctest selbst keine Test Fälle unterstützt besitzt `unittest` eine Integration für doctest, diese ermöglicht es Tests aus Kommentaren sowie Textdateien in unittest zu integrieren und zu gliedern.

Da sich mit Doctest leider Funktionen nicht präparieren lassen, ist dieses Test Modul für die alleinige Anwendung in TDD nicht nutzbar. Das Tool bietet keine Möglichkeiten Objekte zu mocken wodurch Tests an nicht implementieren Methoden und Funktionen scheitern. Auch Fixtures sind nur bedingt realisierbar und benötigen viel Code der dupliziert werden muss, da dieser vor und nach jedem Test geschrieben werden muss.

2.2 Tools abseits der Standard Bibliothek

Abseits der Standard Bibliothek gibt es einige Tools deren Nutzung von Vorteil gegenüber der STDLIB ist. Diese werden unter diesem Punkt aufgeführt.

Auf <https://wiki.python.org/moin/PythonTestingToolsTaxonomy> werden viele externe Tools gelistet, jedoch scheinen viele inaktiv zu sein da ihre commits teilweise mehr als ein Jahr zurück liegen. Dies ist weder für eine Bibliothek noch für ein Tool ein gutes Zeichen, da sich die Anforderungen stetig ändern und niemals alle Bugs gefixt sind.

Manche der dort aufgelisteten Tools sind bereits oder werden gerade in andere Tools integriert. Dies kann zum einem sein, da ein Tool eine Erweiterung für ein anderes war und die Entwickler die Änderungen angenommen haben und zum anderen um die Tools zu verbessern und mehr Entwickler zur Verfügung zu haben. Eventuell sind auch andere Gründe dafür verantwortlich, jedoch war dies der Grund bei zum Beispiel [Testify](#) von Yelp ³.

Da sowohl Software als auch Programmiersprachen sich stetig weiter entwickeln werden in dieser Arbeit nur jene Tools behandelt die sich diesen Entwicklungen Anpassen, sei dies durch das unterstützen der aktuellsten Version von Python als auch durch neue Innovationen, sowie Bug-fixes. Aus diesem Grund werden Tools deren letzter Commit weiter als ein Jahr zurück liegt hier nicht behandelt.

Lässt man zusätzlich die Erweiterungen von Tool zunächst außen vor, so bleiben nach Muthukadan et al., 2011 folgende Modul Test-Tools zur Verfügung:

- [pytest](#)⁴

Tools wie [reahl.tofu](#)⁵ oder [zope.testing](#)⁶ sind zwar mehr oder weniger aktiv, da sie beide auf die neusten Python Versionen patchen, jedoch bieten sie sonst keinen Mehrwert in ihren Updates. [reahl.tofu](#)⁵ selbst ist auch nur eine Erweiterung für bestehende Test Tools, wie zum Beispiel `pytest`, weshalb es hier nicht behandelt wird. So wird [zope.testing](#)⁶ auch nicht behandelt, da dieses Tool wie bereits beschrieben nicht aktiv weiter entwickelt wird und es eher dafür gemacht wurde [zope](#)⁷ Applikationen zu testen und nicht Python Applikationen.

³<https://github.com/Yelp/Testify/>

⁴<https://github.com/pytest-dev/pytest/>

⁵<https://www.reahl.org/docs/4.0/devtools/tofu.d.html>

⁶<https://pypi.org/project/zope.testing/>

⁷<http://www.zope.org/en/latest/>

Ein sehr bekanntest Projekt ist [nose](#)⁸, welches aber nicht mehr in Entwicklung ist. Der Nachfolger [nose2](#)⁹ allerdings ist unter sehr aktiver Entwicklung und auch wenn das Tool nicht von Muthukadan et al., 2011 erwähnt wird, so wird es doch hier mit verglichen werden.

Auch sehr bekannt ist [testtools](#)¹⁰, jedoch ist auch hier keine wirkliche weitere Entwicklung zu verzeichnen.

Demnach wäre die liste an Modul Test-Tools folgende:

- [pytest](#)¹¹
- [nose2](#)¹²

Die Quelle aus dem Offiziellen Python Wiki beschreibt sonst keine weiteren Tools. Sucht auf Google nach weiteren Tools, so findet man die oben gelisteten wenn man nach den gleichen Kriterien Tools heraus filtert.

Als weitere Kategorie werden Mock-Tools geführt. Auch wenn fast alle unittest-Tools integriertes mocking haben, so lässt sich mit diesen Tools meist mehr erreichen. Es wurden die gleichen Filter-Kriterien verwendet wie bei den Test-Tools oben.

- [stubble](#)¹³
- [mocktest](#)¹⁴
- [flexmock](#)¹⁵
- [python-douplex](#)¹⁶
- [python-aspectlib](#)¹⁷

⁸<https://pypi.org/project/nose/1.3.7/>

⁹<https://pypi.org/project/nose2/>

¹⁰<https://pypi.org/project/testtools/>

¹¹<https://github.com/pytest-dev/pytest/>

¹²<https://pypi.org/project/nose2/>

¹³<https://www.reahl.org/docs/4.0/devtools/stubble.d.html>

¹⁴<https://github.com/timbertson/mocktest/tree/master>

¹⁵<https://github.com/bkabrda/flexmock>

¹⁶<https://bitbucket.org/DavidVilla/python-douplex>

¹⁷<https://github.com/ionelmc/python-aspectlib>

Auch hier finden sich sonst keine weiteren Tools die Open-source sind.

Als letzte Kategorie werden hier die Fuzz-testing Tools behandelt, da diese eine gute Möglichkeit bieten Code ausgiebig zu testen. Das wohl umfangreichste und nach den obigen Kriterien einzige Tool ist [hypothesis](#)¹⁸.

2.2.0.1 pytest

Das am 04. August 2009 in Version 1.0.0 veröffentlichte Tool pytest (auch py.test genannt) ist ein sehr umfangreiches und weit entwickeltes Tool. Seit 2009 wird das Tool stets weiter entwickelt und vorangetrieben, wodurch es eine Menge an Features gewonnen hat. Die Basis Features von pytest sind folgende:

- Simple assert statements.

Kein `self.assert`

Error Überprüfung mit Contextmanager

- Informativer Output in Farbe

Der gesamte Output kann angepasst werden

- Feature reiche Fixtures

Vordefinierte Fixtures von pytest

Geteilte Fixtures unter Tests

Globale Fixtures zwischen Modulen

Parametrisierung von Test als Fixtures

- Überprüfung von stdout und stderr

- Gliedern von Test in Fällen

... durch Markierung

... durch Nodes (Auswahl der Modul Abhängigkeit)

... durch String Abgleich der Funktions-Namen

Da pytest nicht in der STDLIB ist, muss es mit einem Paket-Manager installiert werden. Dabei werden für pytest 4.4.0 die in Listing 11 gezeigten Abhängigkeiten installiert.

¹⁸<https://github.com/HypothesisWorks/hypothesis>

Demnach benötigt pytest sechs externe Abhängigkeiten zusätzlich zu sich selbst.

Wie anhand der Basis Features erkennbar ist, bietet pytest einiges um Tests zu schreiben und aus zu führen. So ist mit den gebotenen Fixtures bereits eine Voraussetzung mehr als erfüllt, da pytest viele verschiedene Arten bietet Fixtures zu benutzen. Fixtures werden in pytest allerdings nicht mit `setUp` und `tearDown` geschrieben, sondern werden mithilfe eines decorators markiert und den Test Funktionen als Parameter übergeben. Um die `setUp` und `tearDown` Funktionalität zu bekommen muss lediglich das keyword `yield` verwendet werden, die Fixture wird dann den Code bis zum `yield` ausführen und nach der Funktion den Rest nach `yield` ausführen.

Diese lassen sich durch zusätzliche Parameter weiter anpassen. Diese Funktionen sind in Kapitel fünf nach Krekel und pytest-dev team, 2019 beschrieben.

Des weiteren bietet pytest auch mocking, so lässt sich beispielsweise mit `monkeypatch.setattr()` der return Wert einer Funktion ersetzen. Dies wird in pytest automatisch am ende der Funktion rückgängig gemacht, wodurch der Entwickler sich mehr auf das eigentliche Testen konzentrieren kann.

Auch hier bietet pytest weitere Möglichkeiten Mocks zu verwenden, dazu hat Krekel und pytest-dev team, 2019 in Kapitel sieben einige Worte geschrieben.

Ein weiteres sehr praktisches Feature von pytest ist die Gliederung in Fälle. Dies ist bei pytest sehr fein einstellbar, so lässt sich wie in den Basis Features bereits beschrieben ein Test mit einer oder mehreren Markierungen versehen wodurch eine Gliederung nach Markierung entsteht. Auch durch das selektieren bestimmten Wörtern lassen sich Tests nach ihrem Namen gliedern, so entsteht einerseits eine Gliederung zur Ausführung der Tests und andererseits eine Gliederung für die Entwickler die sie selbst im Code sehen können. Als letzte alternative lassen sich Tests anhand ihrer Module ausführen, dies geschieht durch die Angabe der Module. So würde `test_datei.py::TestKlasse::test_methode` den Test `test_methode` der Klasse `TestKlasse` in der Datei `test_datei.py` ausführen.

Diese Features werden in Kapitel sechs von Krekel und pytest-dev team, 2019 beschrieben.

Selbstverständlich bietet pytest noch weitere Features jedoch sind diese nicht zwangsläufig notwendig um TDD zu betreiben und sind mehr ein nice to have Feature als wirklich benötigt. Für eine Vollständige Auflistung und Erklärung aller Features kann jederzeit

unter <https://docs.pytest.org/en/latest/> die aktuellste Version der Dokumentation abgefragt werden.

Demnach bietet pytest alles um TDD anwenden zu können und die sieben zusätzlichen Abhängigkeiten die ein Entwickler bei der Nutzung von pytest eingeht sollten keinen Entwickler davon abhalten pytest und seine Features zu genießen.

Da pytest, wie bereits erwähnt keine neuen assert Methoden hinzufügt lässt sich sehr schnell und einfach ein Test schreiben. Selbst die Nutzung von Fixtures ist in pytest sehr einfach, da lediglich eine Funktion geschrieben werden muss die mit `@pytest.fixture` markiert wurde und der Test Funktion oder Methode als Parameter übergeben wird. Den Rest erledigt pytest selbst im Hintergrund. Genauso einfach gestaltet sich die Nutzung von Mocks.

Die Effizienz, die bei der Nutzung von pytest entsteht ist demnach sehr hoch. Denn der Entwickler muss nicht wirklich etwas neues dazu lernen um Tests zu schreiben oder zu präparieren. Genauso leicht kann ein Entwickler den Output von pytest auswerten, da dieser erstens, in Farbe ist, was die Lesbarkeit deutlich erhöht, zweitens sehr gut gegliedert ist und Wichtige Strukturen klar darstellt und drittens nach den Wünschen der Entwickler sich gestalten und verbessern lässt.

Durch die eben genannten Features im Bezug auf das schreiben von Tests mit assert, sowie Fixtures und Mocks lässt sich sagen das pytest sehr viel Funktionalität bietet wobei es trotzdem Struktur im Code bietet und komplexe Features einfach ermöglicht. Demnach kann ein Entwickler mit wenig Code viel Testing Funktionalität erstellen.

Sollten dem Entwickler die Features von pytest nicht ausreichen, so findet man unter <http://plugincompat.herokuapp.com/> eine Liste von 618 (Stand: 2. April 2019) Erweiterungen für pytest 4.3.0. pytest selbst kann allerdings auch als Erweiterung zu unittest genutzt werden, indem ein Entwickler pytest zum ausführen der unittests verwendet. Dadurch bietet sich dem Entwickler ein verbesserter Output des Test Ergebnisses.

Im Listing 12 und 13 befindet sich der Code und der Output zum Test von dem in Listing 1 definierten Modul.

2.2.1 Mocking Tools

Blubb

2.2.1.1 stubble

Blubb

2.2.1.2 mocktest

Blubb

2.2.1.3 flexmock

Blubb

2.2.1.4 python-douplex

Blubb

2.2.1.5 python-aspectlib

Blubb

2.2.2 Fuzz-testing Tools

Blubb

2.2.2.1 hypothesis

Blubb

3 Zusammenfassung

Hier werden in kurzen Punkten die Vor- und Nachteile der einzelnen Tools zusammengefasst, um dem Leser einen schnellen Überblick zur Verfügung zu stellen.

4 Vergleich der Tools

Die in Kapitel 3 aufgeführten Vor- und Nachteile werden hier untereinander verglichen. Wodurch eine Ausführung entstehen soll welches Tool am besten für Welchen Zweck geeignet ist.

5 Kombinierung von Tools

Sofern dies möglich ist werden hier mögliche Kombinationen von Tools diskutiert.

6 Diskussion: Test-driven development in der Praxis

Hier wird die Einleitung der Diskussion stehen.

6.1 Stärken von Test-driven development

Hier soll eine Diskussion der Schwächen von TDD entstehen.

6.2 Schwächen von Test-driven development

Dieses Kapitel soll die Schwächen von TDD Diskutieren, dabei wird in [Vermeidbare Schwächen](#) beschrieben welche dieser Schwächen umgangen werden können und in [Unumgängliche Schwächen](#) werden jene Schwächen die nicht vermeidbar sind analysiert.

6.2.1 Vermeidbare Schwächen

Vermeidbare Schwächen

6.2.2 Unumgängliche Schwächen

Unumgängliche Schwächen

6.3 Wirtschaftlichen Aspekte von Test-driven development

Dieses Kapitel befasst sich mit den möglichen Wirtschaftlichen folgen, die die Anwendung von TDD mit sich bringt.

6.4 Zusammenfassung

Hier wird die gesamte Diskussion zusammengefasst um danach im [Fazit](#) eine Fundierte Meinung wieder zu geben.

7 Fazit

FAZIT

8 Nachwort

Nachwort

Literaturverzeichnis

- Brandl, G., van Rossum, G., Heimes, C., Peterson, B., Melotti, E., Murray, R., ... Miss Islington (bot). (2007). *doctest* — *Test interactive Python examples*. Website. Online erhältlich unter <https://github.com/python/cpython/blob/3.7/Doc/library/doctest.rst> oder <https://docs.python.org/3/library/doctest.html>; abgerufen am 25. März 2019.
- Brandl, G., van Rossum, G., Heimes, C., Peterson, B., Pitrou, A., Dickinson, M., ... Palard, J. (2007). *unittest* — *Unit testing framework*. Website. Online erhältlich unter <https://github.com/python/cpython/blob/3.7/Doc/library/unittest.rst> oder <https://docs.python.org/3/library/unittest.html>; abgerufen am 25. März 2019.
- Foord, M., Melotti, E., Coghlan, N., Storchaka, S., Peterson, B., Huang, H. & Wijaya, M. (2012). PEP 417 – Including mock in the Standard Library. Website. Online erhältlich unter <https://github.com/python/peps/blob/master/pep-0417.txt> oder <https://www.python.org/dev/peps/pep-0417/>; abgerufen am 26. Januar 2019.
- Krekel, H. & pytest-dev team. (2019). *pytest Documentation - Release 4.4*. PDF. Online erhältlich unter <https://media.readthedocs.org/pdf/pytest/4.4.0/pytest.pdf>; abgerufen am 2. April 2019.
- Muthukadan, B., jtatum, CPE0014bf07ffd2-CM001ac30d4aca, 71-35-143-156, gjb1002, little-black-box, ... Barnes, S. (2011). *PythonTestingToolsTaxonomy*. Website. Online erhältlich unter <https://wiki.python.org/moin/PythonTestingToolsTaxonomy>; abgerufen am 14. März 2019.
- Schobelt, F. (2017). *Weltweite Smartphone-Verbreitung steigt 2018 auf 66 Prozent*. Online erhältlich unter <https://www.wuv.de/digital/weltweite-smartphone-verbreitung-steigt-2018-auf-66-prozent>; abgerufen am 25. Januar 2019.


```
1 """my_module.py"""
2 from sqlalchemy import (
3     Column,
4     Integer,
5     String,
6     create_engine,
7 )
8 from sqlalchemy.orm import sessionmaker
9 from sqlalchemy.ext.declarative import declarative_base
10
11 engine = create_engine('sqlite:///memory:')
12 Session = sessionmaker(bind=engine)
13 session = Session()
14 Base = declarative_base()
15
16 do_something_which_does_not_exist = None
17
18
19 def my_pow(a, b):
20     result = a
21     for _ in range(1, b):
22         result = result * a
23
24     return result
25
26
27 class Item(Base):
28     __tablename__ = 'items'
29
30     id_ = Column('id', Integer, primary_key=True)
31     name = Column(String(64))
32     storage_location = Column(Integer)
33     amount = Column(Integer)
34
35     def do_something(self):
36         return do_something_which_does_not_exist(self)
37
38     def __repr__(self):
39         return f'item<{self.id_}, {self.name}, {self.storage_location}, {self.amount}>'
40
41
42 Base.metadata.create_all(engine)
```

Listing 1: Basis Modul zum testen

```
1 import unittest
2
3 from my_package.my_module import my_pow
4
5
6 class TestMyModule(unittest.TestCase):
7     def test_my_pow(self):
8         self.assertEqual(my_pow(2, 2), 4)
9         self.assertEqual(my_pow(4, 8), pow(4, 8))
10
11
12 if __name__ == '__main__':
13     unittest.main()
```

Listing 2: unittest einfaches Beispiel

```
1 .
2
3 Ran 1 test in 0.000s
4
5 OK
```

Listing 3: unittest einfaches Beispiel: Output erfolgreich

```
1 F
2
3 FAIL: test_my_pow (unittest_..example.TestMyModule)
4
5 Traceback (most recent call last):
6   File "unittest_..example.py", line 9, in test_my_pow
7     self.assertEqual(my_pow(4, 7), pow(4, 8))
8   AssertionError: 16384 != 65536
9
10
11 Ran 1 test in 0.000s
12
13 FAILED (failures=1)
```

Listing 4: unittest einfaches Beispiel: Output misslungen

```
1 import unittest
2 import unittest.mock as mock
3
4 from my_package.my_module import (
5     Item,
6     session
7 )
8
9 class TestMyDatabase(unittest.TestCase):
10     def setUp(self):
11         self.item = Item(id_=1, name='name', storage_location=7, amount=3)
12
13     def test_creation(self):
14         session.add(self.item)
15         session.commit()
16
17         self.assertEqual(len(session.new), 0)
18         self.assertEqual(session.query(Item).first(), self.item)
19
20     @mock.patch('my_package.my_module.do_something_which_does_not_exist')
21     def test_external_function(self, mock_do_something_which_does_not_exist):
22         mock_do_something_which_does_not_exist.return_value = 42
23
24
25         self.assertEqual(self.item.do_something(), 42)
26         mock_do_something_which_does_not_exist.assert_called_with(self.item)
27
28     def tearDown(self):
29         session.close()
30
31 if __name__ == '__main__':
32     unittest.main()
```

Listing 5: unittest my_module

```

1 import random
2
3 def return_3(s=None, i=None):
4     """
5     >>> return_3(s=True)
6     '3'
7
8     >>> return_3(i=True)
9     3
10
11    >>> return_3(i=True)
12    '4'
13    """
14    if s:
15        return '3'
16    elif i:
17        return 3
18
19
20 if __name__ == '__main__':
21     import doctest
22     doctest.testmod()

```

Listing 6: doctest unterscheidet Typen

```

1 *****
2 File "example.py", line 14, in __main__.return_3
3 Failed example:
4     return_3(i=True)
5 Expected:
6     '4'
7 Got:
8     3
9 *****
10 1 items had failures:
11     1 of 3 in __main__.return_3
12 ***Test Failed*** 1 failures.

```

Listing 7: doctest unterscheidet Typen: Output

```

1 Trying:
2     from advanced import *
3 Expecting nothing
4 ok
5 Trying:
6     my_pow(2, 2)
7 Expecting:
8     4
9 ok

```

Listing 8: doctest verbose Output

```

1 def my_pow(a, b):
2     """
3     >>> my_pow(2, 2)
4     4
5     >>> my_pow(4, 8)
6     65536
7     """
8     result = a
9     for _ in range(1, b):
10         result = result * a
11
12     return result
13
14
15 class Item(Base):
16     """
17     Da Doctest keine Fixtures unterstuetzt muss hier der setUp Code stehen:
18     >>> Base.metadata.create_all(engine)
19     >>> item = Item(id_=1, name='name', storage_location=1, amount=1)
20     >>> session.add(item)
21     >>> session.commit()
22
23
24     Check ob das item committed wurde
25     >>> len(session.new)
26     0
27
28     Check ob das item in der Datenbank ist.
29     Das item wird mit seiner __repr__() Methode repräsentiert, da die
30     Werte bekannt sind kann ueberprueft werden ob diese uebereinstimmen.
31     >>> session.query(Item).first()
32     item<1, name, 1, 1>
33
34     do_something existiert nicht, da mit Doctest kein Mock erstellt werden
35     kann wird auf die Exception ueberprueft.
36     >>> item.do_something()
37     Traceback (most recent call last):
38     ...
39     TypeError: 'NoneType' object is not callable
40
41     Hier wird der tearDown Code ausgefuehrt
42     >>> session.close()
43     """
44     __tablename__ = 'items'
45
46     id_ = Column('id', Integer, primary_key=True)
47     name = Column(String(64))
48     storage_location = Column(Integer)
49     amount = Column(Integer)
50
51     def do_something(self):
52         return do_something_which_does_not_exist(self)

```

```

53
54     def __repr__(self):
55         return f'item<{self.id_}, {self.name}, {self.storage_location}, {self.amount}>'
56
57 if __name__ == '__main__':
58     import doctest
59     # Fuehrt die internen Tests aus
60     doctest.testmod()
61     # Fuehrt die externen Tests aus
62     doctest.testfile('advanced.txt')

```

Listing 9: doctest: my_module

```

1 In dieser Datei stehen alle Docstrings ausgelagert
2
3 Um my_module zu testen muss alles importiert werden.
4 >>> from my_module import *
5 >>> my_pow(2, 2)
6 4
7 >>> my_pow(4, 8)
8 65536
9
10
11 Da Doctest keine Fixtures unterstuetzt muss hier der setUp Code stehen:
12 >>> Base.metadata.create_all(engine)
13 >>> item = Item(id_=1, name='name', storage_location=1, amount=1)
14 >>> session.add(item)
15 >>> session.commit()
16
17
18 Check ob das item committed wurde
19 >>> len(session.new)
20 0
21
22 Check ob das item in der Datenbank ist.
23 Das item wird mit seiner __repr__() Methode representiert, da die
24 Werte bekannt sind kann ueberprueft werden ob diese uebereinstimmen.
25 >>> session.query(Item).first()
26 item<1, name, 1, 1>
27
28 do_something existiert nicht, da mit Doctest kein Mock erstellt werden kann
29 wird auf die Exception ueberprueft.
30 >>> item.do_something()
31 Traceback (most recent call last):
32     ...
33 TypeError: 'NoneType' object is not callable
34
35 Hier wird der tearDown Code ausgefuehrt
36 >>> session.close()

```

Listing 10: doctest my_module: Textdatei

```

1 atomicwrites==1.3.0
2 attrs==19.1.0
3 more-itertools==7.0.0
4 pluggy==0.9.0
5 py==1.8.0
6 pytest==4.4.0
7 six==1.12.0

```

Listing 11: pytest Abhängigkeiten

```

1 import pytest
2
3 from my_package.my_module import (
4     Item,
5     session,
6 )
7
8 class TestMyDatabase:
9     @pytest.fixture
10    def setUp_tearDown(self):
11        # setUp
12        self.item = Item(id_=1, name='name', storage_location=7, amount=3)
13        yield self.item
14        # tearDown
15        session.close()
16
17    def test_creation(self, setUp_tearDown):
18        session.add(self.item)
19        session.commit()
20
21        assert len(session.new) is 0, 'Lenght is not 0, session is dirty'
22        assert session.query(Item).first() is self.item, 'Item is not in databse'
23
24    def test_external_function(self, setUp_tearDown, monkeypatch):
25        def mock_return(something):
26            return 42
27
28        monkeypatch.setattr(Item, 'do_something', mock_return)
29        assert self.item.do_something() is 42, 'Do something wasn\'t patched'
30
31
32 if __name__ == '__main__':
33     pytest.main()

```

Listing 12: pytest my_module

```

1 ===== test session starts =====
2 platform linux -- Python 3.7.3, pytest -4.4.0, py -1.8.0, pluggy -0.9.0
3 rootdir: /this/is/my/secret/path
4 collected 2 items
5
6 pytest_my_module.py ..
7

```

[100%]

8 ===== 2 passed in 0.11 seconds =====

Listing 13: pytest my_module: Output

Eidesstattliche Erklärung

Eidesstattliche Erklärung zur Abschlussarbeit

Hiermit versichere ich, die eingereichte Abschlussarbeit selbständig verfasst und keine andere als die von mir angegebenen Quellen und Hilfsmittel benutzt zu haben. Wörtlich oder inhaltlich verwendete Quellen wurden entsprechend den anerkannten Regeln wissenschaftlichen Arbeitens zitiert. Ich erkläre weiterhin, dass die vorliegende Arbeit noch nicht anderweitig als Abschlussarbeit eingereicht wurde.

Das Merkblatt zum Täuschungsverbot im Prüfungsverfahren der Hochschule Augsburg habe ich gelesen und zur Kenntnis genommen. Ich versichere, dass die von mir abgegebene Arbeit keinerlei Plagiate, Texte oder Bilder umfasst, die durch von mir beauftragte Dritte erstellt wurden.

Unterschrift :

Ort, Datum :