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:	1. 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. Die Tools werden auch auf ihre Tauglichkeit zur Automatisierung mithilfe von Travis-CI[1] geprüft, da die Automatisierung von Tests für Test-driven development eine entscheidende Rolle spielt.

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

[1] https://travis-ci.org/

Inhaltsverzeichnis

Abbildungsverzeichnis

Listingverzeichnis

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. 13, 14

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 Präperation 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 Test läufen zu vermeiden. 10–13

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, 15

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, 11–14

StackTrace Der StackTrace ist ein Protokoll der aufgerufenen Funktionen und Methoden die ineinander verschachtelt sind, bis hin zu der tiefsten Funkion/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 denjenigen 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 Mark 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? (Bei TDD kann man Tests mocken, wodurch sie nicht fehlschlagen.) Mit wie vielen Paketen muss das Tool betrieben werden?

- 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[2] und zum anderen doctest[3]. Diese beiden Tools reichen sind im 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.

[2]http://pyunit.sourceforge.net/pyunit.html
[3]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 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 Test 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 Effizient Tests aus zu werten 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` dar zu stellen 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 aus zu führen was im Code geschrieben ist. Test Fälle 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.

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 [4].

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:

- py.test[5]
- tofu[6]
- zope.testing[7]

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[8]
- mocktest[9]
- flexmock[10]

[4] https://github.com/Yelp/Testify/
[5] https://github.com/pytest-dev/pytest/
[6] https://www.reahl.org/docs/4.0/devtools/tofu.d.html
[7] https://pypi.org/project/zope.testing/
[8] https://www.reahl.org/docs/4.0/devtools/stubble.d.html
[9] https://github.com/timbertson/mocktest/tree/master
[10] https://github.com/bkabrda/flexmock

- python-doublex[11]
- python-aspectlib[12]

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[13].

2.2.1 Unit-test Tools

Um einen besseren Überblick zu bieten werden die in 2.2 behandelten Unit-test Tools hier analysiert.

2.2.1.1 py.test

Blubb

2.2.1.2 tofu

Blubb

2.2.1.3 zope.testing

Blubb

2.2.2 Mocking Tools

Blubb

2.2.2.1 stubble

Blubb

2.2.2.2 mocktest

Blubb

2.2.2.3 flexmock

Blubb

[11] https://bitbucket.org/DavidVilla/python-doublex
[12] https://github.com/ionelmc/python-aspectlib
[13] https://github.com/HypothesisWorks/hypothesis

2.2.2.4 python-doublex

Blubb

2.2.2.5 python-aspectlib

Blubb

2.2.3 Fuzz-testing Tools

Blubb

2.2.3.1 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.

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.

```
"""my_module.py"""
from sqlalchemy import (
    Column,
    Integer,
    String,
    create_engine,
)
from sqlalchemy.orm import sessionmaker
from sqlalchemy.ext.declarative import declarative_base

engine = create_engine('sqlite:///:memory:')
Session = sessionmaker(bind=engine)
session = Session()
Base = declarative_base()

do_something_which_does_not_exist = None


def my_pow(a, b):
    result = a
    for _ in range(1, b):
        result = result * a

    return result


class Item(Base):
    __tablename__ = 'items'

    id_ = Column('id', Integer, primary_key=True)
    name = Column(String(64))
    storage_location = Column(Integer)
    amount = Column(Integer)

    def do_something(self):
        return do_something_which_does_not_exist(self)

    def __repr__(self):
        return f'item<{self.id_}, {self.name}, {self.storage_location}, {self.amount}>'


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.assertIs(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.assertIs(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 Basis Features

```
import random

def return_3(s=None, i=None):
    """
    >>> return_3(s=True)
    '3'

    >>> return_3(i=True)
    3

    >>> return_3(i=True)
    '4'
    """
    if s:
        return '3'
    elif i:
        return 3


if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

Listing 6: doctest unterscheidet Typen

```
**********************************************************************
File "example.py", line 14, in __main__.return_3
Failed example:
    return_3(i=True)
Expected:
    '4'
Got:
    3
**********************************************************************
1 items had failures:
   1 of   3 in __main__.return_3
***Test Failed*** 1 failures.
```

Listing 7: doctest unterscheidet Typen: Output

```
Trying:
    from advanced import *
Expecting nothing
ok
Trying:
    my_pow(2, 2)
Expecting:
    4
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 commited 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 representiert, 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('my_module.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 commited 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 Exceptioni 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: Textdatei

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.

$Unterschrift:$ $Ort, Datum:$