

Libraries

Fortsetzung

Packages

<https://pypi.org/>

Package Manager pip

pip install

Virtual Environment

__init__.py

DRY-Prinzip (Do not Repeat Yourself)

Try

- pytttsx3
- speech_recognition

Beispiel

My_library

+ My_package0 (folder)
 my_module0.py
 my_module1.py
 __init__.py

optional:

+ My_package1 (folder)
 my_module2.py
 (...)
 __init__.py

(...)

```
import speech_recognition as sr

r = sr.Recognizer()

with sr.Microphone() as source:
    audio_data = r.record(source, duration=5)
    print("Verarbeitung ...")
    text = r.recognize_google(audio_data, language="de-DE")
    print(text)
```

```
import pyttsx3
import speech_recognition as sr

engine = pyttsx3.init()
r = sr.Recognizer()

with sr.Microphone() as source:
    audio_data = r.record(source, duration=5)
    print("Denke nach ...")
    text = r.recognize_google(audio_data, language="de-DE")
    engine.say("Ich habe verstanden: " + text)
    engine.runAndWait()
```

```
# __init__.py
```

```
# Kann leer sein!
```

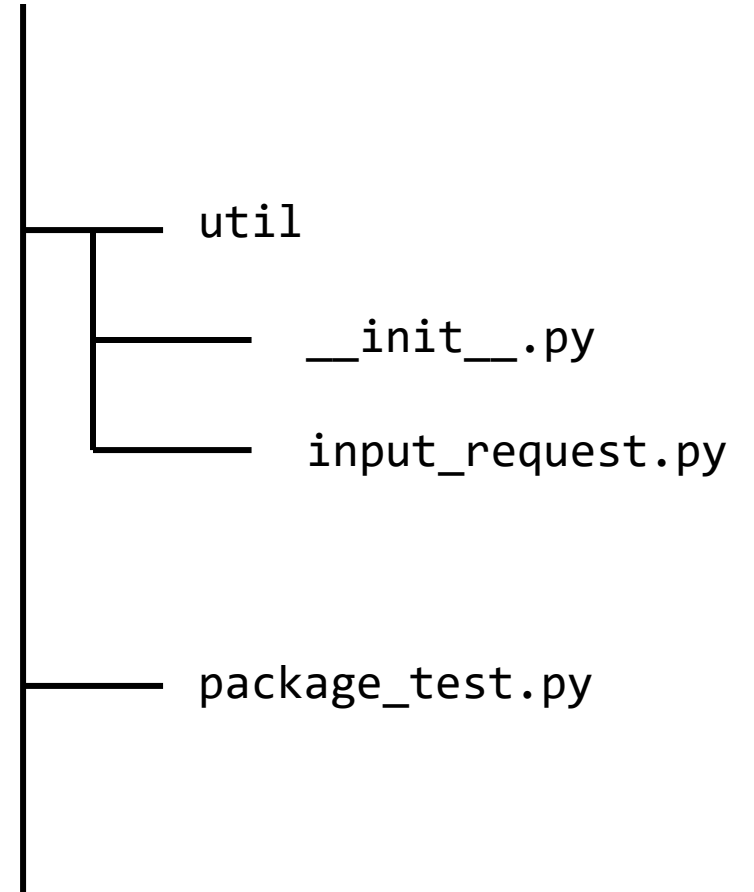
```
# input_request.py
```

```
def input_request(question = "Bitte um Eingabe: ", type = "string"):
    """question: string, type: string (integer or int, float, string or str)"""
    while True:
        try:
            if type == "integer" or type == "int":
                return int(input(question))
            elif type == "float":
                return float(input(question))
            elif type == "string" or type == "str":
                return str(input(question))
        except ValueError:
            print("Bitte geben Sie eine gültige Eingabe ein.")
            continue
```

```
# package_test.py
```

```
from util import input_request as ir
# VARIANTE
# from util.input_request import input_request as ir
```

```
result = ir.input_request("Nenne eine Zahl: ", "integer")
# IN DER VARIANTE
# result = ir("Nenne eine Zahl: ", "integer")
print(result)
```



APIs

Fortsetzung

Beispiele

Modul: requests

<https://requests.readthedocs.io/en/latest/>

JSON (JavaScript Object Notation)

Programmiersprachen unabhängig

Try

<https://api.corrently.io/v2.0/gsi/marketdata?zip=40597>

<https://world.openfoodfacts.org/api/v2/search?code=3263859883713>

```
import requests
import random
from prompt_toolkit import print_formatted_text, HTML

# https://opentdb.com/api_config.php
url = "https://opentdb.com/api.php?amount=10&category=15"

response = requests.get(url)
result = response.json()

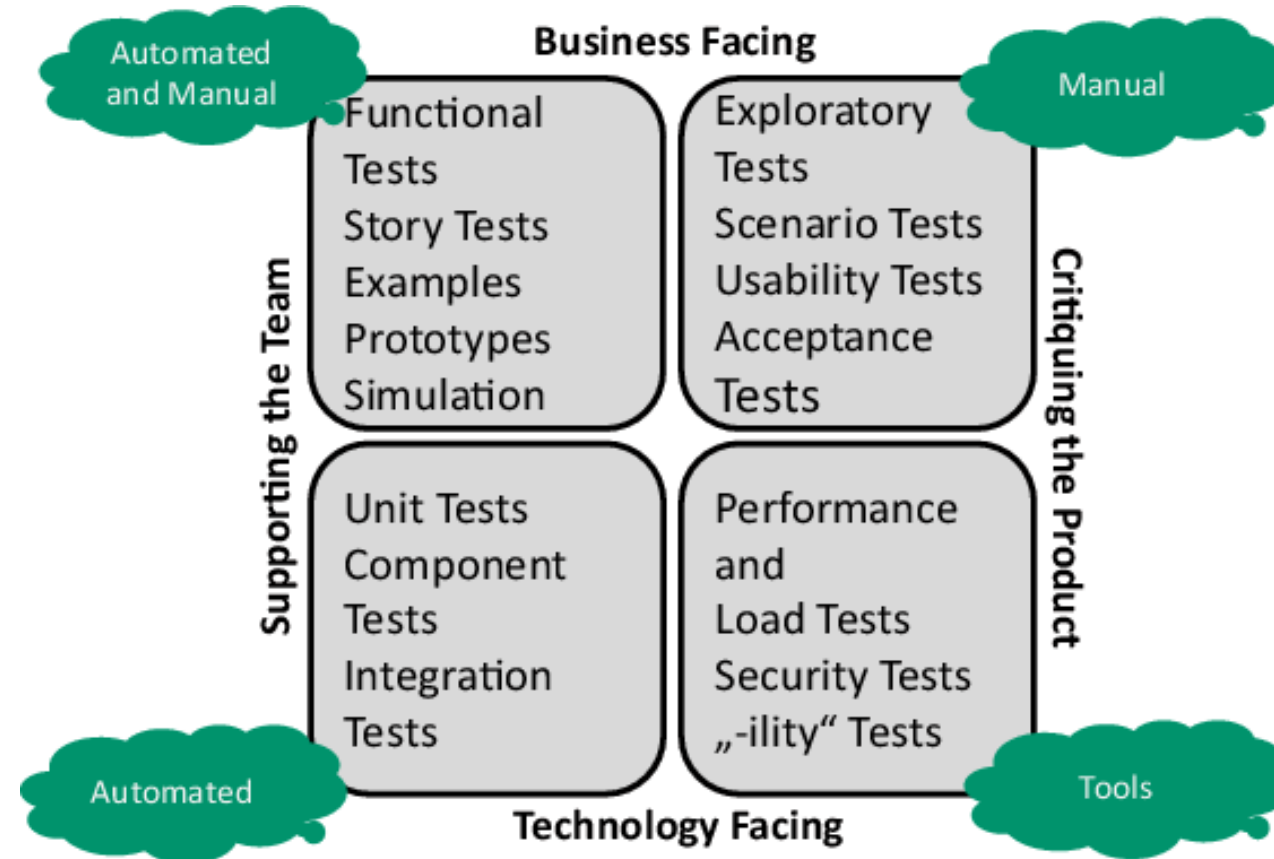
for i in range(10):
    print_formatted_text(HTML(result["results"][i]["question"]))
    if result["results"][i]["type"] == "multiple":
        choice = []
        for j in range(3):
            choice.append(result["results"][i]["incorrect_answers"][j])
        choice.append(result["results"][i]["correct_answer"])
        random.shuffle(choice)
        for j in range(4):
            print_formatted_text(HTML(f"{j+1}: {choice[j]}"))
    input("See the answer: ")
    print_formatted_text(HTML(result["results"][i]["correct_answer"]))
    print("")
```

Tests

Ein paar Beispiele

- **Frontend tests:** Werden zum Beispiel alle essentiellen Informationen angezeigt, ist es möglich einen bestimmten Pfad „durchzuklicken“
- **Szenario tests:** Testen komplexe Abläufe und Systemteile
- **Integration Tests:** Testen das Zusammenspiel mehrere Komponenten
- **Acceptance tests** oder auch „User Acceptance Test“ (UAT): Prüfen, ob die (User-) Anforderungen erfüllt wurden
- **Functional tests:** Ein Feature / eine Funktion wird geprüft
- **Destructive tests:** Testet, ob und wie die Software arbeitet, wenn Teile defekt sind
- **Performance tests:** Prüft das Verhalten der Software unter Last (z. B. viele gleichzeitige User-Interaktionen, große Datenmengen, etc.)
- **Usability** oder **User Experience (UX)** tests: Wie effektiv und effizient ist die Nutzung für den User (Usability), wie ist das Benutzererlebnis vor, während und nach der Nutzung (UX)
- **Security tests** und **Penetration tests:** Prüft die Software auf Sicherheitsschwachstellen
- **Regression tests:** Prüft Wirkung eines Updates auf ein Feature
- **Unit tests:** Grundlegende Prüfung von Code (Robustheit und Konsistenz); unverzichtbar beim Refactoring

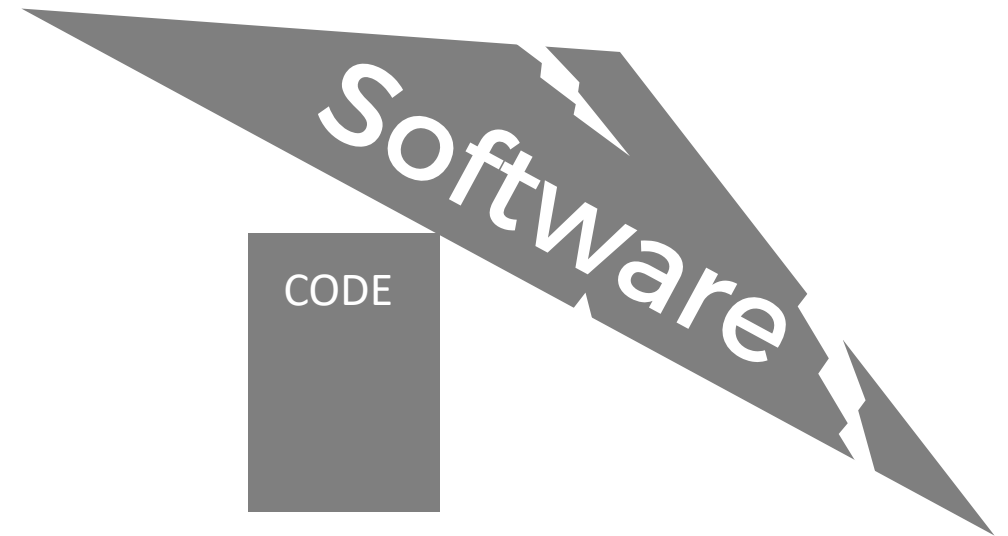
Testbereiche



Unit Tests

Developer's best friend

Produktiver Code ohne Tests ist nicht erlaubt



(Ausnahmen sind zum Beispiel Prototypen, Codeschnipsel zu Übungszwecken;
aber nur, wenn sie niemals in Produktion genommen werden.)

Konventionen

1. Benenne die Datei test_NAME DER FUNKTION.py
2. Benenne die Testfunktion test_NAME DES TESTS

Denke ab jetzt immer an

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

```
# calculator.py
def main():
    x = int(input("Gib eine Zahl ein: "))
    print(square(x))

def square(x):
    """Returns the square of x."""
    return x * 2

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

```
# test_square.py
from calculator import square

def main():
    test_square()

def test_square():
    if square(2) != 4:
        print("2 squared should equal 4")
    if square(3) != 9:
        print("3 squared should equal 9")

if __name__ == '__main__':
    main()
```

assert

```
# test_square.py

from calculator import square

def main():
    test_square()

def test_square():
    assert square(2) == 4, "2 squared should equal 4"
    assert square(3) == 9, "3 squared should equal 9"

if __name__ == '__main__':
    main()
```


AssertionError

```
# test_square.py

from calculator import square

def main():
    test_square()

def test_square():
    try:
        assert square(2) == 4
    except AssertionError:
        print("2 squared should equal 4")
    try:
        assert square(3) == 9
    except AssertionError:
        print("3 squared should equal 9")
    try:
        assert square(0) == 0
    except AssertionError:
        print("0 squared should equal 0")
    try:
        assert square(-2) == 4
    except AssertionError:
        print("-2 squared should equal 4")
    try:
        assert square(-3) == 9
    except AssertionError:
        print("-3 squared should equal 9")

if __name__ == '__main__':
    main()
```

pytest

`pip install pytest`
docs.pytest.org

```
# test_square.py
from calculator import square
import pytest

def test_square():
    assert square(2) == 4
    assert square(3) == 9
    assert square(0) == 0
    assert square(-2) == 4
    assert square(-3) == 9
```

PS C:\Users\witzmann\test2> **pytest .\test_square.py**

```
===== test session starts =====
platform win32 -- Python 3.8.5, pytest-6.1.1, py-1.9.0, pluggy-0.13.1
rootdir: C:\Users\witzmann\test2
plugins: cov-2.10.1, django-4.1.0
collected 1 item
```

test_square.py F [100%]

```
===== FAILURES =====
_____ test_square _____
```

```
def test_square():
    assert square(2) == 4
>    assert square(3) == 9
E      assert 6 == 9
E      + where 6 = square(3)
```

test_square.py:6: AssertionError

```
===== short test summary info =====
FAILED test_square.py::test_square - assert 6 == 9
===== 1 failed in 0.13s =====
```

PS C:\Users\witzmann\test2>

```
# calculator.py
def main():
    x = int(input("Gib eine Zahl ein: "))
    print(square(x))

def square(x):
    """Returns the square of x."""
    return x * x

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

```
# test_square.py
from calculator import square
import pytest

def test_square_positive():
    assert square(2) == 4
    assert square(3) == 9

def test_square_zero():
    assert square(0) == 0

def test_square_negative():
    assert square(-2) == 4
    assert square(-3) == 9

def test_square_string():
    with pytest.raises(TypeError):
        square("2")
```

PS C:\Users\witzmann\test2> **pytest** .\test_square.py

```
===== test session starts =====
platform win32 -- Python 3.8.5, pytest-6.1.1, py-1.9.0, pluggy-0.13.1
rootdir: C:\Users\witzmann\test2
plugins: cov-2.10.1, django-4.1.0
collected 4 items
```

test_square.py [100%]

```
===== 4 passed in 0.04s =====
```

PS C:\Users\witzmann\test2>

Aufbau eines Unit-Tests

1. Vorbereitung (Preparation oder **Arrange**)
2. Ausführung (Execution oder **Act**)
3. Prüfung (Verification oder **Assert**)

3A Modell (Arrange, Act, Assert)

Mocks, Mock patchen, Mockup

Best Practice

1. Ein Test sollte **einfach** gehalten werden
2. Ein Test prüft genau **eine Sache**

Hinweis: Eine Funktion sollte auch genau eine Sache machen. Aber eine Funktion braucht oft mehrere Tests.

3. Ein Test sollte **keine** unnötigen **Annahmen** machen
4. Ein Test sollte das **Was** und nicht das Wie prüfen (Blackbox)
5. Ein Test sollte **schnell** durchlaufen
6. Tests sollten möglichst **wenig Ressourcen** nutzen

Test-Driven Development

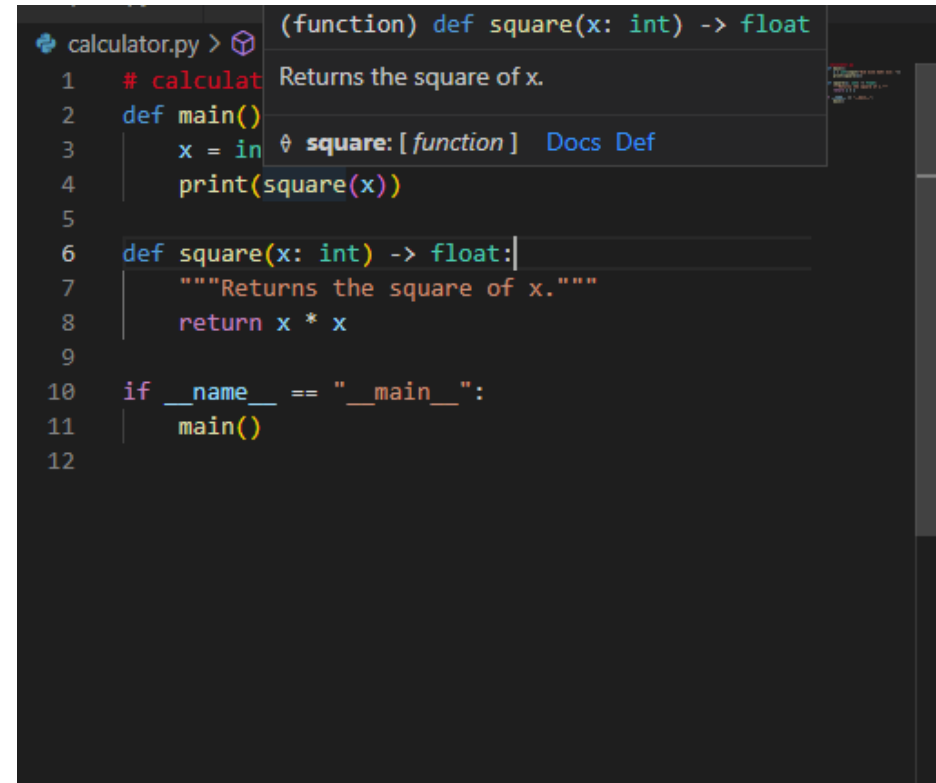
TDD

Dokumentation

Clean Code

- Sinn & Zweck
- Kommentare
- Lesbarkeit des Codes
- TMI
- Docstrings
- PEP
- black
- Hints zu Datentypen
- mypy DATEiname.py

Tipp: Schau dir die Clean Code Videos auf YouTube von Robert C . Martin „Uncle Bob“ an (mehrteilig):
<https://www.youtube.com/watch?v=7EmboKQH8IM>



```
calculator.py > (function) def square(x: int) -> float
1 # calculat Returns the square of x.
2 def main()
3     x = in  ↳ square: [function] Docs Def
4     print(square(x))
5
6 def square(x: int) -> float:
7     """Returns the square of x."""
8     return x * x
9
10 if __name__ == "__main__":
11     main()
12
```

```
PS C:\Users\witzmann\test2> mypy .\calculator.py
Success: no issues found in 1 source file
```