Programmierung mit Python

Grundlagen Grundbegriffe Grundübungen

Alfatraining Kurs 2026-05 Dozent: Karsten Flügge

Themen Heute

Tests

test driven development
TDR mit pytest

~bitoperator
string slicing []

Offene Themen / Spezialsyntax

```
Iteratoren + yield
immutable types
```

… Vertiefungen

… spezielle module (interne, externe) / Anwendungen

z.B. random, numpy (numerische Berechnungen, Matrizen und Vectoren),

Tests

Tests sind kleine Funktionen oder Programme welche die **Funktionalität** des geschriebenen Codes **überprüfen** / sicherstellen / **spezifizieren.**

Warum?

- **Korrektheit**
- **Sicherheit, Verlässlichkeit**
- **saubere Trennung:**
- **Spezifikation vs Implementation**

Statt seine Ergebnisse ad-hoc mit print() zu überprüfen, kann man sie lieber gleich '**formal**' absichern.

Tests

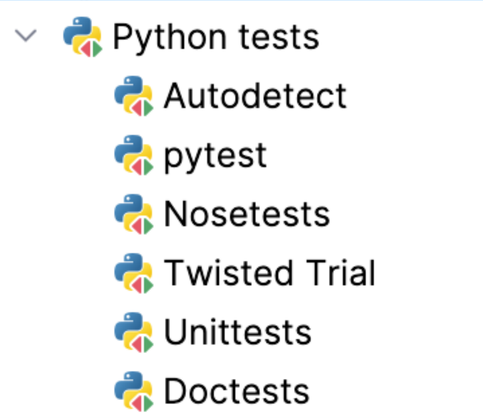
Tests sind kleine Methoden oder Programme welche die **Funktionalität** des geschriebenen Codes **überprüfen** / sicherstellen.

Was und Wie?

- einzelne Funktionen
- möglichst alle "**Fälle**" QA Quality Assessment
- Unit testing: Teste höheres Zusammenspiel der Komponenten

Tests

Es gibt verschiedene **Test frameworks.** In python sind das vor allem **pytest,** unittest, …



Wir benutzen hier pytest:

```
# cmd.exe> pip install pytest

# in Datei test_test.py :
import pytest

def test_test():
  assert True == True    # spezialkeyword 'stelle sicher'
```

Tests starten

Es gibt verschiedene möglichkeiten **Tests** zu **starten…**

System Kommandozeile (cmd.exe / Powershell / Terminal.app)

pip install pytest

test_test.py anlegen:

```
import pytest

def test_test():
   assert True == True
```

Befehl zum Starten im aktuellen Ordner: **pytest** oder **python -m pytest** (optionales Argument: Ordner oder einzelne Dateien)

pytest findet automatisch alle Dateien im aktuellen Ordner welche mit test_ beginnen:

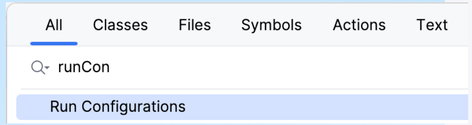
```
~/alfa/tag11/ pytest .
============================================== test session starts ==============================================
platform darwin -- Python 3.12.2, pytest-8.2.0, pluggy-1.5.0
rootdir: /Users/me/alfa/tag11
plugins: anyio-4.3.0
collected 2 items

test_module.py .                                                                                        [ 50%]
test_test.py .                                                                                          [100%]

=============================================== 2 passed in 0.00s ===============================================
```

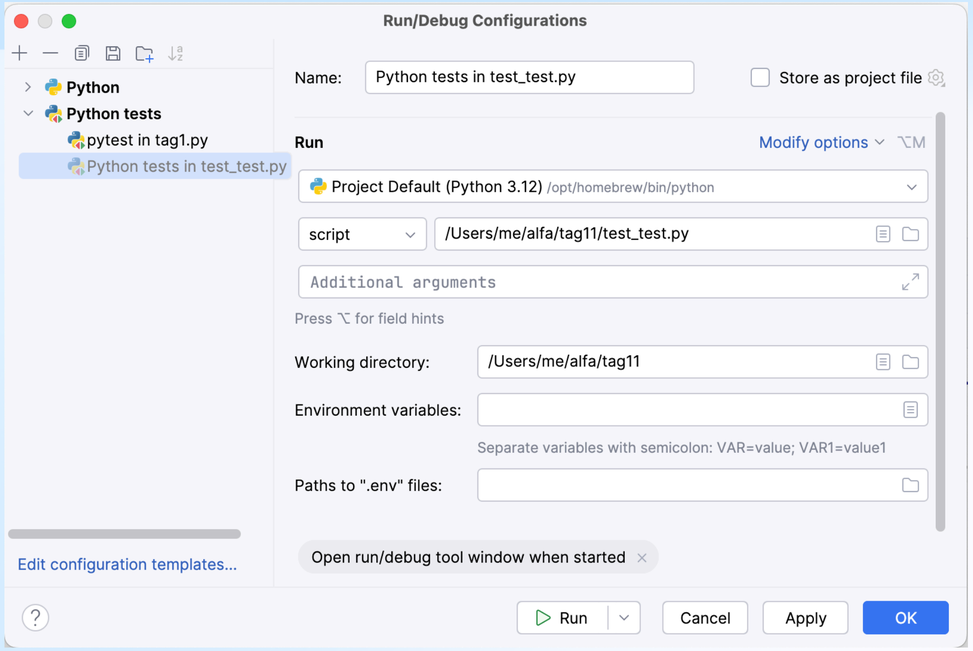
Tests starten

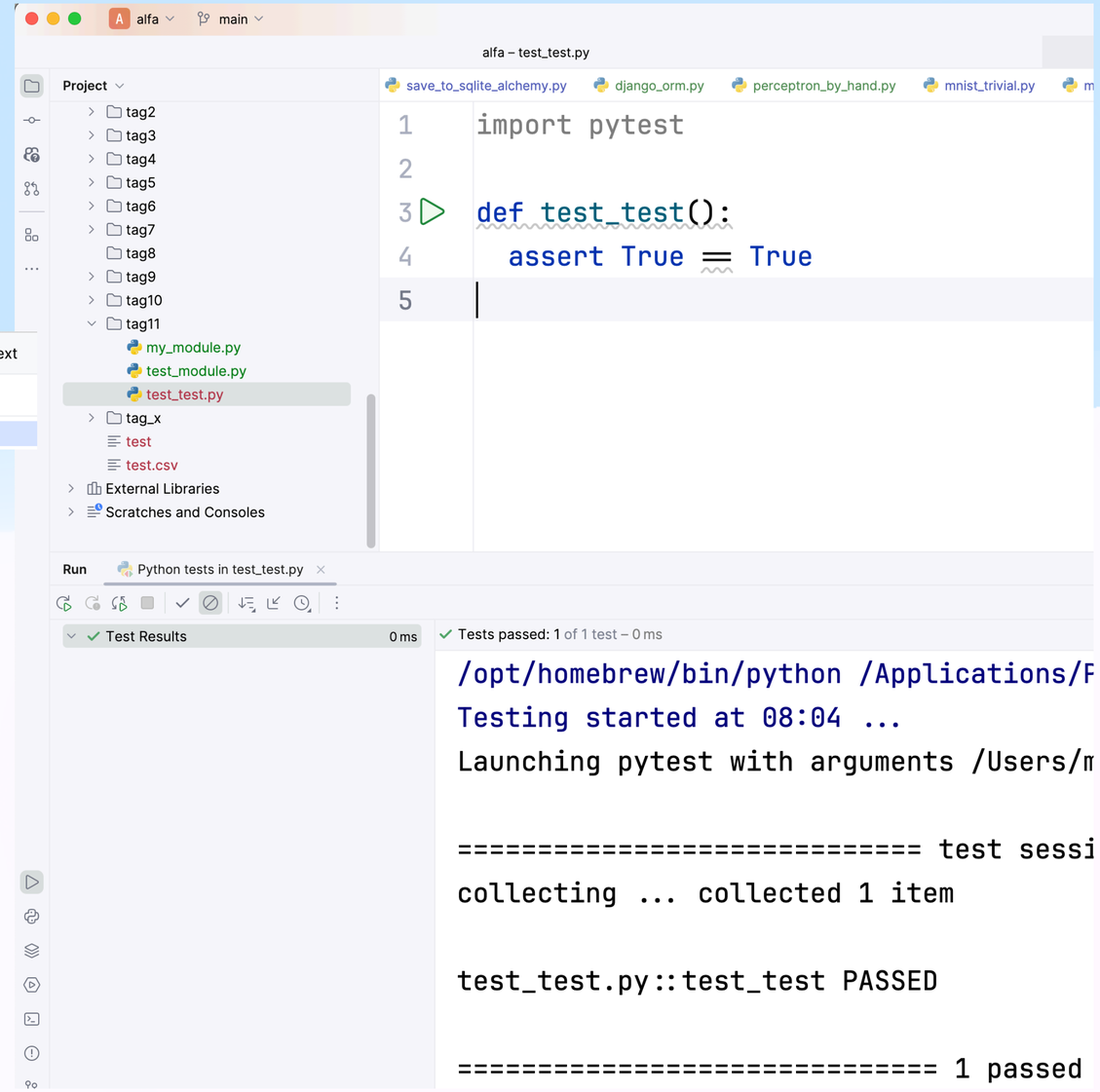
Es gibt verschiedene möglichkeiten **Tests** zu **starten…**

In der IDE (**Pycharm**):

- grüner Pfeil neben der test Funktion
- über **Suche -> Run Configurations**







Tests starten

Es gibt verschiedene möglichkeiten **Tests** zu **starten…**

In der IDE (**VSCode**):

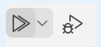
Links die **Flasche "Testing"**

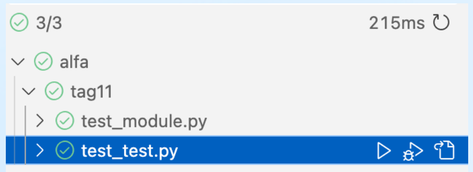
Dann **Configure Python Tests**

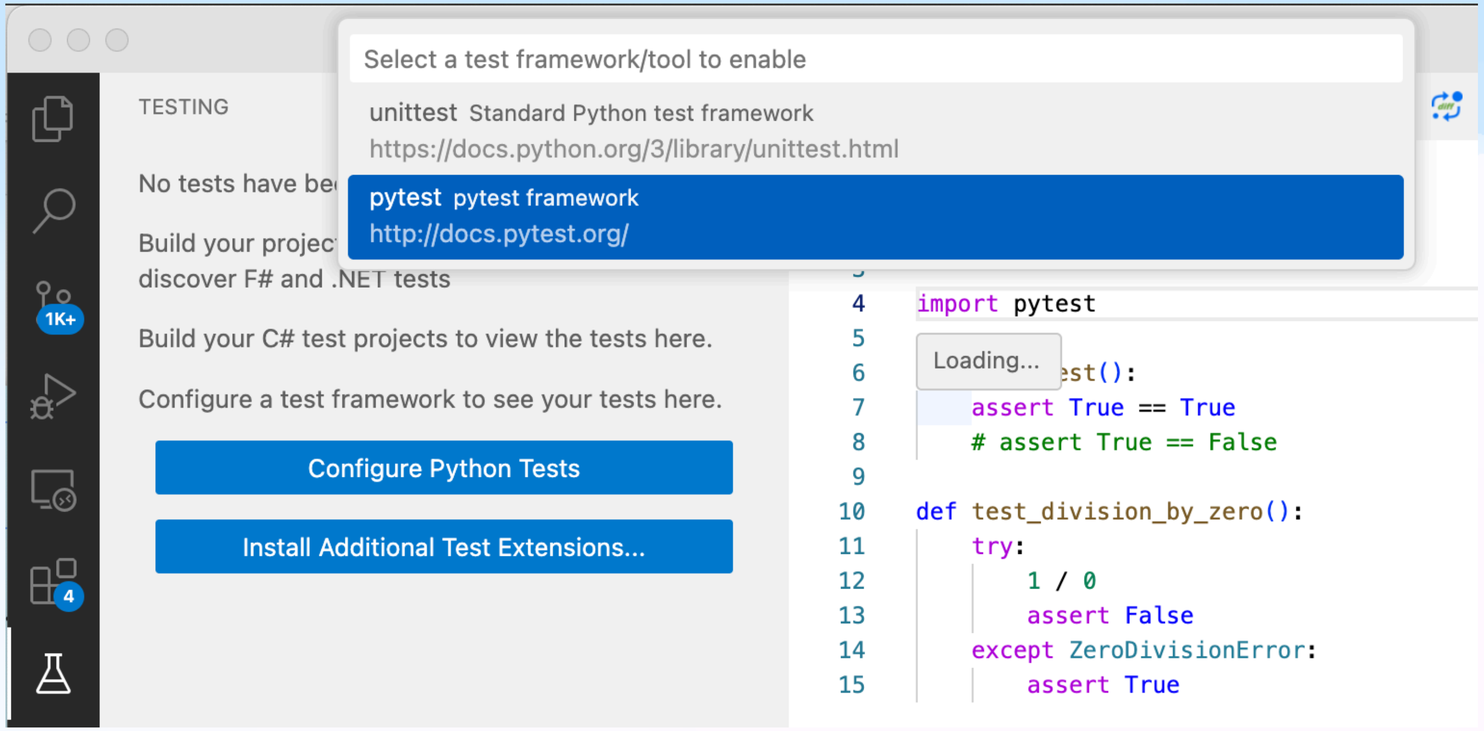
Dann **pytest auswählen**

Dann **den Ordner mit Tests**

und **kleiner Pfeil**







Tests von Exceptions

Um zu prüfen, ob eine Funktion **wie gewünscht** einen **Fehler** wirft, kann man den Funktionsaufruf mit `with pytest.raises(Exception): BLOCK` umhüllen. z.B.:

```
def division_by_zero(x,y):
  return x/y

def test_division_error():
  with pytest.raises(Exception):
    division_by_zero(3,0)
```

Test **OK**, division_by_zero wirft **wie erwartet** einen **Fehler!**

Ausdruck versus Anweisung

Ausdruck (Expression) : Wert oder Code Fragment oder Berechnung welches Wert/Ergebnis Erzeugt
Anweisung (Statement) : Komplette Code Zeile welche eine Aktion Ausführt, z.B. Zuweisung oder Aufruf()

```
sum = 3 + 1; # '3 + 1' ist ein Ausdruck, das ganze ist ein Statement ( Zuweisung )
```

Ausdruck (Expression) "das was auf der rechten Seite vom `=` steht"
Ausdruck (Expression) "das was in runden Klammer steht" print(9*9)

Arithmetic Expression:

```
sum = a + b; // sum is 8
wert = ( 1 + 2 ) * 3
```

Relational Expression:

```
isEqual = (a == b); // isEqual is false
kleiner10 = ( x < 10 )
```

Logical Expression:

```
jaOderNein = True or False;
ungleichZehn = x < 10 and x > 10
isEitherEven = (a % 2 == 0) || (b % 2 == 0); // isEitherEven is false
```

Function Call (Expression):

laenge = **len**("abc")

```
maxVal = max(a, b); // maxVal is 5
```

print("abc") # Statement !

Ausdruck versus Anweisung

Anweisung (statement)

Deklaration:
x:int;
def test():pass

Definition:
Kontroll Ausdruck:
if, if:else:, while, for(:), for(;;)
return Statement (in a function):

Compound Statement (block / body / Körper / Anweisungsliste)
if x: (…)

Ausdruck versus Anweisung

R-Value versus L-Value

not explicitly applicable because Python handles variables and memory management differently.

Ausdrücke (Expressions) teilen sich in R-Value ≠ L-Value
Namensgebung von right/left, aber Fehlbezeichnung “misnomer”
i = 3; # 3 ist ein r value 'rechts', i ist ein l value 'left'
j = i; # ⚠️ i ist immer noch ein l value!
j = i + 1; # ist wieder ein R-Value

r-value:
Ein temporärer oder nicht-adressierbarer Wert, der nicht über den Ausdruck, der ihn verwendet, hinaus bestehen bleibt.

Merke: "Ein R value hat keinen Namen"

l-value:
Ein Ausdruck, der auf ein Objekt verweist, das einen dauerhaften Speicherort im Speicher hat und daher adressierbar ist.

Merke: "Ein L value hat einen Namen"

Speichern / Laden

Texte (strings) und allgemeine Daten lassen sich in Dateien schreiben und lesen
Dafür können wir die open() Funktion auf folgende Weise nutzen:

```
file_name = "test"

file_out = open(file_name, "w") # "w" ≈ "write"
file_out.write("Hallo Welt")
file_out.close() # ⚠️ nicht vergessen, nach dem Schreiben die Datei schliessen!

file_in = open(file_name)  # "r" ≈ "read" optional
zeilen = file_in.readlines()

print(zeilen)


Aufgabe: was passiert wenn ich ein dict in eine Datei speichern möchte?



# pro tip / später : "wb" für write binary
```

Serialisieren / Speichern von dict als json

Die Ähnlichkeit zu json lässt sich zunutze machen, um dictionaries leicht zu speichern und zu laden:

```
import json

obj = {"name": "Max", "alter": 25}
file_name = "data.json"

s = json.dumps(obj) # object als String serialisieren
obj = json.loads(s) # object aus String laden


json.dump(obj, open(file_name, "w"))  # object als Datei serialisieren
obj = json.load(open(file_name))      # object aus Datei laden

print(obj)
```

json ok für einfache Daten, für komplexere nimm **pickle**

Syntaktischer Zucker

Syntaktischer Zucker

Bezeichnet Syntax, welche nicht notwendig ist, d.h. welche man auch mit anderen Mitteln umsetzen kann, aber welche das Leben schöner oder leichter macht.

z.B.

```
mein_bester_freund["alter"] = mein_bester_freund["alter"] + 1
mein_bester_freund["alter"] += 1    "in-place addition operator"
```

"read-only syntax" reicht meisst, die Syntax zu erkennen, muss man nicht selber aktiv tippen.

Offene Themen

try
immutable types
@dekaratoren
@dataclass
@classmethod

… Vertiefungen

… spezielle module / Anwendungen

json, sql

Aufgaben dictionary

- **Zusammenführen von zwei Dictionaries:** Schreibe eine Funktion, die zwei Dictionaries zusammenführt und dabei die Werte von übereinstimmenden Schlüsseln addiert
- **Umkehren der Schlüssel-Werte-Paare in einem Dictionary:** Erstelle eine Funktion, die die Schlüssel und Werte eines Dictionaries umkehrt.
- darf eine Datei (f = open(file_name)) als Schlüssel verwendet werden? warum(nicht)?
-