АКАДЕМИЈА ТЕХНИЧКО-ВАСПИТАЧКИХ СТРУКОВНИХ СТУДИЈА

ОДСЕК НИШ

PROGRAMSKI ALATI ZA RAZVOJ SOFTVERA

Vežba 3: Pisanje testova u Pythonu

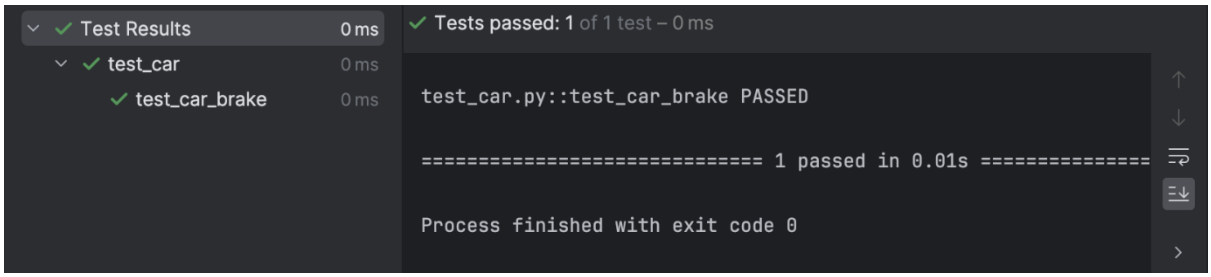
pytest je jedan od najpoznatijih framework-a za testiranje u Pythonu. Omogućava jednostavno i efikasno testiranje koda u Pythonu. To je alat koji omogućava automatsko izvršavanje testova, lako detektovanje grešaka i generisanje detaljnih izveštaja. Osnovna prednost **pytest**-a je njegova jednostavnost i fleksibilnost, jer omogućava pisanje testova koji se lako čitaju i održavaju.

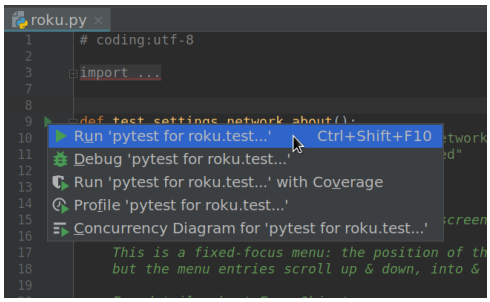
Glavne Karakteristike pytest-a

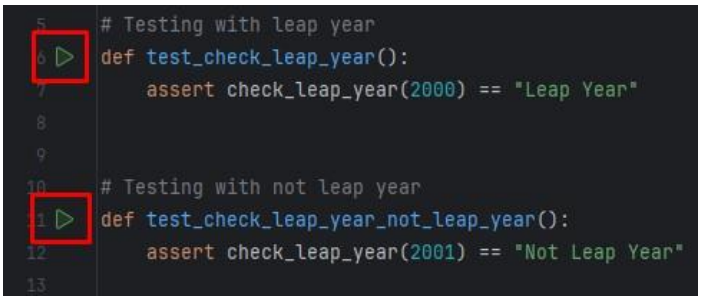
1. **Jednostavnost**:
    - Testovi su napisani u obliku običnih funkcija koje se nazivaju sa prefiksom **test**_.
    - Testovi mogu biti organizovani u hijerarhiji direktorijuma i fajlova.
2. **Prepoznavanje testova**:
    - **pytest** automatski prepoznaje testove. Sve funkcije koje počinju sa test_ automatski se prepoznaju kao test funkcije.
    - Takođe, klase koje počinju sa Test prepoznaju se kao test klase (iako to nije obavezno).
3. **Fleksibilnost i proširivost**:
    - Postoji mnogo dodatnih opcija i pluginova koji omogućavaju proširenje funkcionalnosti **pytest**-a.
    - Možete koristiti fixture za postavljanje podataka koji se koriste u više testova.
4. **Poboljšan izveštaj o grešci**:
    - Kada test ne prođe, **pytest** pruža detaljne informacije o grešci, uključujući vrednosti promenljivih u trenutku greške, što olakšava dijagnostiku problema.
5. **Kratkoća koda**:
    - Testovi u **pytest**-u obično su kraći i jednostavniji u poređenju sa nekim drugim framework-ovima (kao što je unittest).
6. **Kompatibilnost sa drugim framework-ovima**:
    - **pytest** je kompatibilan sa testovima napisanim u **unittest**-u, tako da možete koristiti i stare testove dok pređete na **pytest**.

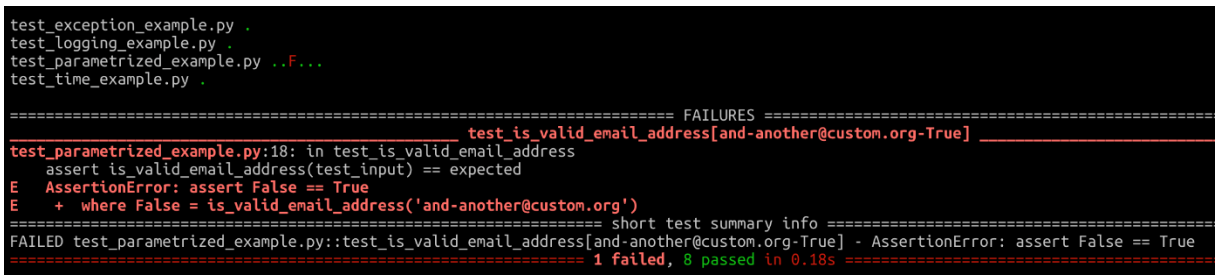
Instalacija pytest u PyCharm-u

1. **Instalacija paketa:** Da biste koristili pytest u svom Python projektu, potrebno je prvo da ga instalirate. Otvorite terminal u PyCharm-u i izvršite sledeću komandu: *pip install pytest*
2. **Kreiranje test projekta u PyCharm-u:**
    - Otvorite PyCharm i kreirajte novi Python projekat.
    - Zatim u tom projektu kreirajte direktorijum tests, gde će se nalaziti vaši testovi.
3. **Dodavanje fajlova u proejkat:**
    - Desnim klikom na direktorijum u kojem želite da dodate fajl (npr. na src ili tests), pojaviće se kontekstni meni.
    - U meniju koji se pojavi, izaberite New i zatim kliknite na Python File.
    - Za ovaj projekat dodaćemo order.py u src i u order_test.py u tests.









U PyCharm-u, možete pojedinačno pokrenuti testove tako što desnim klikom kliknete na ime test metode ili test klase, pa odaberete opciju **Run 'test_name'**. Takođe, možete pokrenuti sve testove unutar fajla klikom desnim tasterom na sam fajl i izborom opcije **Run 'All Tests in 'file_name'**.

Ako dođe do greške u testiranju, PyCharm će prikazati detalje o neuspehu, uključujući **expected** i **actual** vrednosti koje se ne poklapaju. Takođe, možete videti u logu da li je greška nastala zbog pogrešno napisanog testa (npr. greška u sintaksi) ili ako se očekivani rezultat razlikuje od stvarnog (npr. greška u testiranoj funkcionalnosti).

Primer koda:

U ovom primeru ćemo simulirati i testirati rad sa porudžbinama u nekoj e-trgovini.

Klasa **Order** predstavlja model za narudžbinu u e-trgovini. Njena osnovna funkcionalnost je da izračuna ukupnu cenu na osnovu cene po komadu proizvoda i količine. Klasa sadrži dva atributa: **price_per_item** (cena jednog proizvoda) i **quantity** (količina naručenih proizvoda).

Metoda **calculate_total()** računa ukupnu cenu porudžbine tako što množi cenu po komadu sa količinom proizvoda. Ova metoda se koristi za računanje u raznim situacijama, uključujući kada je količina 0, kada je cena negativna, ili kada je potrebna obrada nevalidnih ulaza.

```
class Order:
    def __init__(self, price_per_item, quantity):
        self.price_per_item = price_per_item
        self.quantity = quantity

    def calculate_total(self):
        """Izračunava ukupnu cenu narudžbine."""
        return self.price_per_item * self.quantity
```

Sada ćemo u klasi **TestOrder** napisati testove za metodu **calculate_total**. Testiraćemo različite slučajeve, kao što su pozitivni brojevi i provera da li metoda pravilno računa ukupan iznos za različite količine i cene.

```
import unittest

class TestOrder(unittest.TestCase):

    def test_calculate_total_positive(self):
        """Testira da li metoda pravilno računa iznos kada je cena i količina pozitivna"""
        order = Order(100, 3)  # Cena po komadu je 100, a količina 3
        self.assertEqual(order.calculate_total(), 300)  # Očekivana ukupna cena je 300

    def test_calculate_total_zero_quantity(self):
        """Testira da li metoda pravilno vraća 0 kada je količina 0"""
        order = Order(100, 0)  # Cena po komadu je 100, a količina je 0
        self.assertEqual(order.calculate_total(), 0)  # Ukupna cena treba biti 0
```

```
def test_calculate_total_negative_price(self):
    """Testira kako metoda reaguje na negativnu cenu"""
    order = Order(-100, 3)  # Cena po komadu je -100, a količina je 3
    self.assertEqual(order.calculate_total(), -300)  # Očekivana ukupna cena je -300


def test_calculate_total_invalid_input(self):
    """Testira kako metoda reaguje na nevalidne ulaze"""
    order = Order("100", 3)  # Cena je string, a ne broj
    with self.assertRaises(TypeError):  # Očekivana greška: TypeError
        order.calculate_total()
```

assertEqual(actual, expected): Ovaj metod se koristi da uporedimo stvarnu vrednost (actual) sa očekivanom vrednošću (expected). Ako se vrednosti ne poklapaju, test će pasti.

Testiranje grešaka sa assertRaises: Ako očekujemo da neka metoda baci izuzetak (kao na primer u test_calculate_total_invalid_input), koristimo **assertRaises**. Na ovaj način se proverava da li je određeni izuzetak bačen u toku izvršavanja testa.

Evo još nekoliko korisnih **assert** metoda koje se koriste u testiranju:

1. **assertTrue(condition)**: Ovaj metod proverava da li je uslov (argument condition) **True**. Ako nije, test će pasti. Na primer, može se koristiti za proveru da li je neka vrednost istinita ili da li se funkcija ponaša kao što se očekuje.

   ```
   self.assertTrue(order.calculate_total() > 0)
   # Proverava da li je ukupna cena veća od 0
   ```

2. **assertFalse(condition)**: Ovaj metod proverava da li je uslov (argument condition) **False**. Ako nije, test će pasti. Slično kao assertTrue, ali ovde očekujemo da uslov bude netačan.

   ```
   self.assertFalse(order.calculate_total() < 0)
   # Proverava da li ukupna cena nije manja od 0
   ```

3. **assertIsNone(object)**: Ovaj metod proverava da li je objekat **None**. Koristi se kada želimo da testiramo da li neki rezultat nije dodeljen ili nije inicijalizovan.

   ```
   self.assertIsNone(order.calculate_total())  # Proverava da li metoda vraća None
   ```

4. **assertIsNotNone(object)**: Ovaj metod proverava da li objekat **nije** None. Koristi se kada očekujemo da vrednost nije None, što znači da je metoda uspešno izvršila neku operaciju.

   ```
   self.assertIsNotNone(order.calculate_total()) # Proverava da li vraća valid vrednost
   ```

5. **assertAlmostEqual(a, b, places=7)**: Ovaj metod upoređuje dve vrednosti sa određenim brojem decimalnih mesta (zadato sa places, podrazumevano je 7). Koristi se kada radimo sa brojevima koji se mogu zaokružiti ili koji sadrže decimale, a nije bitno da budu potpuno identični, već samo skoro identični.

```
self.assertAlmostEqual(order.calculate_total(), 300.0001, places=4)
 # Upoređuje sa zaokruživanjem
```

6. **assertGreater(a, b)**: Ovaj metod proverava da li je vrednost a veća od vrednosti b. Ako nije, test će pasti.

```
self.assertGreater(order.calculate_total(), 100)
# Proverava da li je ukupna cena veća od 100
```

7. **assertLess(a, b)**: Ovaj metod proverava da li je vrednost a manja od vrednosti b. Ako nije, test će pasti.

```
self.assertLess(order.calculate_total(), 500)
# Proverava da li je ukupna cena manja od 500
```

8. **assertIn(item, container)**: Ovaj metod proverava da li je neka stavka (item) prisutna u kontejneru (npr. listi, setu, stringu). Ako nije, test će pasti.

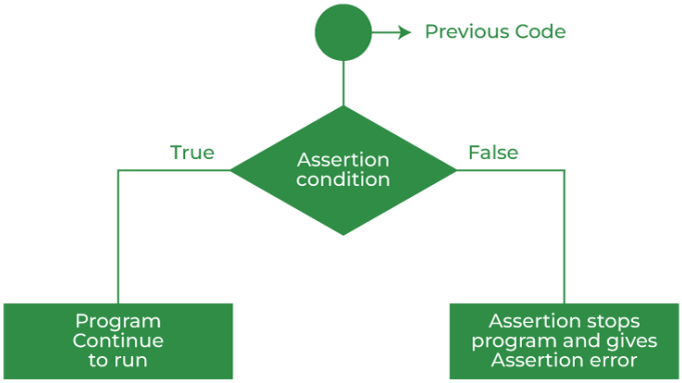
```
self.assertIn('item', order.get_items())
# Proverava da li je 'item' u listi proizvoda narudžbine
```

9. **assertNotIn(item, container)**: Ovaj metod proverava da li stavka (item) **nije** prisutna u kontejneru. Ako jeste, test će pasti.

```
self.assertNotIn('item', order.get_items())
 # Proverava da li 'item' nije u listi proizvoda
```

10. **assertCountEqual(list1, list2)**: Ovaj metod proverava da li dve liste sadrže iste elemente u istom broju, nezavisno od njihovog redosleda. Razlikuje se od assertEqual jer ne upoređuje redosled elemenata.

```
self.assertCountEqual(order.get_items(), ['item1', 'item2', 'item3'])
# Proverava sadrži li lista tačno te stavke
```



Zadatak za samostalni rad

Napravite klasu **StudentResult** koja prati rezultate studenta na ispitima. Klasa bi trebalo da omogući unos ocena, kao i prosek i status studenta (da li je položio sve ispite). Implementirajte ovu klasu i napišite pytest testove za njene metode. Poželjno je imati oko 10 testova. Za odbranu vežbe donosite projekat u .zip datoteci i screenshot koji pokazuje da su svi testovi prošli.

Koraci:

1. **Implementirajte klasu StudentResult sa sledećim funkcionalnostima:**
    - **Inicijalizacija**: Prilikom kreiranja objekta, klasa treba da prima ime studenta i praznu listu ocena.
    - **Dodavanje ocene**: Napravite metodu **add_grade(grade)**, koja prima ocenu kao celobrojnu vrednost (npr. 6, 7, 8, 9, 10). Ako ocena nije između 6 i 10, metoda treba da baci ValueError.
    - **Prosečna ocena**: Napravite metodu **average()**, koja vraća prosečnu ocenu kao decimalnu vrednost. Ako student nema ocena, metoda treba da vrati None.
    - **Status položenih ispita**: Napravite metodu **all_passed(),** koja proverava da li su svi ispiti položeni (sve ocene moraju biti 6 ili više). Ako student nema nijednu ocenu, metoda treba da vrati False.
2. **Napišite pytest testove za klasu StudentResult:**
    - Proverite da li metoda add_grade() ispravno dodaje ocene u listu i baci ValueError za nevalidne ocene.
    - Proverite da li metoda average() ispravno računa prosek ocena i vraća None kada student nema ocena.
    - Proverite da li metoda all_passed() ispravno vraća True ili False u zavisnosti od ocena.
    - Testirajte granične slučajeve, kao što su ocena tačno 6 i tačno 10, kao i dodavanje više ocena odjednom.

Primer testova kakvi mogu da budu

```
import pytest
from student_result import StudentResult


def test_add_grade_valid():
    student = StudentResult("Petar")
    student.add_grade(8)
    assert student.grades == [8]
```

```
def test_add_grade_invalid():
    student = StudentResult("Ana")
    with pytest.raises(ValueError):
        student.add_grade(5)

def test_average_no_grades():
    student = StudentResult("Jovan")
    assert student.average() is None
```

Primer implementacije klase

```
class StudentResult:
    def __init__(self, name):
        self.name = name
        self.grades = []

    def add_grade(self, grade):
        # Proverite da li je ocena između 6 i 10
        if grade < 6 or grade > 10:
            raise ValueError("Ocena mora biti između 6 i 10.")
        self.grades.append(grade)
```

Metode average() i all_passed() implementirajte sami na osnovu priloženog.

Korisni resursi

Jednostavno vežbanje assert-a

https://www.datacamp.com/tutorial/understanding-the-python-assert-statement

Odlično objašnjenje za pisanje i pokretanje testova

https://www.jetbrains.com/help/pycharm/pytest.html

Dobar „poligon za vežbanje“

https://coderefinery.github.io/testing/exercises/

Detaljan i koristan tutorijal

https://www.youtube.com/watch?v=cHYq1MRoyI0

Solidni tutorijali i vežbanja

https://learn.microsoft.com/en-us/training/modules/test-python-with-pytest/