

# Testiranje jedinica

Sa ovom lekcijom započinjemo rad u oblasti testiranja programa; krećemo od prvog nivoa testiranja, gde testiramo najmanje delove (jedinice) našeg koda.

Testiranje jedinice obuhvata proveru najmanjih celina koda koje se mogu testirati, nazvanih jedinice (units). Provera je nezavisna, individualna i bez uticaja ostalih komponenata poput drugog koda, baze podataka ili mreže. U ovoj proceduri se jedinice sistema testiraju na validnost tokom kompletnog procesa razvoja; dakle, jednom napisan test koristimo da u fazama daljeg razvoja možemo prekontrolisati da li jedinica i dalje pravilno radi.

Ovo ujedno predstavlja i poseban pristup razvoju sofvera pod nazivom *test driven development*, gde se rad ne završava napisanom funkcijom programa, već i svim testovima koji proveravaju njenu ispravnost.

Testiranje jedinice je white box oriented tehnika testiranja, gde, naravno, imamo uvid u kod i pravimo direktne promene u samoj strukturi programa. Jediničnim testiranjem proverava se da li pojedinačne komponente ispravno funkcionišu sa svim očekivanim tipovima ulaza, u skladu sa dizajnom komponente.

Testiranje jedinice podrazumeva pristup kodu koji se testira sa podrškom alata i uključuje programere koji su pisali kod. Programer koji je pisao programsku rutinu ili klasu vrši i prvu proveru samog koda: radi pregled koda, proverava funkcionalnost, performanse i u dogovoru sa kolegama u timu optimizuje kod, proverava konvenciju o imenovanju i dr.

Pre svega, hajde odmah da definišemo šta je to jedinica. Pod jedinicom možemo posmatrati jedan:

- modul;
- klasu;
- metodu;
- objekat;
- funkciju.

Dakle, proces testiranja se obavlja samo na jednoj izdvojenoj programskoj celini. Najbolji način pristupa unit testovima jeste da krenemo od najmanje celine koju možemo testirati. Nakon toga testiramo druge jedinice i proveravamo kako one vrše interakciju. Na ovaj način dolazimo do detaljnih testova koje možemo koristiti za klasu ili modul.

Zašto vršimo unit testiranje?

- Unit testiranje je važno najviše zbog toga da ne bismo novim funkcionalnostima projekta poremetili postojeće funkcionalnosti.
- Unit testiranje treba sprovoditi nakon izmena u kodu.
- Unit testovi bi trebalo da budu u sastavu projekta, ali dovoljno odvojeni da možemo da ih po potrebi isključimo iz procesa prevođenja.

Rekli smo da u unit testu proveravamo da li metode imaju očekivano ponašanje tako što ih startujemo, a zatim proverimo da li su na osnovu unetih parametara vratile očekivane vrednosti. Pogledajmo kako se sličan proces obavlja ukoliko ne koristimo unit testove.

Uzećemo jednu jednostavnu funkciju, čija svrha je da sabere dva broja i vrati rezultat na mesto poziva:

```
def sumFromNumbers(a,b):  
    return a+b
```

Dakle, ovo je jedna naša programska celina, jedinica (unit). Sada želimo da utvrdimo da li je ishod rada funkcije tačan. Ovde, dakle, formiramo jedan test slučaj; taj test slučaj treba da nam pomogne da proverimo tačnost rezultata. Stoga, dodaćemo sledeće:

```
firstNumber = 2  
secondNumber = 3  
expectedResult = 5  
  
result = sumFromNumbers(firstNumber,secondNumber)
```

Dakle, definisali smo prvi i drugi broj koji će se koristiti unutar funkcije i jednu odvojenu promenljivu koja će predstavljati naš očekivani rezultat rada funkcije. Pored ovoga, u odvojenoj promenljivoj result čuvamo poziv funkcije sa prosleđenim prethodno definisanim brojevima.

Dolazimo do pisanja testa. Dakle, sada nam je potrebno da uporedimo rezultate rada programa sa očekivanim rezultatom za date brojeve. Pa stoga možemo iskoristiti jednu if else strukturu:

```
if result == expectedResult:  
    print("Test passed")  
else:  
    print("Test failed")
```

Kao što možete videti, jednostavno proveravamo da li je vrednost promenljive result jednaka vrednosti promenljive expectedResult. U zavisnosti od toga, ispisujemo da li je program prošao na testu ili pao. Ovo je primer jednog manuelnog testa funkcije; možemo reći da je ovo unit test, jer radimo provere jedne jedinice, u ovom slučaju funkcije, i pratimo njene ishode rada i njih upoređujemo sa nekom referentnom očekivanom vrednošću.

Napravite funkciju prvo\_slovo koja kao argument prima string i vraća prvi karakter datog stringa. Proverite da li je povratna vrednost funkcije jednaka vrednosti promenljive expectedResult koja se unosi preko input funkcije. Ako jeste ispisati "Slova su ista", u suprotnom slučaju ispisati "Slova nisu ista".

## Biblioteke za razvoj testova

Složit ćete se, ovim pristupom sve moramo pisati ručno, svaki put definisati uslove, napisati strukture i ishode. Sa vremenom se ova neefikasnost u radu uočila i za skoro svaki jezik je

kreirana biblioteka ili u nekim slučajevima čitav framework koji je posvećen bržem i standardizovanom pisanju testova. Python je upravo jedan od jezika koji imaju odličnu podršku za testiranje. Ovaj jezik u okviru svojih standardnih biblioteka sadrži i paket za testiranje jedinica (unit testing). Takođe, za Python postoji eksterna biblioteka/framework za razvoj testova pod nazivom `pytest`.

Krenućemo prvo od integrisane biblioteke Pythona i pokazati kako pišemo test za Python kod.

Biblioteka nosi naziv `unittest` i predstavlja framework za razvijanje testova inspirisan drugim poznatim radnim okvirima poput JUnit i kao takav vrlo je sličan kao u ostalim programskim jezicima. Podržava automatizaciju testova, kolekcije i zavisnosti, što je takođe nešto o čemu ćemo pričati tokom kursa.

Pre nego što nastavimo dalje, potrebno je da spomenemo i neke pojmove testiranja jedinica koji su opšte korišćeni u programiranju, a to su:

- **Fixture (test fixture)** – Fixture predstavlja pripremu testnog okruženja. Ova priprema može uključivati sve od kreiranja posebnih baza podataka, direktorijuma do kreiranja posebnih serverskih procesa itd. Naravno, sve zavisi od konteksta aplikacije, ali suštinski, predstavlja podešavanje kompletnog okruženja tako da se test može izvesti bez uticaja testa na ostatak koda, ili suprotno, tako da neki drugi kod ne utiče na rezultate testa. Dobro organizovan fixture testa omogućava lak prelaz sa testiranja koda na njegovu upotrebu.
- **Case (test case)** – Kao što smo se upoznali ranije, case je slučaj ili jedinica testiranja. Test slučaj smo u prethodnim lekcijama prikazali kao proces dokumentacije, gde pratimo moguće ishode; ovo takođe važi i unutar unit testa, gde pišemo pažljivo isplanirane slučajeve. `unittest` pruža i dodatnu klasu `TestCase`, koja se koristi za definisanje novih test slučajeva.
- **Suite (test suite)** – Suite predstavlja kolekciju slučajeva. Služi nam za to da smestimo testove vezane za jednu jedinicu i omogućava da ih pokrenemo sve u isto vreme.
- **Runner (test runner)** – Runner predstavlja aplikaciju koja pokreće/startuje test. Runner aplikacija može imati grafički korisnički interfejs, ali i konzolni/terminal interfejs.

Sada smo se upoznali sa osnovnim terminima i spremni smo da napišemo svoj prvi unit test. Uzećemo primer sa početka lekcije i promeniti pristup na QOP. Kao što smo govorili i ranije, fajl koji drži klasu bi trebalo sačuvati pod istim nazivom kao i samu klasu. Kod koji ćemo koristiti možete videti u nastavku.

```
class Calculator:
    def sumFromNumbers(self, firstNumber, secondNumber):
        result = firstNumber + secondNumber
        return result

calc = Calculator()
print(calc.sumFromNumbers(2, 3))
```

Dakle, imamo jedan funkcionalan primer realizovan pomoću klase; sada želimo da testiramo metodu `sumFromNumbers` i njenu tačnost. U tu svrhu korišćićemo prvo klasičan metod testiranja, sa dva fajla – jednim koji drži klasu i drugim koji drži test. Nakon toga, upotrebićemo i `unittest` framework.

Postupak pisanja prvog testa jedinica započinjemo fajlom koji čuva test. Ono što je važno kod test driven development pristupa jeste: test se nikada ne piše u istom fajlu gde je kod koji testiramo. Testovi se najčešće smeštaju u projektni folder, ali u odvojenom folderu, pod odvojenim fajlom. Pa stoga, neka novi fajl nosi naziv TestCalculator.py.

U okviru ovoga fajla, obavićemo prvenstveno import klase i definisaćemo jednu funkciju koja će instancirati klasu i pokrenuti metodu. **Napomena:** Iz fajla koji drži samu klasu ukloniti instanciranje klase i ispis rezultata. Razlog ovoga jeste što će to sada obaviti u okviru test fajla. Test fajl sada izgleda ovako:

```
from Calculator import Calculator

def test_Calculator():
    calc = Calculator()
    print(calc.sumFromNumbers(2,3))
```

Ovim postizemo da u ovom novom fajlu, koji će služiti za test, dobijamo rezultat rada metode koju želimo da proverimo. Sledeći korak je da napišemo proveru rezultata. Kao i ranije, znamo da je očekivan izlaz 5, pa to i proveravamo:

```
from Calculator import Calculator

def test_sumFromNumbers():
    calc = Calculator()
    result = calc.sumFromNumbers(2,3)
    if result == 5:
        print("Test passed")
    else:
        print("Test failed")

test_sumFromNumbers()
```

Kada pokrenemo ovaj kod, dobijamo ispis *Test passed* i potvrdu da smo dobili očekivani izlaz. Test koji je jednom napisan na ovaj način uvek čuvamo i kako radimo na klasi, proširujemo je novim metodama i slično. S vremena na vreme, pokrećemo ovaj test da proverimo da li je sve i dalje tačno.

## unittest framework

Nastavimo sa radom na prethodnom primeru, ali ovog puta test realizujemo kroz unittest framework. Prvi korak je da obavimo import unittest paketa. Ovaj paket je podrazumevano ugrađen u Python, pa je stoga dovoljno samo iskoristiti komandu import (`import unittest`).

**Napomena:** Ceo prikazani proces se odvija u okviru .py fajla u kojem je sačuvan test:

```
import unittest
from Calculator import Calculator

def test_sumFromNumbers():
    calc = Calculator()
    result = calc.sumFromNumbers(2,3)
    if result == 5:
```

```

        print("Test passed")
    else:
        print("Test failed")

test_sumFromNumbers()

```

Sada dolazimo do prvog pojma u okviru unittest frameworka, a to je test case, ili slučaj koji proveravamo. Ovaj slučaj je potrebno držati u okviru klase, zbog nasleđivanja logike iz frameworka. Na našem primeru, to izgleda ovako:

```

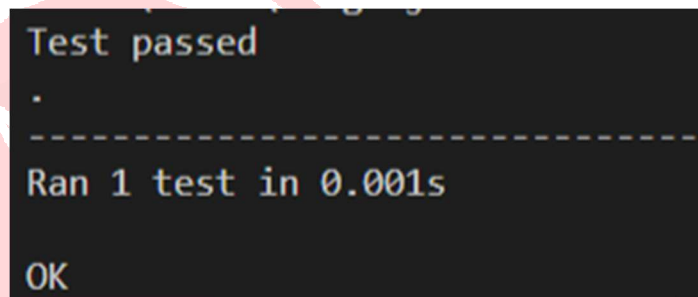
import unittest
from Calculator import Calculator

class testSumFromNumbers(unittest.TestCase):
    def test_sumFromNumbers(self):
        calc = Calculator()
        result = calc.sumFromNumbers(2,3)
        if result == 5:
            print("Test passed")
        else:
            print("Test failed")

unittest.main()

```

Dakle, sada imamo klasu `testSumFromNumbers` koja nasleđuje `unittest.TestCase`. Ovo će reći Pythonu da, ukoliko unutar klase postoji metoda čiji naziv počinje sa `test`, ta metoda je ona koju ćemo koristiti kao slučaj testiranja. U našem slučaju, to je metoda `test_sumFromNumbers`, koja sada ima `self` parametar. I na kraju našeg koda, umesto direktnog poziva metode, sada imamo poziv `main` metode modula `unittest`. Ovo će pokrenuti našu metodu i sam framework za obradu testa. Ako pokrenemo kod, dobijamo rezultat kao na slici u nastavku:



```

Test passed
.
-----
Ran 1 test in 0.001s

OK

```

*Slika 5.1. Rezultat pokretanja koda*

Kao što možete videti, sada imamo dva prikaza rezultata. Prvi je iz `if-else` strukture naše metode, gde smo dobili ispis da je test uspešno prošao, a nakon isprekidane linije vidimo ispis iz `unittest` frameworka, gde je navedeno koliko je testova izvršeno i koliko je trajalo vreme testa, i dobijamo statusnu poruku `OK` – dakle, da je kod uspešno izvršen.

Nama je važno da iskoristimo potencijal frameworka i da koristimo njegove naredbe za sprovođenje testa, jer na taj način ukidamo uticaj programera na samu logiku testa.

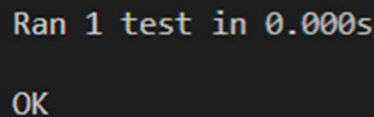
Više nećemo koristiti strukture za provere uslova, već integrisane komande frameworka, pa je sledeći korak izmena našeg koda na sledeće:

```
import unittest
from Calculator import Calculator

class testSumFromNumbers(unittest.TestCase):
    def test_sumFromNumbers(self):
        calc = Calculator()
        result = calc.sumFromNumbers(2,3)
        self.assertEqual(result,5)

unittest.main()
```

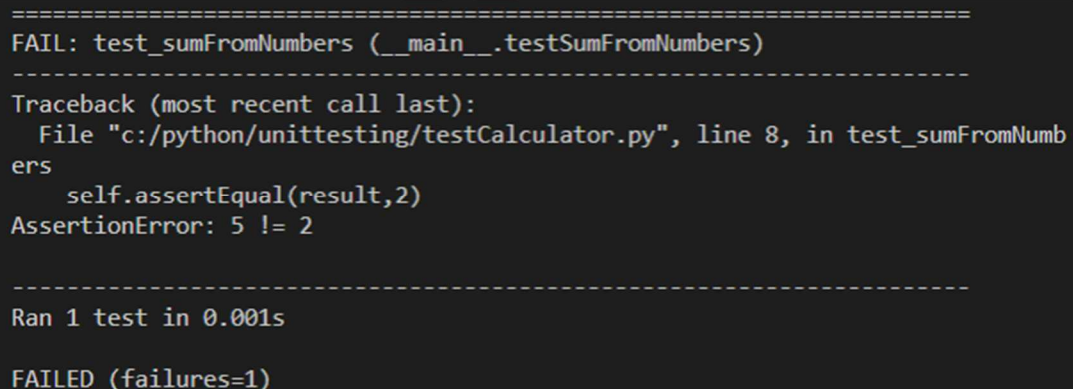
Dakle, sada smo kompletnu if-else strukturu zamenili sa: `self.assertEqual(result,5)`. `assertEqual` je samo jedna od desetina naredbi koje možemo koristiti prilikom pisanja testa. U ovom slučaju, `assertEqual` proverava da li je rezultat jednak očekivanom rezultatu. Ukoliko je kod prošao test, kao ispis dobijamo:



```
Ran 1 test in 0.000s
OK
```

Slika 5.2. Prikaz ispisa prilikom uspešnog testa

U slučaju da test nije uspeo, dobijamo mnogo detaljniji izveštaj:



```
=====
FAIL: test_sumFromNumbers (__main__.testSumFromNumbers)
-----
Traceback (most recent call last):
  File "c:/python/unittesting/testCalculator.py", line 8, in test_sumFromNumbers
    self.assertEqual(result,2)
AssertionError: 5 != 2
-----

Ran 1 test in 0.001s

FAILED (failures=1)
```

Slika 5.3. Prikaz rezultata prilikom neuspešnog testa

Na prvoj liniji ispisa dobijamo poruku da test nije prošao i navedenu metodu u okviru koje je test izvršen. Na drugoj liniji dobijamo informaciju o kojem fajlu je reč i gde je tačno došlo do greške na testu. Kako se navodi linija 8, reč je o našoj proveru rezultata. Odmah sledi i podatak zašto je test pao:

```
self.assertEqual(result,2) AssertionError: 5 != 2
```

Dakle, ovde smo sa namerom postavili pogrešnu pretpostavku, tj. da rezultat treba da bude 2, iako znamo da je rezultat 5. Stoga dobijamo poruku AssertionError: 5 != 2, koju možemo pročitati kao: Greška u pretpostavci, 5 nije jednako 2.

I za kraj, kao i ranije, dobijamo potvrdu da je test izvršen i da nije prošao. Sa ovim smo uradili jedan unit test pomoću unittest frameworka.

Assert metode su metode za proveru određenog stanja (uslov/pretpostavka/tvrdnja). Do sada smo koristili metodu assertEquals, koja proverava da li jedna vrednost odgovara drugoj. Ali, takođe, postoje i druge assert metode, koje možete videti u narednoj tabeli:

Method	Checks that
assertEqual(a, b)	a == b
assertNotEqual(a, b)	a != b
assertTrue(x)	bool(x) is True
assertFalse(x)	bool(x) is False
assertIs(a, b)	a is b
assertIsNot(a, b)	a is not b
assertIsNone(x)	x is None
assertIsNotNone(x)	x is not None
assertIn(a, b)	a in b
assertNotIn(a, b)	a not in b
assertIsInstance(a, b)	isinstance(a, b)
assertNotIsInstance(a, b)	not isinstance(a, b)
assertRaises(exc, fun, args, *kwds)	fun(*args, **kwds) raises exc
assertRaisesRegexp(exc, r, fun, args, *kwds)	round(a-b, 7) == 0
assertAlmostEqual(a, b)	round(a-b, 7) == 0
assertNotAlmostEqual(a, b)	round(a-b, 7) != 0
assertGreater(a, b)	a > b
assertGreaterEqual(a, b)	a >= b
assertLess(a, b)	a < b
assertLessEqual(a, b)	a <= b
assertRegexpMatches(s, r)	r.search(s)
assertNotRegexpMatches(a, b)	not r.search(s)
assertItemsEqual(a, b)	sorted(a) == sorted(b)
assertDictContainsSubset(a, b)	All the key/value pairs in a exist in b

*Tabela 5.1. Prikaz svih assert metoda i njihovih funkcija*

Pogledajmo kako može izgledati formiranje više assert naredbi u okviru jednog testa:

```
import unittest
from Calculator import Calculator

class testSumFromNumbers(unittest.TestCase):
    def test_sumFromNumbers(self):
        calc = Calculator()
        result = calc.sumFromNumbers(2,3)
```

```

        expected = 5
        self.assertEqual(result, expected)
        self.assertNotEqual(result, expected)
        self.assertGreaterEqual(result, expected)

unittest.main()

```

Kao što možete videti, proverili smo prvo pretpostavku da li su očekivani i dobijeni rezultat jednaki, što jeste slučaj. Zatim da nisu jednaki, što će vratiti pad testa, tj. grešku. Nakon toga, imamo treći assert, gde smo rekli da rezultat mora biti veći ili jednak, što je takođe ispunjeno. Dakle, važno je znati da test neće biti prekinut sve dok se ne provere svi uslovi.

### Kreiranje grupe testova (test suite)

Do sada smo radili samo sa jednim testom, dakle jednom metodom koja drži test slučaj. Često u slučaju kompleksnih aplikacija dolazi do potrebe da se testovi grupišu da bi bili startovani svi odjednom ili po tačno zacrtanom redosledu. U ovu svrhu, koristi se klasa `TestSuite`, pa pogledajmo kako izgleda primer kreiranja test suitea:

```

import unittest
from Calculator import Calculator

class testSumFromNumbers(unittest.TestCase):
    def test_sumFromNumbers(self):
        calc = Calculator()
        result = calc.sumFromNumbers(2,3)
        expected = 5
        self.assertEqual(result, expected)
        self.assertNotEqual(result, expected)
        self.assertGreaterEqual(result, expected)

def suite():

    # gather all tests in a test suite

    test_suite = unittest.TestSuite()
    test_suite.addTest(unittest.makeSuite(testSumFromNumbers))
    return test_suite

mySuite=suite()

#define Runner

runner=unittest.TextTestRunner()
runner.run(mySuite)

```

Na primeru možete videti da dodajemo jednu novu funkciju. Ova funkcija će držati logiku kreiranja skupa testova. Na prvoj liniji kreiramo novu promenljivu u okviru koje instanciramo klasu `unittest.TestSuite()`. Svrha ove klase je da omogući kreiranje jednog skupa. Nakon toga, u narednoj liniji, pomoću naredbe `addTest` dodajemo našu test metodu u skup. U narednom koraku je dovoljno samo da vratimo vrednost `test_suite` promenljive na mesto poziva. U novoj promenljivoj pozivamo funkciju, a ona će u promenljivu `mySuite` sačuvati novokreirani test suite.



Krajnji korak je kreiranje runnera. Kao što smo na samom početku naveli, runner predstavlja aplikaciju ili interfejs koji pokreće/startuje test. U našem slučaju, korist ćemo runner da pokrenemo ceo test suite. Kao što smo naveli, runner aplikacija može imati grafički korisnički interfejs, ali i konzolni/terminal interfejs. Kako je nama dovoljan prikaz rezultata, u konzoli kreiramo runner tog tipa sa linijom:

```
runner=unittest.TextTestRunner()
```

Nakon izvršavanja ove linije, koristimo metodu run() da pokrenemo kompletan test suite.

Naravno, ovaj test suite ima malo poente, jer imamo samo jednu metodu, koja ima tri test slučaja, ali ako bismo jednostavno kopirali metodu još dva puta, da dobijemo ukupno tri metode sa nekim svojim test slučajevima, imali bismo realniju potrebu za skupom testova. Čisto primera radi, to bi moglo da izgleda ovako sa klasom Calculator u istom fajlu:

### Radno okruženje

```
import unittest

class Calculator:
    def sumFromNumbers(self, firstNumber, secondNumber):
        result = firstNumber + secondNumber
        return result

class testSumFromNumbers(unittest.TestCase):
    def test_isCorect(self):
        calc=Calculator()
        result = calc.sumFromNumbers(2,3)
        expected = 5
        self.assertEqual(result,expected)

    def test_isNegative(self):
        calc=Calculator()
        result = calc.sumFromNumbers(-4,1)
        expected = 0
        self.assertLess(result,expected)

    def test_isGreater(self):
        calc = Calculator()
        result = calc.sumFromNumbers(2,3)
        expected = 5
        self.assertGreaterEqual(result,expected)

def suite():
    # gather all tests in a test suite

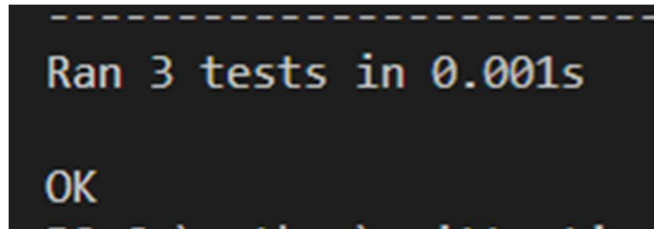
    test_suite = unittest.TestSuite()
    test_suite.addTest(unittest.makeSuite(testSumFromNumbers))
    return test_suite

mySuite=suite()
```

```
#define Runner

runner=unittest.TextTestRunner()
runner.run(mySuite)
```

Dakle, sada runner interfejsom pokrećemo test klasu koja ima tri metode, od kojih svaka ima svoj poseban test slučaj. Kada pokrenemo kod, uočićemo da su se izvršila tri testa.



*Slika 5.4. Rezultat pokretanja test suitea*

## pytest

pytest je još jedan od testing frameworka za Python koji ćemo prikazati u kontekstu unit testova. Svrha pytesta je slična ugrađenom unittest paketu. Dakle, glavni cilj joj je olakšavanje pisanja testova u Pythonu i koristi se za sve od unit testova, preko integracionih, do testiranja baza podataka i API testiranja.



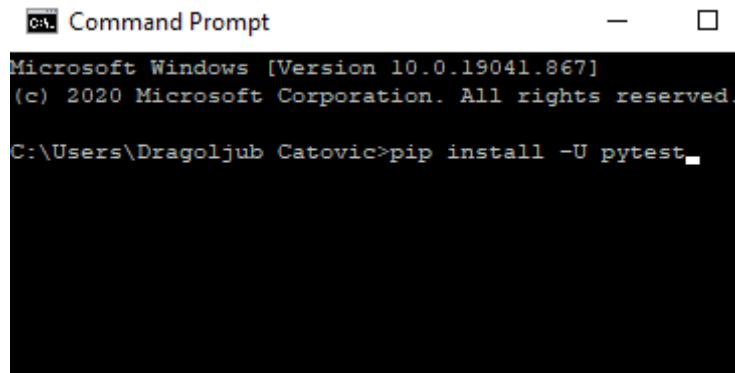
*Slika 5.5. Logo pytest frameworka*

Sada se možda pitate – ako unittest već omogućava testiranje u Pythonu, zašto imamo potrebu za pytest frameworkom? Glavne prednosti ovog frameworka su:

- pytest je lakši za korišćenje usled jednostavnije sintakse;
- testovi se mogu izvršavati paralelno;
- može se pokrenuti samo jedan test iz fajla sa testovima;
- automatski prepoznaje fajlove koji sadrže testove, kao i same testove unutar njih;
- postoji mogućnost preskakanja izvršavanja testova.

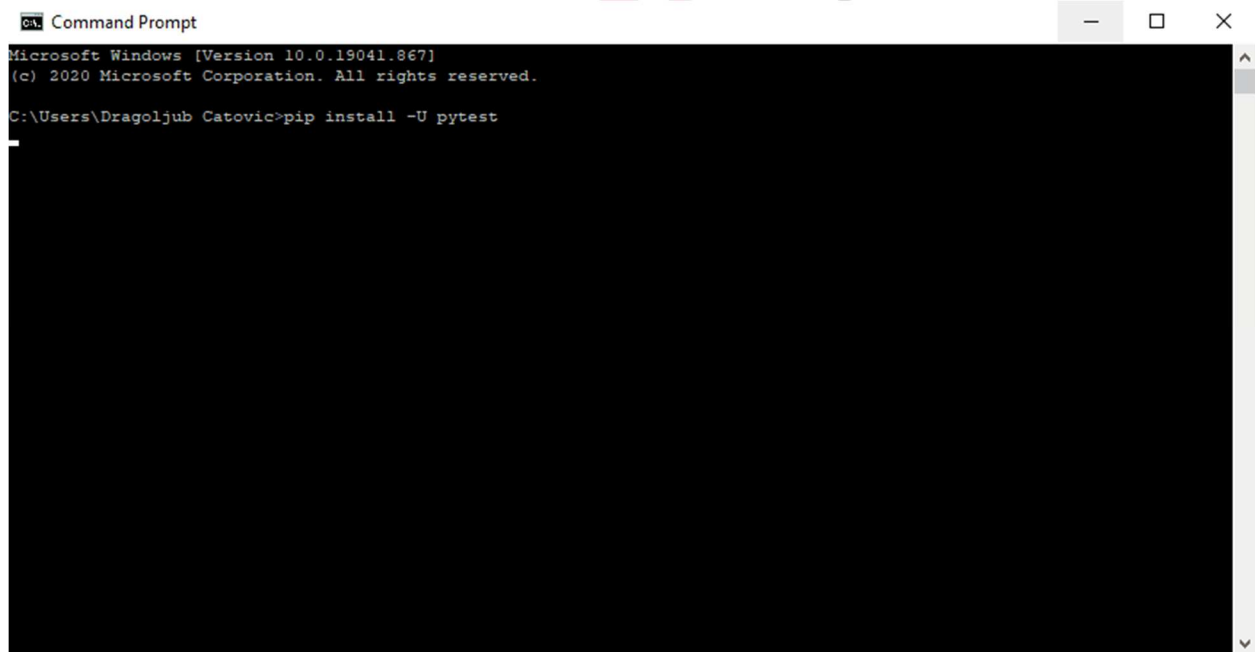
## Instalacija pytest frameworka

U prvom koraku preuzimamo i instaliramo sam framework u Python; stoga se očekuje da imate ažurnu instalaciju Pythona na svom računaru. Kada je to slučaj, dovoljno je pokrenuti narednu komandu u komandnoj liniji (konzoli) operativnog sistema: `pip install -U pytest`



*Slika 5.6. Komanda za instalaciju pytest frameworka*

Nakon pokretanja komande, počinje instalacija frameworka i postupak se završava potvrdom instalacije. Slično obaveštenje možete videti u sledećoj animaciji:



*Animacija 5.1. Izvršavanje komande za instalaciju pytesta*

Nakon potvrde o uspešnoj instalaciji, možemo još jednom proveriti da li je sve instalirano. Ovo postićemo komandom: `pytest --version`

Za razliku od unittest frameworka, vrlo moćna opcija ovoga frameworka jeste automatska detekcija fajlova i metoda (funkcija) koji sadrže testove. Ovo se postiže prilikom nazivanja fajlova, ali i blokova koda unutar njih.

Za sve fajlove važi pravilo: svi fajlovi u aktuelnom direktorijumu koji imaju prefiks **test\_** ili sufiks **\_test** biće automatski uzeti u obzir za testiranje prilikom pokretanja komande pytest, koja započinje izvršavanje testova.

Kada pytest pristupi fajlu označenom kao fajl koji sadrži testove, biće izvršene sve funkcije/metode koje u svom nazivu kao prefiks imaju reč test. Pokretanje testova se realizuje kroz konzolnu komandu pytest u okviru direktorijuma unutar koga se nalaze test fajlovi. Pogledajmo to na primeru:

Kako možemo kroz konzolu doći do foldera u kojem su testovi? Najlakše nam je da kreiramo folder u samom korisničkom folderu našeg operativnog sistema. Ovaj folder se najčešće nalazi na putanji C:\Users\Username, gde Username predstavlja naziv vašeg korisničkog naloga, i ovo je upravo folder koji nam je potreban. Razlog je to što konzole uvek kreću sa ove putanje i na taj način najlakše dolazimo do potrebnih foldera. U okviru foldera, kreiraćemo novi folder; npr. neka nosi naziv tests.

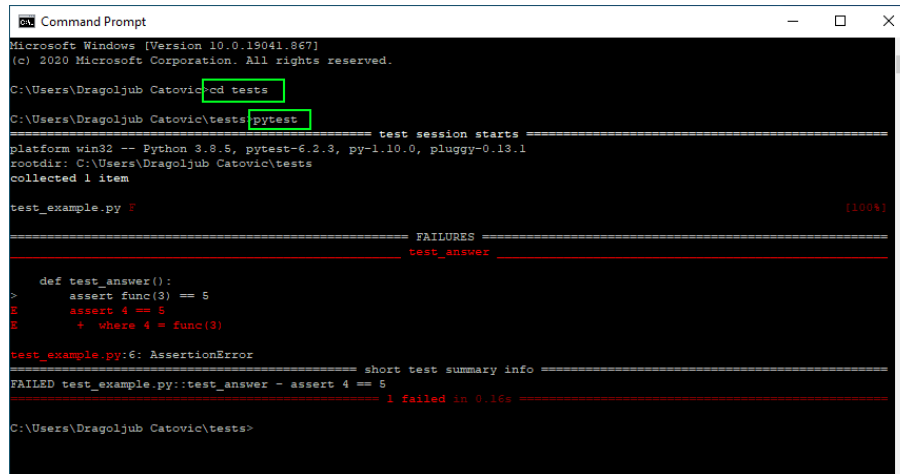
Nakon toga, unutar foldera kreiraćemo jedan novi fajl, neka se zove test\_example.py. U ovom novom fajlu, postavimo sledeći kod:

```
def func(x):  
    return x + 1  
  
def test_answer():  
    assert func(3) == 5
```

Malo detaljnije o samom kodu govorili smo u nastavku, ali sada želimo da pokrenemo test da proverimo da li sve radi pravilno. Pa stoga, otvorimo konzolu operativnog sistema i sada, ukoliko ste sačuvali folder i fajl prema prethodno navedenom postupku, dovoljno je da unesete komandu:

```
cd tests
```

pomoću koje pristupate folderu tests, a nakon toga komandom pytest pokrećete sve testove u folderu. Kako mi imamo samo jednu funkciju sa prefiksom test\_, izvršiće se samo jedan test. Ovaj postupak možete videti na sledećoj slici:



```
Microsoft Windows [Version 10.0.19041.867]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\Users\Dragoljub Catovic>cd tests
C:\Users\Dragoljub Catovic\tests>pytest
===== test session starts =====
platform win32 -- Python 3.8.5, pytest-6.2.3, py-1.10.0, pluggy-0.13.1
rootdir: C:\Users\Dragoljub Catovic\tests
collected 1 item

test_example.py F [100%]

===== FAILURES =====
test_answer

  def test_answer():
>     assert func(3) == 5
E       assert 4 == 5
E        + where 4 = func(3)

test_example.py:6: AssertionError
===== short test summary info =====
FAILED test_example.py::test_answer - assert 4 == 5
===== 1 failed in 0.16s =====

C:\Users\Dragoljub Catovic\tests>
```

Slika 5.7. Prikaz testa izvršenog kroz pytest framework

Kao što možete videti, nakon izvršavanja komande `pytest` dobijamo znatno detaljniji prikaz nego što je to slučaj sa izveštajem `unittest` frameworka.

U prvoj sekciji, pod nazivom `test session starts` dobijamo podatke o tome koji se folder sa testovima koristi, kao i koliko je testova prepoznato u njemu. Ovaj podatak nam daje linija: `collected 1 item`. Odmah u narednoj liniji dobijamo i naziv samog fajla, kao i oznaku da je u okviru njega jedan od testova pao.

Sljedeća sekcija je `Failures`. U okviru ove sekcije dobijamo prikaz funkcije kojom smo vršili test i tačno mesto gde je test pao. U narednoj sekciji dobijamo manje detaljan prikaz, gde samo vidimo razlog zbog kojeg je test pao.

Vratimo se sada na kod našeg primera:

```
def func(x):
    return x + 1

def test_answer():
    assert func(3) == 5
```

Kao što možete videti, funkciju koja drži test case smo nazvali sa prefiksom `test_` i na ovaj način smo omogućili `pytest` frameworku da je automatski prepoznaje. Takođe, u kodu možete uočiti da koristimo naredbu `assert`, slično kao i u `unittest` frameworku, ali ovde nemamo metode objekta `assert` poput `assertEqual`, već izraz koji proveravamo pišemo direktno u okviru naredbe, u ovom slučaju operatorom `==`.

Osim jednostavnijeg pristupa definisanju prepostavki u testu, značajna je opcija da `assert` ostavlja korisnički definisanu poruku u samom izveštaju. Možda se pitate – zašto je ovo važno?

Ukoliko pogledamo izveštaj koji smo dobili od `pytest`a, možemo videti da je test pao i da je greška u određenom `assertu`:

FAILED test\_example.py::test\_answer - assert 4 == 5

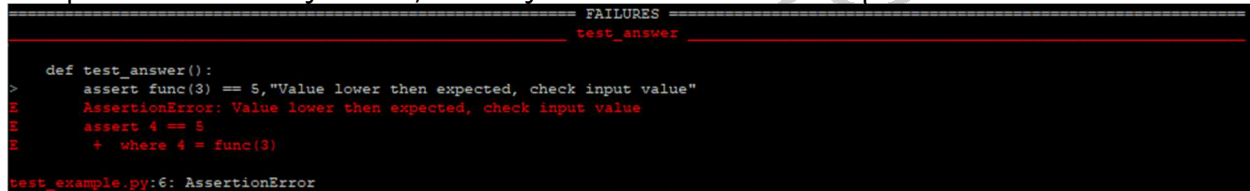
Ali da bismo znali šta je problem, moramo otići u funkciju koju smo testirali, da pogledamo koji je podatak ušao, kako je nastala promena i zašto nismo dobili očekivani rezultat. Ukoliko promenimo kod primera na sledeći način:

```
def func(x):  
    return x + 1  
  
def test_answer():  
    assert func(3) == 5, "Value lower then expected, check input value"
```

Sada smo iskoristili mogućnost dodavanja poruke, prema sintaksi:

assert izraz , "message"

Ako pokrenemo izmenjeni kod, izveštaj će sada u delu Failures prikazati sledeće:



```
===== FAILURES =====  
test_answer  
  
def test_answer():  
> assert func(3) == 5, "Value lower then expected, check input value"  
E       AssertionError: Value lower then expected, check input value  
E       assert 4 == 5  
E       + where 4 = func(3)  
test_example.py:6: AssertionError
```

Slika 5.8. Failures sekcija izvršenog testa

Nama je važna nova linija izveštaja:

AssertionError: Value lower then expected, check input value

Sada, pored toga što dobijamo obaveštenje u kojoj assert komandi dolazi do greške, dobijamo i našu poruku sa kojom možemo sebe usmeriti na određeni deo koda gde je greška mogla da nastane.

### Pitanje

Koji pojam predstavlja aplikaciju koja pokreće/startuje test?

- **Runner**
- Case
- Suite

### Objašnjenje:

*Runner predstavlja aplikaciju koja pokreće/startuje test. Ova aplikacija može imati grafički korisnički interfejs, ali i konzolni/terminal interfejs.*

## Kreiranje test klase

Kao što smo i ranije govorili, za grupisanje testova u jednu celinu koristimo test klasu. Uzećemo naš primer Calculator i za njega napisati test klasu, ali sada u okviru pytest frameworka.

Fajl calculator.py ostaje nepromenjen:

```
class Calculator:
    def sumFromNumbers(self, firstNumber, secondNumber):
        result = firstNumber + secondNumber
        return result
```

dok sada za potrebe testiranja kreiramo novi fajl pod nazivom test\_calculator.py; još jednom, važno je da imamo prefiks ili sufiks nazivu u vidu test\_ ili \_test.

U okviru fajla, sada možemo do dodamo našu logiku testova; ona bi mogla da izgleda ovako:

```
from calculator import Calculator

class TestCalculator:

    def test_isCorect(self):
        calc = Calculator()
        result = calc.sumFromNumbers(2,3)
        expected = 5
        assert result == expected

    def test_isNegative(self):
        calc = Calculator()
        result = calc.sumFromNumbers(-4,1)
        expected = 0
        assert result < expected

    def test_isGreater(self):
        calc = Calculator()
        result = calc.sumFromNumbers(2,3)
        expected = 5
        assert result >= expected
```

Sada su delovi koji se tiču pretpostavki test slučaja znatno jednostavniji; važno je obratiti pažnju na nazive klase i metoda. Kao što smo ranije naveli, klasa mora započeti rečju Test da bi je framework prepoznao.

```

from calculator import Calculator

class TestCalculator:
    def test_isCorrect(self):
        calc = Calculator()
        result = calc.sumFromNumbers(2,3)
        expected = 5
        assert result == expected

    def test_isNegative(self):
        calc = Calculator()
        result = calc.sumFromNumbers(-4,1)
        expected = 0
        assert result < expected

    def test_isGreater(self):
        calc = Calculator()
        result = calc.sumFromNumbers(2,3)
        expected = 5
        assert result >= expected

```

Slika 5.9. Pravilno nazivanje klase i metoda test fajla

U ovoj lekciji ste imali priliku da se upoznate sa dva frameworka za unit testiranje. Izbor frameworka će uvek zavisiti od obima projekta. unittest je lakši za korišćenje kada imamo par test fajlova i broj testova izražen u desetinama. Sa druge strane, pytest se, sa svojom automatizacijom prepoznavanja testova, može koristiti za veće projekte, pa čak i testiranje sistema. Upravo ovu primenu ćete videti u narednoj lekciji, gde govorimo o integracionim testovima.

### Vežba 1:

U nastavku je prikazan program koji obračunava zapreminu kvadra:

```

def cuboid_volume(a,b,c):
    return (a*b*c)

```

Korišćenjem unittest frameworka, u okviru test klase definisati test slučajeve za:

1. Ukoliko je vrednost ispod nule. (Nisu dozvoljene negativne vrednosti stranica figure.)
2. Ukoliko jedna od stranica za vrednost ima string. (Dozvoljene su samo numeričke vrednosti.)

**Napomena:** U okviru tabele 5.1. ove lekcije pronađite assert metodu koja najviše odgovara slučaju. Za test slučaj 2 pratiti grešku u tipu podataka i koristiti metodu koja odgovara ovom slučaju.

### Rešenje:



```

import unittest

def cuboid_volume(a,b,c):
    return a*b*c

print(cuboid_volume(10,10,10))

class testVolume(unittest.TestCase):
    def test_isNegative(self):
        #result = cuboid_volume(10,-2,10)
        #test nece proci zato sto je rezultat negativan

        result = cuboid_volume(10,2,10)

        self.assertGreater(result,0)
        #if the result is greater than 0 / test passed

    def test_isString(self):
        #result = int(cuboid_volume(10,"test",10))
        #test nece proci zato sto pokusavamo da nesto sto je tipa string
        prebacimo u celobrojni tip

        result = int(cuboid_volume(10,2,10))

        self.assertRaises(TypeError,result,True)
        # If the result variable contains type Error drop test wrong input,
        int expected

    def suite():
        # gather all tests in a test suite

        test_suite = unittest.TestSuite()
        test_suite.addTest(unittest.makeSuite(testVolume))
        return test_suite

mySuite=suite()

#define Runner
runner=unittest.TextTestRunner()
runner.run(mySuite)

```

**Objašnjenje:** Prvi test slučaj koristi tvrdnju `assertGreater`, jer želimo da proverimo da li je vrednost koja se čuva u promenljivoj `result` veća od nula. Ukoliko vrednost jeste veća od nula, to znači da su korišćeni pozitivni brojevi i test je prošao.

Za drugi test slučaj, obratite pažnju na to da u promenljivoj `result` prilikom poziva funkcije želimo da dobijemo vrednost u obliku integera. Otuda i upotreba funkcije `int`. Kako je string trenutno u ulaznim parametrima funkcije, dobićemo `Type Error` prilikom izvršavanja; umesto pisanja obrade izuzetka, dovoljno je da iskoristimo tvrdnju `assertRaises`, koja proverava da li postoji `TypeError` u promenljivoj `result`. Ukoliko je to slučaj, test pada i to je indikator da u ulaznim parametrima funkcije nije došla prava vrednost.

## Rezime

- Testiranje jedinice obuhvata proveru najmanjih celina koda koje se mogu testirati, nazvanih jedinice ili unita. Jednom napisan unit test koristimo da u fazama daljeg razvoja možemo prekontrolisati da li jedinica i dalje pravilno radi.
- Testiranje jedinice je white box oriented tehnika testiranja, gde imamo uvid u kod i pravimo direktne promene u samoj strukturi programa.
- Pod jedinicom možemo posmatrati modul, klasu, metodu, objekat ili funkciju.
- Python je jedan od jezika koji imaju odličnu podršku za testiranje. Ovaj jezik u okviru svojih standardnih biblioteka sadrži i paket za testiranje jedinica (unit testing). Takođe, za Python postoji eksterna biblioteka/framework za razvoj testova pod nazivom `pytest`.
- Biblioteka za testiranje jedinica integrisana u Python nosi naziv `unittest` i predstavlja okruženje za razvoj testova.
- Test Fixture predstavlja pripremu testnog okruženja tako da se test može izvesti bez uticaja testa na ostatak koda ili suprotno, tako da neki drugi kod ne utiče na rezultate testa. Dobro organizovan fixture testa omogućava lak prelaz sa testiranja koda na njegovu upotrebu.
- Test case je slučaj ili jedinica testiranja. Test slučaj unutar unit testa predstavlja pažljivo isplanirane pretpostavke o ishodu rada programa.
- Test suite predstavlja kolekciju test slučajeva. Služi nam da smestimo testove vezane za jednu jedinicu i omogućava da ih pokrenemo sve u isto vreme.
- Test runner predstavlja aplikaciju koja pokreće/startuje test. Runner aplikacija može imati grafički korisnički interfejs, ali i konzolni/terminal interfejs.

