**Testare Unitara folosind Python**

# Despre Testarea Unitara

“Testarea”, in termeni generali de programare reprezinta procesul de a scrie cod (separat de codul aplicatiei) ce invoca o parte din codul pe care il testeaza pentru a determina aparitia unor erori. Testarea nu releva faptul ca aplicatia este scrisa corect, ci doar determina daca anumite conditii sunt tratate corect. Ce anume poate scoate la iveala testarea automata?

Erorile de sintaxa sunt greseli ce apar din folosirea incorecta a limbajului de programare, cum ar fi un punct in plus in “my\_list..append(foo)”. Greselile de logica apar cand algoritmul folosit este incorect. Poate programatorul a uitat ca Python foloseste ca index pozitia zero si incearca sa printeze ultimul caracter dintr-un string, scriind “print(my\_string[len(my\_string)]), ceea ce cauzeaza eroarea “IndexError”. Multe alte erori sistemice pot aparea ce pot fi verificate folosind testarea unitara. Poate ca programul se intrerupe de fiecare data cand utilizatorul introduce un numar mai mare decat 100 sau poate ca ingheata daca web site-ul pe care incearca sa-l acceseze nu este valabil.

Toate aceste erori pot fi evidentiate prin testarea codului. Testarea unitarea, in mod specific, testeaza o singura unitate de cod izolata. O unitate de cod poate fi un intreg modul, o singura clasa sau o functie. Este important faptul ca se izoleaza codul testat de restul, pentru a prinde erori specifice.

Caracteristicile de baza ale procesului de testare unitara includ urmatoarela aspecte:  
• fiecare test valideaza o functionalitate de la nivelul aplicatiei;  
• testele trebuie sa ruleze rapid, sa fie scurte si usor de inteles;  
• sa nu fie necesare configurari suplimentare.

# Framework-ul PyUnit

PyUnit este un framework pentru testare unitara, similar cu [JUnit](https://junit.org/) (testare unitara pentru limbajul Java). Permite testarea automata, distributia codului de initializare sau eliberare a resurselor pentru teste si agregarea testelor in colectii.

Testarea unitara realizata prin intermediul framework-ului PyUnit (modulul unittest) are in vedere urmatoarele concepte: test fixture, test case, test suite si test runner. Pregatirile necesare pentru realizarea testelor, dar si actiunile de eliberare a resurselor sunt grupate sub numele de test fixture.

Un test sau test case reprezinta o unitate de testare care verifica raspunsul pentru un set de valori de intrare specificate. Modulul unittest ofera o clasa de baza, numita TestCase, clasa care poate fi utilizata pentru a implementa teste. Agregarea testelor care trebuie sa fie rulate impreuna se realizeaza prin intermediul unei colectii de teste unitare cunoscuta sub numele de test suite.

Componenta care orchestreaza executia testelor si ofera rezultatul utilizatorului porta numele de test runner. Poate rula in interfata grafica sau in mod text, si poate returna o valoare pentru a indica rezultatul executiei testelor.

Modulul unittest ofera o colectie importanta de unelte ce permit constructia si rularea de teste. O unitate de testare/evaluare este creata prin derivarea clasei TestCase din modulul unittest.

Metode individuale pentru evaluarea functiilor de la nivelul modulului hello sunt introduse la nivelul clasei de test TestHello. Prin conventie, denumirea metodelor de test incepe cu termenul test\_. In acest fel este instiintata si componenta care orchestreaza executia testelor si ofera rezultatul utilizatorului (test runner) cu privire la metodele de test.

Punctul central al fiecarui test este un apel al unor metode, precum: assertEqual() pentru a verifica un rezultat asteptat, assertTrue() sau assertFalse() pentru a verifica o conditie, sau assertRaises() pentru a verifica daca este generata o anumita exceptie. Astfel de metode sunt utilizate pentru ca test runner-ul sa poata obtine rezultatele rularii testelor si sa genereze un raport.

# Organizarea testelor

Unitatile de baza din testarea unitara poarta numele de test cases si reprezinta scenarii individuale care trebuie initializate si evaluate. O astfel de unitate de testare este o instanta a clasei unittest.TestCase.

Codul de testare al unei instante TestCase trebuie să fie complet autonom, astfel incat sa poata fi executat fie in mod izolat, fie in combinatie arbitrara cu orice numar de alte unitati de testare. Cea mai simpla subclasa TestCase va implementa pur si simplu o metoda de testare (a carei denumire incepe cu test) pentru a executa un cod de testare specific.

Cum testele pot fi numeroase, iar configurarea lor poate fi repetitiva, exista posibilitatea izolarii codului de initializare intr-o metoda specifica, numita setUp(), pe care cadrul de testare o va solicita automat pentru fiecare test pe care il executam.

In mod similar, putem implementa o metoda tearDown() care se executa dupa executia fiecarei metode de test. Un astfel de mediu de lucru stabilit la nivelul codului de test poarta numele de fixture.

Executia metodelor de test se face in ordinea alfabetica a numelor acestora. Mai multe instante de tip TestCase pot fi grupate in functie de caracteristicile testate. Modulul unittest ofera un mecanism pentru gruparea mai multor elemente de tip TestCase prin intermediul clasei TestSuite.

In marea majoritate a cazurilor, apelul lui unittest.main() va determina colectarea tuturor instantelor de tip TestCase de la nivelul unui modul. Cu toate acestea, daca se doreste implementarea unui astfel de grup (suite) de teste, acest lucru poate fi realizat si manual.

Definitiile unitatilor de testare si grupurile de testare pot fi introduse si la nivelul modulelor ce introduc codul care urmeaza a fi testat, dar sunt o serie de avantaje cu privire la plasarea acestora in module separate:

• modulul de testare poate fi rulat individual din linia de comanda;  
• codul de test trebuie modificat mult mai rar fata de codul pe care il evalueaza;  
• codul de test poate fi mult mai usor separat de codul testat;  
• daca strategia de testare se modifica, nu este necesara modificarea codului sursa.

# Exemplificare a Testarii

Pentru a evidentia modul de folosire a modulului unittest in testarea codului Python, am ales cateva functii simple, scrise in python care ne ajuta sa aplicam si anumite edge-case-uri. Motivul aplicarii testelor este faptul ca salveaza foarte mult timp din etapa de dezvoltare, astfel se evita scenariile in care trebuie sa punem functii de printare in consola a anumitor rezultate ale functiilor sau a variabilelor si sa incepem sa facem debugging pentru a descoperi greseala din cod. Testarea se scrie o singura data si fiecare modificara adusa codului nu trebuie sa produca un alt rezultat in urma rularii codului de testare.

Functiile din codul urmator reprezinta adunarea, scaderea, inmultirea si impartirea a doua numere primite ca parametri:

* + calc.py

1. **def** add(x, y):
3. **return** x + y

6. **def** subtract(x, y):
8. **return** x - y

11. **def** multiply(x, y):
13. **return** x \* y

16. **def** divide(x, y):
18. **if** y == 0:
19. **raise** ValueError('Can not divide by zero!')
20. **return** x / y

Functiile anterioare sunt scrise in fisierul “calc.py”. Daca rulam codul adaugand functii de print, codul va functiona, dar nimic nu este automatizat astfel incat facand orice tip de refactorizare sau modificare in cod sa ne asiguram ca functia nu doar ruleaza, ci ruleaza si corect.

Pentru a incepe testarea trebuie sa cream un nou fisier pe care il vom numi “test\_calc.py”. Ideea de a scrie orice fisier de test cu prefixul “test\_”, urmat de numele fisierului asupra caruia se executa testarea este de fapt o conventie in scrierea testelor unitare. Pentru inceput trebuie importat modulul unittest, precum si fisierul modulului pe care dorim sa il testam.

* + test\_calc.py

1. **import** unittest
2. **import** calc

Numele clasei din noul fisier trebuie sa fie cat mai descriptiv pentru ceea ce testam, in cazul nostru se va numi “TestCalc” si va mosteni unittest.TestCase. Mostenirea ne va oferi acces catre multe functionalitati de testare din acea clasa.

Pentru a scrie primul test trebuie sa definim o metoda care sa inceapa cu prefixul “test\_”, daca metoda nu incepe cu acest prefix, ea pur si simplu nu va rula. Pentru inceput vom testa functia de adunare, care primeste “self” ca argument principal. In metoda noastra vom scrie primul test si cum am mostenit din “unittest.TestCase”, avem acces la toate metodele “assert”



Pentru a testa functia noastra de adunare, vom folosi metoda “assertEqual(a,b)” in felul urmator:

* + test\_calc.py

1. **import** unittest
2. **import** calc

5. **class** TestCalc(unittest.TestCase):
7. **def** test\_add(self):
8. self.assertEqual(calc.add(10, 5), 15)

Pentru a rula codul de testare, nu este de ajuns sa folosim comanda “ $ python test\_calc.py” pentru ca nu va returna nimic. Pentru a executa acest fisier trebuie sa folosim comanda “$ python –m unittest test\_calc.py”. In consola pe prima linie testul rulat cu succes este reprezentat de un punct (“.”). Numarul de puncte reprezinta numarul de teste efectuate cu succes. Pe urmatoarea linie apare un sir de linii, dupa care un text de tipul “Ran 1 test in 0.000s” si “OK” daca testele s-au executat cu succes. In caz contrar testele esuate vor fi reprezentate prin “fail” in loc de punct pe primia linie, dupa care consola ne spune unde s-a produs esecul.

* + consola

.

----------------------------------------------------------------------

Ran 4 tests in 0.001s

OK

Daca totusi dorim sa folosim simpla comanda “$ python test\_calc.py”, trebuie sa intervenim in cod si sa adaugam urmatoarele linii:

* + - test\_calc.py

1. **import** unittest
2. **import** calc

5. **class** TestCalc(unittest.TestCase):
7. **def** test\_add(self):
8. self.assertEqual(calc.add(10, 5), 15)
10. **if** \_\_name\_\_ == '\_\_main\_\_':
11. unittest.main()

Dorim sa testam si anumite edge-case-uri, adaugand alte teste in aceeasi functie. Desi adaugam mai multe cazuri de testare, programul ne va spune ca a rulat un singur test deoarece ele se afla in aceeasi functie de testare.

* + test\_calc.py

1. **import** unittest
2. **import** calc

5. **class** TestCalc(unittest.TestCase):
7. **def** test\_add(self):
8. self.assertEqual(calc.add(10, 5), 15)
9. self.assertEqual(calc.add(-1, 1), 0)
10. self.assertEqual(calc.add(-1, -1), -2)
12. **if** \_\_name\_\_ == '\_\_main\_\_':
13. unittest.main()

Pentru a testa toate metodele din fisierul “calc.py” putem copia metoda “test\_add” si sa o modificam sa corespunda celorlalte metode.

* + test\_calc.py

1. **import** unittest
2. **import** calc

5. **class** TestCalc(unittest.TestCase):
7. **def** test\_add(self):
8. self.assertEqual(calc.add(10, 5), 15)
9. self.assertEqual(calc.add(-1, 1), 0)
10. self.assertEqual(calc.add(-1, -1), -2)
12. **def** test\_subtract(self):
13. self.assertEqual(calc.subtract(10, 5), 5)
14. self.assertEqual(calc.subtract(-1, 1), -2)
15. self.assertEqual(calc.subtract(-1, -1), 0)
17. **def** test\_multiply(self):
18. self.assertEqual(calc.multiply(10, 5), 50)
19. self.assertEqual(calc.multiply(-1, 1), -1)
20. self.assertEqual(calc.multiply(-1, -1), 1)
22. **def** test\_divide(self):
23. self.assertEqual(calc.divide(10, 5), 2)
24. self.assertEqual(calc.divide(-1, 1), -1)
25. self.assertEqual(calc.divide(-1, -1), 1)
26. self.assertEqual(calc.divide(5, 2), 2.5)
28. **if** \_\_name\_\_ == '\_\_main\_\_':
29. unittest.main()

Rezultatul rularii acestor patru teste va fi urmatorul:

* + consola

….

----------------------------------------------------------------------

Ran 4 tests in 0.001s

OK

In cazul in care avem o greseala in cod, rezultatul va arata in felul urmator:

* + consola

..F.

======================================================================

FAIL: test\_multiply (\_\_main\_\_.TestCalc)

----------------------------------------------------------------------

Traceback (most recent call last):

File "test\_calc.py", line 18, in test\_multiply

self.assertEqual(calc.multiply(10, 5), 50)

AssertionError: 100000 != 50

----------------------------------------------------------------------

Ran 4 tests in 0.001s

FAILED (failures=1)

In metoda “divide” din fisierul “calc.py” se face o verificare pentru cazul in care al doilea argument este egal cu 0. Astfel, functia arunca o eroare (ValueError). Trebuie sa ne asiguram ca testam si acest lucru, astfel trebuie sa folosim o alta metoda “assert”, in felul urmator:

* + test\_calc.py

1. **def** test\_divide(self):
2. self.assertEqual(calc.divide(10, 5), 2)
3. self.assertEqual(calc.divide(-1, 1), -1)
4. self.assertEqual(calc.divide(-1, -1), 1)
5. self.assertEqual(calc.divide(5, 2), 2.5)
7. with self.assertRaises(ValueError):
8. calc.divide(10, 0)

Uneori testele sunt putin mai complicate de atat si pentru a exemplifica utilitatea modulului unittest pe o clasa, am creat clasa Employee ce are in constructor proprietatile “first”, “last” (pentru nume) si “pay” pentru salariu, precum si trei metode prin care se concateneaza “first” si “last” pentru a crea email-ul angajatului, se formeaza numele intreg din cele doua si se calculeaza o marire de salariu inmultind “pay” cu o valoare procentuala.

* + employee.py

1. **class** Employee:

4. raise\_amt = 1.05
6. **def** \_\_init\_\_(self, first, last, pay):
7. self.first = first
8. self.last = last
9. self.pay = pay
11. @property
12. **def** email(self):
13. **return** '[{}.{}@email.com'.format(self.first](mailto:{}.{}@email.com'.format(self.first), self.last)
15. @property
16. **def** fullname(self):
17. **return** '{} {}'.format(self.first, self.last)
19. **def** apply\_raise(self):
20. self.pay = int(self.pay \* self.raise\_amt)

Pentru a testa aceasta clasa vom crea un fisier “test\_employee.py”. In interiorul sau vom importa “unittest” si modulul “employee”. Prima data testam metoda prin care se creeaza email-ul, astfel facem doua instante ale clasei employee cu parametrii aferenti. Dupa ce avem obiectele create, vom verifica cu “assertEqual” daca email-ul creat din parametrii dati reflecta rezultatul asteptat.

* + test\_employee.py

1. **import** unittest
2. **from** employee **import** Employee

5. **class** TestEmployee(unittest.TestCase):
7. **def** test\_email(self):
8. emp\_1 = Employee('Ion', 'Popescu', 50000)
9. emp\_2 = Employee('Andrei', 'Gheorghe', 60000)
11. self.assertEqual(emp\_1.email, 'Ion[.Popescu@email.com](mailto:Corey.Schafer@email.com)')
12. self.assertEqual(emp\_2.email, 'Andrei[.Gheorghe@email.com](mailto:Sue.Smith@email.com)')
14. emp\_1.first = 'Ionut'
15. emp\_2.first = 'Andreea'
17. self.assertEqual(emp\_1.email, 'Ionut[.Popescu@email.com](mailto:John.Schafer@email.com)')
18. self.assertEqual(emp\_2.email, 'Andreea[.Gheorghe@email.com](mailto:Jane.Smith@email.com)')

Modul de testare este similar si pentru celelalte metode. Ceea ce apare aici ca o inconvenienta este faptul ca la inceputul fiecarei metode de testare, trebuie sa repetam liniile de cod prin care instantiem obiectul Employee. Daca dorim sa schimbam ceva in aceasta instantiere, trebuie sa modificam in fiecare loc in care apare instantierea. Exista o metoda prin care putem instantia o singura data la inceputul fisierului pentru toate metodele. Una din metode se numeste “setUp”, iar cea de-a doua “tearDown”. Prima metoda va rula de fiecare data cand se executa un test, iar a doua va rula la sfarsitul fiecarui test. Astfel implementam “setUp” la inceputul fisierului si folosindu-ne de “self” codul va rula pentru fiecare test in parte. Codul final din acest fisier este urmatorul:

* + test\_employee.py

1. **import** unittest
2. **from** employee **import** Employee

5. **class** TestEmployee(unittest.TestCase):
7. **def** setUp(self):
8. **print**('setUp')
9. self.emp\_1 = Employee('Ion', 'Popescu', 50000)
10. self.emp\_2 = Employee('Andrei', 'Gheorghe', 60000)
12. **def** tearDown(self):
13. **print**('tearDown\n')
15. **def** test\_email(self):
16. **print**('test\_email')
17. self.assertEqual(self.emp\_1.email, 'Ion[.Popescu@email.com](mailto:Corey.Schafer@email.com)')
18. self.assertEqual(self.emp\_2.email, 'Andrei[.Gheorghe@email.com](mailto:Sue.Smith@email.com)')
20. self.emp\_1.first = 'Ionut'
21. self.emp\_2.first = 'Andreea'
23. self.assertEqual(self.emp\_1.email, 'Ionut[.Popescu@email.com](mailto:John.Schafer@email.com)')
24. self.assertEqual(self.emp\_2.email, 'Andreea[.Gheorghe@email.com](mailto:Jane.Smith@email.com)')
26. **def** test\_fullname(self):
27. **print**('test\_fullname')
28. self.assertEqual(self.emp\_1.fullname, 'Ion Popescu')
29. self.assertEqual(self.emp\_2.fullname, 'Andrei Gheorghe')
31. self.emp\_1.first = 'Ionut'
32. self.emp\_2.first = 'Andreea'
34. self.assertEqual(self.emp\_1.fullname, 'Ionut Popescu')
35. self.assertEqual(self.emp\_2.fullname, 'Andreea Gheorghe')
37. **def** test\_apply\_raise(self):
38. **print**('test\_apply\_raise')
39. self.emp\_1.apply\_raise()
40. self.emp\_2.apply\_raise()
42. self.assertEqual(self.emp\_1.pay, 52500)
43. self.assertEqual(self.emp\_2.pay, 63000)

46. **if** \_\_name\_\_ == '\_\_main\_\_':
47. unittest.main()

# Testarea White Box

Termenul “White Box” a fost utilizat datorita conceptului de cutie vizibila si simbolizeaza capacitatea de a vedea prin interfata exterioara a software-ului in fuctionalitatea sa interioara.

White Box Testing, testeaza structura interna, designul si codarea solutiei software. In acest tip de testare, codul este vizibil testerului. Se concentreaza in principal pe verificarea fluxului de intrari si iesiri prin aplicatie, imbunatatirea proiectarii si a capacitatii de utilizare, consolidarea securitatii. Testarea in cutie alba este cunoscuta si sub numele de testare Clear Box, testare Open box, testare structurala, testare Transparent Box si testare Glass Box. In general, este efectuata de catre dezvoltatori. Este una dintre cele doua parti ale abordarii Box Testing pentru testarea software, cealalta find Black Box Testing.

Prin aceasta metoda se verifica urmatoarele:

* Brese interne de Securitate
* Cai rupte sau slab structurate in procesele de codare
* Fluxul de intrari specifice prin cod
* Functionalitatea buclelor conditionale
* Testarea fiecarei declaratii, fiecarui obiect si fiecarei functii

## Tipuri de testare White Box:

Unit testing – In general este primul tip de testare efectuat pe o aplicatie. Testarea unitatii se efectueaza pe fiecare unitate sau bloc de cod pe masura ce acesta este dezvoltat. Ajuta la identificarea majoritatii erorilor inca din cicul de dezvoltare al software-ului. Erorile identificate in aceasta etapa sunt mai ieftine sim ai usor de remediat.

Testarea scurgerilor de memorie(memory leaks) – Reprezinta principala cauza pentru care aplicatiile ruleaza mai lent.

## Exemple de intrumente de testare White Box:

* PyUnit
* HTMLUnit
* NUnit

## Avantajele testarii White Box:

* Optimizarea codului prin gasirea unor erori ascunse.
* Testarea poate fi inceputa inca dintr-o faza incipienta si nu necesita o interfata grafica pentru utilizator.
* Testarea este mai amanuntita, cu posibiltiatea de a acoperi cele mai multe cai.

## Dezavantajele testarii White Box:

* Deoarece testele pot fi complexe, sunt necesare persoane calificate, care dispun de cunostinte bune de programare si implementare.
* Testare aplicatiilor complexe necesita mult timp.

# Testarea Black Box

Termenul “Black Box” a fost utilizat datorita conceptului de cutie invizibila si simbolizeaza faptul ca nu se poate vedea functionarea interioara a software-ului, astfel incat doar experienta utilizatorului final poate fi testate.

Black Box Testing, este definita ca o tehnica de testare in care este testata functionalitatea aplicatiei sub test, fara a privi structura codului intern, detaliile implementarii si cunoasterea cailor interne ale software-ului. Acest tip de testare se bazeaza in totalitate pe cerintele si specificatiile software.

Prin aceasta metoda se verifica urmatoarele:

* Erori de interfata
* Erori in structurile de date sau accesul extern la baza de date
* Erori de comportament sau de performanta
* Erori de initializare si de incheiere

## Tipuri de testare Black Box:

Testarea functionala – Acest tip de testare este legata de cerintele functionale ale unui sistem;

Testarea nefunctionala – Acest tip de testare este legata de cerintele non-functionale, cum ar fi performanta, scalabilitatea si capacitatea de utilizare.

Testarea de regresie – Se face dupa ce remedierea codului, actualizarile sau orice alta intretinere a sistemului a fost facuta pentru a verifica daca noul cod nu l-a afectat pe cel deja existent.

## Exemple de instrumente de testare Black Box:

* LoadRunner
* Jmeter
* Selenium

## Avantajele testarii Black Box:

* Testele se fac din punctul de vedere al utilizatorului si vor ajuta la expunerea discrepantelor din specificatii.
* Persoana care testeaza nu trebuie sa cunoasca limbaje de programare sau modul in care software-ul a fost implementat.
* Testele pot fi efectuate de un organism independent de dezvoltatori, permitand o perspectiva obiectiva si evitarea prejudecatii dezvoltatorilor.
* Cazurile de testare pot fi proiectate imediat ce specificatiile sunt complete.

## Dezavantajele testarii Black Box:

* Doar un mic numar de intrari posibile pot fi testate si multe cai ale programului vor fi lasate neteastate.
* Fara specificatii clare, cazurile de testare vor fi dificile de proiectat.
* Testele pot fi redundante daca dezvoltatorul software a executat deja un caz de testare.