

Python Unit Testing

Drobnîchi Daniel – Nicușor

Cuprins

1. Introducerea Funcției.

2. Blackbox Testing.

A. Equivalence Partitioning

B. Category Partitioning

3. Whitebox Testing.

A. Statement Coverage

B. Condition Coverage

4. Mutații.

5. Concluzie.

1.Introducerea funcției

Pentru proiectul de testare unitară în python am decis să folosesc următoarele tool-uri :

- Pytest
- Coverage for pytest
- Mutmut – Mutation Testing

Funcția testată preia 4 argumente :

- N – Număr de caractere ale unui șir. [Int, 1-20]
- Nr – Număr de la tastatură [Int, >0]
- Np – Număr de la tastatură [Int, >0]
- Sir – Sir de N caractere.

De fiecare când o data este introdusă împotriva specificațiilor, funcția cere un nou input în locul variabilei introduse eronat.

Funcția preia cele două numere, și, în funcție de două criterii, modifică sau nu șirul:

1. Numerele Nr si Np sunt coprime.
2. Nr este palindrom.

Atunci, dacă una dintre acestea este satisfăcută :

Primele [x | x este nr. de cifre din nr1] litere din sir sunt înlocuite cu litere corespunzătoare cifrelor din Nr. [E.g. : Nr = 0010 și sir = ‘Animal’, sir devine ‘aabaal’.

Dacă ambele condiții sunt satisfăcute :

Primele [x | x este nr. de cifre din nr1] litere din sir sunt înlocuite cu litere corespunzătoare cifrelor din Nr + 10. [E.g. : Nr = 0010 și sir = ‘Animal’, sir devine ‘kklkal’.

Dacă niciuna din cele două condiții nu este satisfăcută, atunci șirul nu este schimbat.

Mai jos este prezentat codul sursă și graful funcției.

```

def simple(n, nr1, nr2, sir):
    string1 = "abcdefghij", string2 = "klmnopqrst"
    sirRetinut = sir

    if n <= 0 or n > 20:
        print("Intrudouceti valoare corecta ")
        x = int(input())
        return simple(x, nr1, nr2, sir)
    if nr1 <= 0:
        print("Intrudouceti valoare corecta ")
        y = int(input())
        return simple(n, y, nr2, sir)
    if nr2 <= 0:
        print("Intrudouceti valoare corecta ")
        z = int(input())
        return simple(n, nr1, z, sir)

    clonaNr1 = nr1, clonaNr2 = nr2, count = 0

    while clonaNr2 != 0:
        clonaNr1, clonaNr2 = clonaNr2, clonaNr1 % clonaNr2 #Nr Prime

    if clonaNr1 == 1:
        count += 1

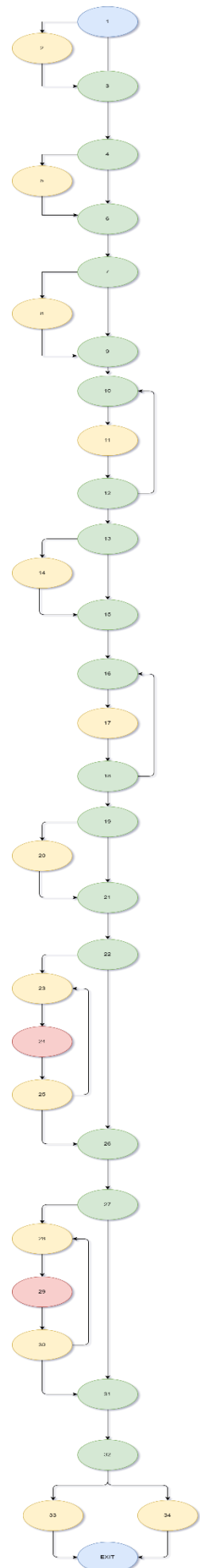
    nr1Flipped = 0
    clona2Nr1 = nr1
    while (clona2Nr1 > 0):
        temp = clona2Nr1 % 10
        nr1Flipped = nr1Flipped * 10 + temp
        clona2Nr1 = clona2Nr1 // 10

    if (nr1Flipped == nr1):
        count += 1

    if (count == 1):
        nr1 = str(nr1)
        for i in range(len(nr1)):
            x = nr1[i]
            sir = sir[:i % n] + string1[int(x)] + sir[i % n + 1:]
    if count == 2:
        nr1 = str(nr1)
        for i in range(len(nr1)):
            x = nr1[i]
            sir = sir[:i % n] + string2[int(x)] + sir[i % n + 1:]

    if sirRetinut == sir:
        return True
    else:
        return False

```



La finalul funcției, dacă șirul este schimbat, funcția returnează 'False', dacă acesta rămâne la fel, aceasta returnează 'True'.

Precizări cu privire la funcțiile de mutare și eventuale erori :

Pentru acest test, am rezumat la folosirea librăriei mutmut pentru generarea de mutanți.

Am încercat să implementez și librăria MutPy, însă după lupte îndelungate și multe versiuni diferite de python, pycharm, anaconda și schimbări de kernel-uri și interpretatoare, nu am reușit să fac scriptul să ruleze pe niciun calculator pe care îl dețin.

Suita de teste pe care am implementat-o pe această funcție simplă urmărește două strategii de BlackBox testing și două strategii de WhiteBox Testing.

2. Blackbox testing

A. Equivalence Partitioning:

După cum am prezentat, avem 4 date de intrare și una de ieșire.

Una din date de intrare este determinată direct de cifra n (sir).

Pentru n:

1. $n_1 : n \mid 0 < n < 20$
2. $n_2 : n \mid n < 1$
3. $n_3 : n \mid n > 20$

Pentru nr:

1. $nr : nr \mid nr < 1$
2. $nr : nr \mid nr > 0$

Pentru np:

1. $np : np \mid np < 1$
2. $np : np \mid np > 0$

Pentru datele de iesire:

1. False
2. True

Metoda pe care am ales-o se apropie foarte mult ca și eficiență și ca tipul de date pe care-l țintește față de metoda valorilor de frontieră.

Este o metodă naivă, în care, realistic, testăm o cantitate foarte mică din funcția propriu-zisă, dar este rapidă și ușor de implementat.

Obținem următoarele date de testat:

1. `c_1111 = (3, 215, 5, 'abc')`
2. `c_1112 = (3, 20, 2, 'abc')`
3. `c_1121 = (3, 215, -8, '')`
4. `c_1122 = (3, 215, -8, '')`
5. `c_1211 = (3, -80, '', '')`
6. `c_1212 = (3, -80, '', '')`
7. `c_2111 = (-5, '', '', '')`
8. `c_...`

```
def test_partitioning(self):
    self.assertEqual(simple(3,20, 2, "superanimal"), True)
    self.assertEqual(simple(3,215, 18, "superanimal"), False)
    self.assertEqual(simple(3,215, -8, "superanimal"), True)
    self.assertEqual(simple(3,215, -8, "superanimal"), True)
    self.assertEqual(simple(3,-80, 8, "superanimal"), True)
    self.assertEqual(simple(3,-80, -8, "superanimal"), True)
    self.assertEqual(simple(0,-20, 2, "superanimal"), True)
    self.assertEqual(simple(0, 215, 18, "superanimal"), False)
    self.assertEqual(simple(0, 215, -8, "superanimal"), True)
    self.assertEqual(simple(0, 215, -8, "superanimal"), True)
    self.assertEqual(simple(0, -80, 8, "superanimal"), True)
    self.assertEqual(simple(0, -80, -8, "superanimal"), True)
    self.assertEqual(simple(21, -20, 2, "superanimal"), True)
    self.assertEqual(simple(21, 215, 18, "superanimal"), False)
    self.assertEqual(simple(21, 215, -8, "superanimal"), True)
    self.assertEqual(simple(21, 215, -8, "superanimal"), True)
    self.assertEqual(simple(21, -80, 8, "superanimal"), True)
    self.assertEqual(simple(21, -80, -8, "superanimal"), True)
```

B. Category partitioning

Pentru această metodă, împărțim funcția în unități care sunt formate din variabile, iar pe acestea le împărțim pe categorii.

Pentru n :

1. $N < 0$
2. $N = 0$
3. $N = 1$
4. $N = 2 \dots 19$
5. $N = 20$
6. $N = 21$

Pentru nr :

1. $Nr < 0$
2. $Nr = 0$
3. $Nr = \text{Palindrom mai mare decat } n$
4. $Nr1 = \text{Palindrom mai mic decat } n$
5. $Nr = \text{Numar care nu este palindrom}$

Pentru np:

1. $Np < 0$
2. $Np = 0$
3. $Np = \text{Coprim cu } nr1, \text{ mai mic}$
4. $Np = \text{Coprim cu } nr1, \text{ mai mare}$

Pentru sir (Conform N).

Pentru Output :

1. True
2. False

În total avem 240 de teste, însă putem să eliminăm cazurile care nu au sens:

1. Valorile aflate în afara valorilor de frontieră (care au fost testate precedent)
2. Cazurile care nu pot exista: palindrom mai mic decât n, mai mare decât n.

3. Cazurile în care output-ul nu poate fi atins.
4. Cazurile care sunt deja identificate de equivalence partitioning.

Obținem următoarele valori pentru fiecare variabilă :

FALSE	TRUE
<ol style="list-style-type: none"> 1. n3xnr3xnp3xsir1xo1 2. n3xnr3xnp4xsir1xo1 3. n3xnr3xnp5xsir1xo1 4. n4xnr3xnp3xsir2xo1 5. n4xnr3xnp4xsir2xo1 6. n4xnr3xnp5xsir2xo1 7. n5xnr3xnp3xsir3xo1 8. n5xnr3xnp4xsir3xo1 9. n5xnr3xnp5xsir3xo1 10. n4xnr4xnp3xsir2xo1 11. n4xnr4xnp4xsir2xo1 12. n4xnr4xnp5xsir2xo1 13. n5xnr4xnp3xsir3xo1 14. n5xnr4xnp4xsir3xo1 15. n5xnr4xnp5xsir3xo1 16. n4xnr5xnp3xsir2xo1 17. n4xnr5xnp4xsir2xo1 18. n5xnr5xnp3xsir3xo1 19. n5xnr5xnp4xsir3xo1 	<ol style="list-style-type: none"> 1. n3xnr3xnp3xsir1xo2 2. n3xnr3xnp4xsir1xo2 3. n3xnr3xnp5xsir1xo2 4. n4xnr3xnp3xsir2xo2 5. n4xnr3xnp4xsir2xo2 6. n4xnr3xnp5xsir2xo2 7. n5xnr3xnp3xsir3xo2 8. n5xnr3xnp4xsir3xo2 9. n5xnr3xnp5xsir3xo2 10. n4xnr4xnp3xsir2xo2 11. n4xnr4xnp4xsir2xo2 12. n4xnr4xnp5xsir2xo2 13. n5xnr4xnp3xsir3xo2 14. n5xnr4xnp4xsir3xo2 15. n5xnr4xnp5xsir3xo2 16. n3xnr5xnp3xsir1xo2 17. n3xnr5xnp4xsir1xo2 18. n4xnr5xnp3xsir2xo2 19. n4xnr5xnp4xsir2xo2 20. n5xnr5xnp3xsir3xo2 21. n5xnr5xnp4xsir3xo2 22. n5xnr5xnp5xsir3xo2

Codul:

```

def test_category_partitioning(self):
    self.assertEqual(simple(1, 55, 2, "a"), False)
    self.assertEqual(simple(1, 55, 67, "a"), False)
    self.assertEqual(simple(1, 55, 11, "a"), False)
    self.assertEqual(simple(5, 555555, 2, "abcde"), False)
    self.assertEqual(simple(5, 555555, 68777, "abcde"), False)
    self.assertEqual(simple(5, 555555, 11, "abcde"), False)
    self.assertEqual(simple(20, 555555555555555555555555, 2,
"abcdefghijklmnpqrstu"), False)
    self.assertEqual(simple(20, 555555555555555555555555,
1015402101225201202154011, "abcdefghijklmnpqrstu"), False)
    self.assertEqual(simple(20, 555555555555555555555555, 11,
"abcdefghijklmnpqrstu"), False)
    self.assertEqual(simple(5, 5555, 2, "abcde"), False)
    self.assertEqual(simple(5, 5555, 68777, "abcde"), False)
    self.assertEqual(simple(5, 5555, 11, "abcde"), False)
    self.assertEqual(simple(20, 555555555555555555555555, 2,
"abcdefghijklmnpqrs"), False)
    self.assertEqual(simple(20, 555555555555555555555555,
1015402101225201202154011, "abcdefghijklmnpqrs"), False)
    self.assertEqual(simple(20, 555555555555555555555555, 11,
"abcdefghijklmnpqrs"), False)
    self.assertEqual(simple(5, 54555, 2, "abcde"), False)
    self.assertEqual(simple(5, 54555, 68777, "abcde"), False)
    self.assertEqual(simple(20, 54, 5, "abcdefghijklmnpqrstu"), False)
    self.assertEqual(simple(20, 54, 1015402101225201202154011,
"abcdefghijklmnpqrstu"), False)
    self.assertEqual(simple(1, 55, 67, "p"), True)
    self.assertEqual(simple(1, 55, 11, "f"), True)
    self.assertEqual(simple(5, 555555, 2, "ppppp"), True)
    self.assertEqual(simple(5, 555555, 68777, "ppppp"), True)
    self.assertEqual(simple(5, 555555, 11, "fffff"), True)
    self.assertEqual(simple(20, 555555555555555555555555, 2,
"pppppppppppppppppppppp"), True)
    self.assertEqual(simple(20, 555555555555555555555555,
1015402101225201202154011, "pppppppppppppppppppppp"), True)
    # self.assertEqual(simple(20, 555555555555555555555555, 11,
"ffffffffffffffffffffffff"), True)
    self.assertEqual(simple(5, 5555, 2, "ppppp"), True)
    self.assertEqual(simple(5, 5555, 68777, "ppppp"), True)
    self.assertEqual(simple(5, 5555, 11, "fffff"), True)
    self.assertEqual(simple(20, 555555555555555555555555, 2,
"pppppppppppppppppppppp"), True)
    self.assertEqual(simple(20, 555555555555555555555555,
1015402101225201202154011, "pppppppppppppppppppppp"), True)
    self.assertEqual(simple(20, 555555555555555555555555, 11,
"ffffffffffffffffffffffff"), True)
    self.assertEqual(simple(1, 54, 5, "e"), True)
    self.assertEqual(simple(1, 54, 67, "e"), True)
    self.assertEqual(simple(6, 555554, 5, "fffffe"), True)
    self.assertEqual(simple(6, 555554, 68777, "fffffe"), True)
    self.assertEqual(simple(20, 55555555555555555555555554, 5,
"efffffffffffffffffffff"), True)
    self.assertEqual(simple(20, 555555555555555555555554,
1015402101225201202154011, "efffffffffffffffffffff"), True)

```


3. Whitebox Testing:

A. Statement coverage:

Statement coverage este cea mai simplă în ceea ce privește testarea structurală.

```
def test_statement_coverage(self):
    main.input = lambda: '5'
    output = main.simple(11, 215, -8, "superanimal")
    assert output == True

    main.input = lambda: '11'
    output = main.simple(11, -215, 2, "1lperanimal")
    assert output == True

    main.input = lambda: '4'
    output = main.simple(27, 22, 11, "ccbi")
    assert output == True
```

Pentru aceasta am obținut următorul coverage, cu 2 branch-uri executate parțial.

Coverage for **main.py**: 100%

44 statements

44 run

0 missing

0 excluded

2 partial

B. Condition coverage:

Pentru partea de condition coverage, am urmat să testăm ca fiecare condiție să fie atinsă complet.

Intrari				Rezultat afișat
N	Nr	Np	Sir	
11	215	-8	superanimal	Se cere introducerea unui Np valid

		5		True
11	-215	5	superanimal	Se cere introducerea unui Nr valid
	215			True
25	215	5	superanimal	Se cere introducerea unui N valid
11				True
11	515	2	superanimal	False
11	515	5	superanimal	False

```
def test_condition_coverage(self):
    main.input = lambda : '5'
    output = main.simple(11, 215, -8, "superanimal")
    assert output == True

    main.input = lambda: '215'
    output = main.simple(11, -215, 5, "superanimal")
    assert output == True

    main.input = lambda: '11'
    output = main.simple(25, 215, 5, "superanimal")
    assert output == True

    self.assertEqual(simple(11, 515, 2, "superanimal"), False)
    self.assertEqual(simple(11, 515, 5, "superanimal"), False)
```

Pentru această metodă putem obține 100% coverage pe cod și 100% pe testarea parțială a codului:

Coverage for **main.py**: 100%

44 statements

44 run

0 missing

0 excluded

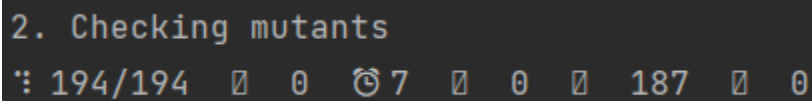
0 partial

4. Mutanți.

Pentru această etapă am folosit mutmut, însă acest tool reușește doar să-mi genereze mutanții, nu să îi și aplice.

Pentru acest lucru am folosit mutmut apply [n] unde n este numărul mutantului, după care am rulat funcția de testare.

Librăria mi-a generat 194 de mutanți :



```
2. Checking mutants
: 194/194 0 0 7 0 0 187 0 0
```

A terminal window with a dark background. The first line is '2. Checking mutants' in a light blue font. The second line shows test results in a light green font: ': 194/194 0 0 7 0 0 187 0 0'. The numbers are separated by small square icons.

Am decis să testez 20 mutanți random, care au fost omorâți de teste, cele mai eficiente fiind cele de tip BlackBox, deoarece acestea conțineau foarte multe assert-uri.

5. Concluzie.

Deoarece funcția este una făcută de mine, limitată la schimbări de șisuri / verificări de numere și de boolean-uri, este clar că nu este una foarte bună pentru metode complicate de testare.

Am ales să urmez această cale deoarece am vrut să văd cum ar trebui să decurgă development-ul, începând cu un schelet simplu și încercând să înțeleg cum funcționează testarea.

Nu pot spune că sunt mulțumit de rezultate, dar consider că am învățat foarte multe cu ajutorul acestui proiect, dacă ar urma alt proiect, cu siguranță aș urma altă idee, cum ar fi să implementez o funcție simplă / să testez ceva mai complicat.