Documentatie

## Echipa

Bobei Vlad-Șerban

Chițu Otilia-Maria

Popescu Pavel-Yanis

## Descrierea clasei pe care o testăm

isValidSudoku(board) determină dacă argumentul său este o tablă Sudoku validă. Poate verifica table complet completate, dar și table care conțin celule goale (reprezentate prin .), acolo unde jucătorul nu a completat încă o cifră.

**În primul rând**, funcția face verificări de validare pentru a se asigura că:

* board este o listă de liste 9x9
* fiecare dintre cele 81 de elemente este fie un punct (.), fie un caracter între '1' și '9'

Dacă oricare dintre aceste condiții nu este îndeplinită, funcția returnează False.

Dacă aceste verificări sunt trecute, funcția returnează True doar dacă toate următoarele condiții sunt îndeplinite:

* fiecare cifră de la 1 la 9 apare cel mult o dată în fiecare rând
* fiecare cifră de la 1 la 9 apare cel mult o dată în fiecare coloană
* fiecare cifră de la 1 la 9 apare cel mult o dată în fiecare dintre cele 9 subgrile 3x3 ale tablei (numite și "box"-uri)

## Configuratia hardware

MacBook Pro – ARM64 M1 Pro, 16 GB RAM

Masini Windows – Intel i9, i7, AMD Ryzen 7, 16/32GB RAM

## Configuratia software

Limbaj de programare: Python 3.11

Framework testare: PyTest 8.3.5

Tool mutanti: mutmut 3.2.3

Tool generare diagrama CFG: PyCFG 0.1

## Bucati de cod si rularea testelor

### Clasa principala care urmeaza a fi supusa testarii

class Solution:  
 def valid\_board\_structure(self, board):  
 valid\_characters = {".", "1", "2", "3", "4", "5", "6", "7", "8", "9"}  
  
 if not isinstance(board, list) or len(board) != 9:  
 return False  
  
 for row in board:  
 if not isinstance(row, list) or len(row) != 9:  
 return False  
  
 for cell in row:  
 if cell not in valid\_characters:  
 return False  
  
 return True  
  
  
 def isValidSudoku(self, board):  
  
 if not self.valid\_board\_structure(board):  
 return False  
 else:  
 columns = defaultdict(set)  
 rows = defaultdict(set)  
  
 def checkRowsAndColsForUniqueElems():  
 for i in range(9):  
 for j in range(9):  
 if board[i][j] == ".":  
 continue  
  
 if board[i][j] in rows[i] or board[i][j] in columns[j]:  
 return False  
  
 rows[i].add(board[i][j])  
 columns[j].add(board[i][j])  
  
 return True  
  
 def checkSubBox(noOfBox):  
 elems = set()  
 startingRow = (noOfBox // 3) \* 3  
 startingColumn = (noOfBox % 3) \* 3  
  
 for i in range(startingRow, startingRow + 3):  
 for j in range(startingColumn, startingColumn + 3):  
 if board[i][j] == ".":  
 continue  
  
 if board[i][j] in elems:  
 return False  
  
 elems.add(board[i][j])  
  
 return True  
  
 if not checkRowsAndColsForUniqueElems():  
 return False  
  
 for i in range(9):  
 if not checkSubBox(i):  
 return False  
  
 return True

### Exemple de testare a claselor de echivalenta [1], [3]

Cum ne-am gândit să facem testele:

Am identificat grupurile de input care dau același rezultat și am scris câte un test pentru fiecare dintre ele. Testele sunt numite sugestiv, tocmai în acest sens. Am găsit următoarele clase de echivalență:

1. Clasa inputurilor valide:

* D = {toate tablele 9×9 cu caractere în {".","1", ..., "9"}}
* CE1​={board∈D ∣ board respectă toate regulile Sudoku}

1. Clasa tablelor cu o linie invalidă(duplicate):

* CE2 ​= {board∈D ∣ ∃i∈[0,8], ∃j,k∈[0,8], j≠k: board[i][j]=board[i][k]≠"."}

1. Clasa tablelor cu o coloană invalid(duplicate):

* CE3​={board∈D ∣ ∃j∈[0,8], ∃i,k∈[0,8], i≠k:board[i][j]=board[k][j]≠"."}

1. Clasa tablelor cu o submatrice de 3x3 din matricea de 9x9, submatrice în configurație invalidă:

* CE4={board∈D ∣ ∃subgrila S de 3×3: ∃(i,j), (k,l)∈S, (i,j)≠(k,l), board[i][j]=board[k][l]≠"."}

1. Clasa tablelor cu număr invalid de linii:

* CE5​={board ∣ board nu are 9 randuri}

1. Clasa tablelor cu număr invalid de coloane:

* CE6​={board ∣ ∃i∈[0,8]: len(board[i])≠9}

1. Clasa tablelor care conțin caractere invalide în configurație:

* CE7​={board ∣ ∃(i,j): board[i][j]∈/{".","1",...,"9"}}

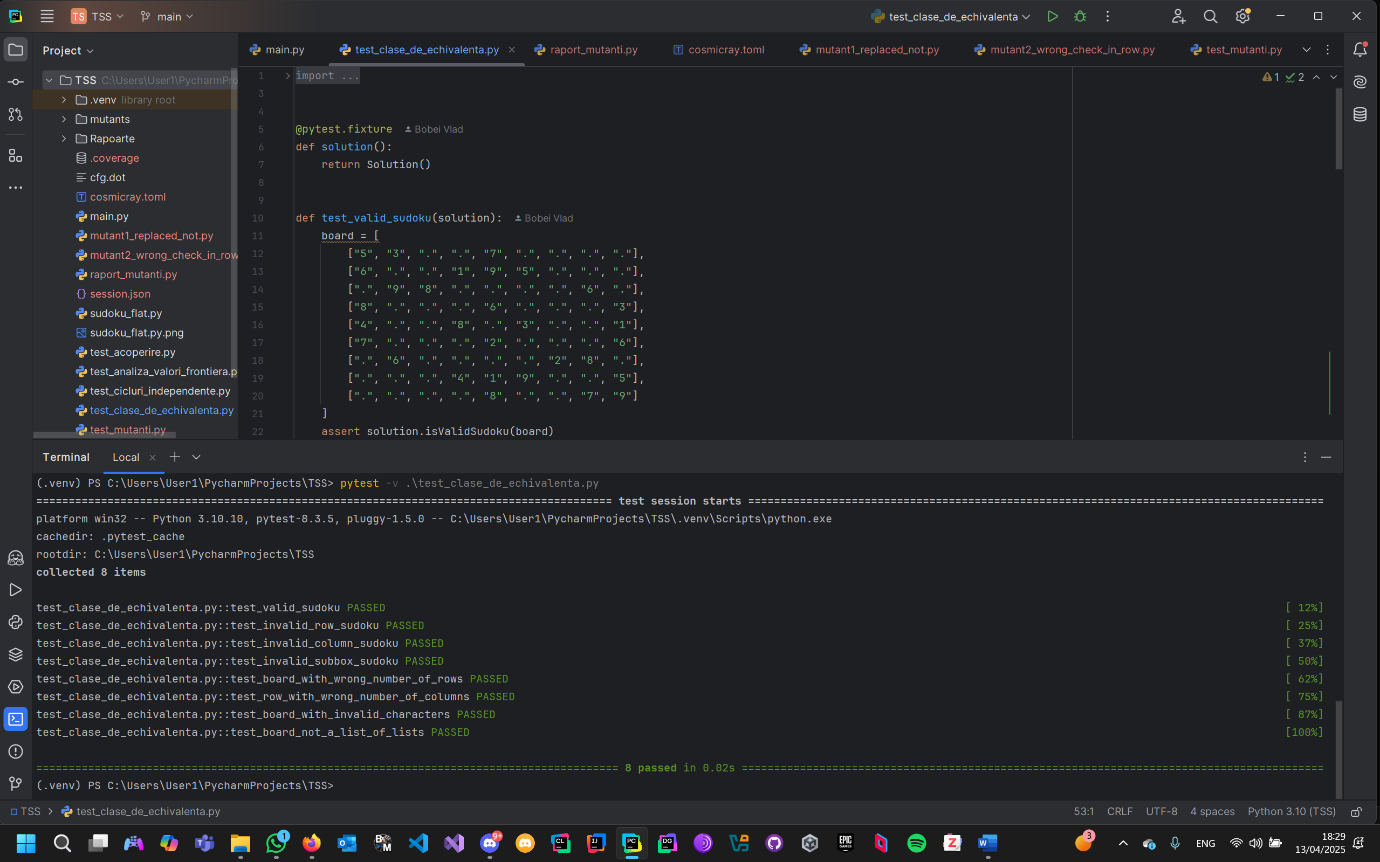
1. Clasa tablelor care nu sunt defapt table, sunt un input total invalid(nu o matrice cu linii și coloane pe care se află cifre de la ‘1’ la ‘9’ sau ‘.’):

* CE8​={board ∣ board nu este o lista de liste de caractere}

def test\_valid\_sudoku(solution):  
 board = [  
 ["5", "3", ".", ".", "7", ".", ".", ".", "."],  
 ["6", ".", ".", "1", "9", "5", ".", ".", "."],  
 [".", "9", "8", ".", ".", ".", ".", "6", "."],  
 ["8", ".", ".", ".", "6", ".", ".", ".", "3"],  
 ["4", ".", ".", "8", ".", "3", ".", ".", "1"],  
 ["7", ".", ".", ".", "2", ".", ".", ".", "6"],  
 [".", "6", ".", ".", ".", ".", "2", "8", "."],  
 [".", ".", ".", "4", "1", "9", ".", ".", "5"],  
 [".", ".", ".", ".", "8", ".", ".", "7", "9"]  
 ]  
 assert solution.isValidSudoku(board)  
  
  
def test\_invalid\_row\_sudoku(solution):  
 board = [  
 ["5", "3", ".", ".", "7", ".", ".", ".", "5"],  
 ["6", ".", ".", "1", "9", "5", ".", ".", "."],  
 [".", "9", "8", ".", ".", ".", ".", "6", "."],  
 ["8", ".", ".", ".", "6", ".", ".", ".", "3"],  
 ["4", ".", ".", "8", ".", "3", ".", ".", "1"],  
 ["7", ".", ".", ".", "2", ".", ".", ".", "6"],  
 [".", "6", ".", ".", ".", ".", "2", "8", "."],  
 [".", ".", ".", "4", "1", "9", ".", ".", "5"],  
 [".", ".", ".", ".", "8", ".", ".", "7", "9"]  
 ]  
 assert not solution.isValidSudoku(board)

### Rularea tuturor testelor referitoare la clasele de echivalenta [3]

Comanda: pytest -v .\test\_clase\_de\_echivalenta.py



### Exemple de testare a valorilor la frontiera [3]

Cum ne-am gândit să facem testele:

Am identificat valorile de limită inferioară și limită superioară relevante pentru problema noastră:

* numărul de rânduri
* numărul de coloane
* valorile admise în celule

Am testat fiecare frontieră imediat sub și imediat peste valorile permise, precum și valorile valide exacte de pe frontieră.

**Domeniul:**

D={toate tablele de Sudoku de dimensiune 9×9 cu caractere în {".","1",...,"9"}}

**Clase de valori de frontieră:**

1. Dimensiuni corecte (valid)

Valoare de frontieră VB1:

* VB1={board∈D ∣ ∣board∣ =9 și ∀i∈[0,8]:∣board[i]∣=9}

2. Prea puține rânduri (8)

Valoare de frontieră VB2:

* VB2={board∣ ∣board∣ =8}

3. Prea multe rânduri (10)

Valoare de frontieră VB3:

* VB3={board∣ ∣board∣ =10}

4. Un rând are prea puține coloane (8)

Valoare de frontieră VB4:

* VB4={board ∣ ∃i∈[0,8]:∣board[i]∣=8}

5. Un rând are prea multe coloane (10)

Valoare de frontieră VB5:

* VB5={board ∣ ∃i∈[0,8]:∣board[i]∣=10}

6. Valoare invalidă în celulă: "0"

Valoare de frontieră VB6:

* VB6={board ∣ ∃(i,j):board[i][j]="0"}

7. Valoare invalidă în celulă: "10"

Valoare de frontieră VB7:

* VB7={board ∣ ∃(i,j):board[i][j]="10"}

8. Valoare invalidă în celulă: "#"

Valoare de frontieră VB8:

* VB8={board ∣ ∃(i,j):board[i][j]="#"}

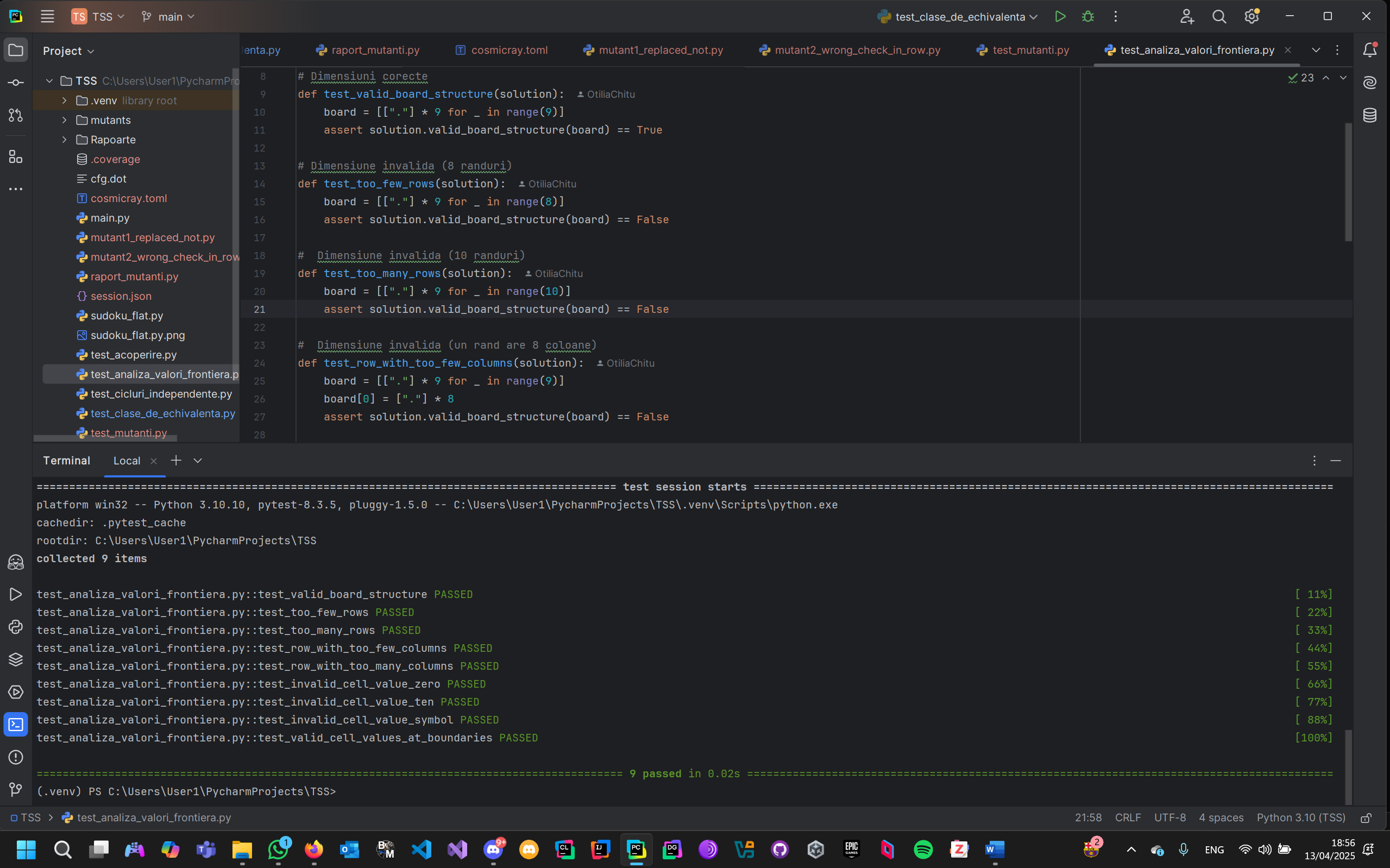
9. Valori valide la margini: "1" și "9"

Valoare de frontieră VB9:

* VB9={board ∣ ∃(i,j):board[i][j]∈{"1","9"} și restul "."}

# Dimensiuni corecte  
def test\_valid\_board\_structure(solution):  
 board = [["."] \* 9 for \_ in range(9)]  
 assert solution.valid\_board\_structure(board) == True  
  
# Dimensiune invalida (8 randuri)  
def test\_too\_few\_rows(solution):  
 board = [["."] \* 9 for \_ in range(8)]  
 assert solution.valid\_board\_structure(board) == False

### Rularea tuturor testelor referitoare la valorile de frontiera [3]



Comanda: pytest -v .\test\_analiza\_valori\_frontiera.py

### Exemple de testare a acoperirii [3]

Cum ne-am gândit să facem testele:

Am urmărit să acoperim:

* **Toate instrucțiunile** (Instruction Coverage)
* **Toate deciziile** (Decision Coverage)
* **Toate ramurile** (Branch Coverage)
* **Toate condițiile** (Condition Coverage)

Am creat teste pentru:

* Cazuri în care validarea structurii eșuează
* Cazuri în care validarea structurii trece, dar validarea logicii Sudoku eșuează
* Cazuri în care toate validările trec

**Domeniul:**

D={toate inputurile posibile pentru isValidSudoku(board)}

**Clase de acoperire:**

1. Board gol valid

Clasă de acoperire AC1:

* AC1={board ∣ board este 9×9, toate celulele "."}

Condiții:

* Structură corectă
* Logica Sudoku corectă

2. Board complet valid (fără puncte)

Clasă de acoperire AC2:

* AC2={board ∣ board complet, cifre "1"-"9", fără duplicate}

Condiții:

* Structură corectă
* Sudoku complet valid

3. Input invalid: nu e listă

Clasă de acoperire AC3:

* AC3={board ∣board ∉ listă de liste}

Condiții:

* Structura invalidă

4. Structură invalidă: prea puține rânduri

Clasă de acoperire AC4:

* AC4={board∣ ∣board∣ ≠ 9}

Condiții:

* Structură invalidă

5. Structură invalidă: un rând nu e listă

Clasă de acoperire AC5:

* AC5={board ∣ ∃i∈[0,8]:type(board[i])≠list}

Condiții:

* Structură invalid

6. Structură invalidă: prea puține coloane într-un rând

Clasă de acoperire AC6:

* AC6={board ∣ ∃i∈[0,8]:∣board[i]∣≠9}

Condiții:

* Structură invalidă

7. Structură invalidă: celulă cu valoare nepermisă

Clasă de acoperire AC7:

* AC7={board ∣ ∃(i,j):board[i][j]∉{".","1",...,"9"}}

Condiții:

* Structură invalidă

8. Logică invalidă: duplicate pe rând

Clasă de acoperire AC8:

* AC8={board ∣ ∃i∈[0,8],∃j,k∈[0,8],j≠k:board[i][j]=board[i][k]≠"."}

Condiții:

* Structură corectă
* Sudoku invalid (duplicate pe rând)

9. Logică invalidă: duplicate în subgrilă 3x3

Clasă de acoperire AC9:

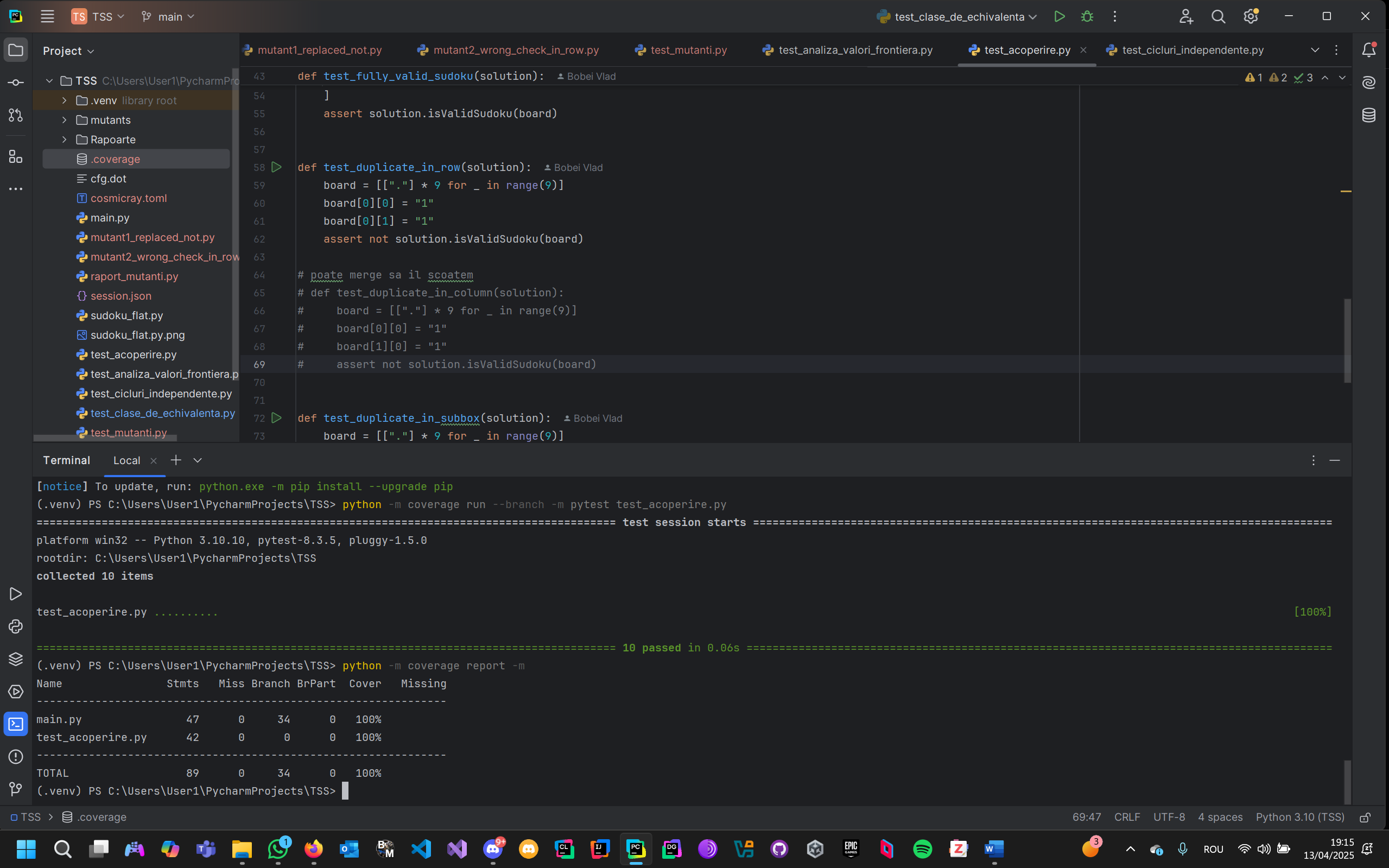
* AC9={board ∣ ∃box S:∃(i,j),(k,l)∈S,(i,j)≠(k,l),board[i][j]=board[k][l]≠"."}

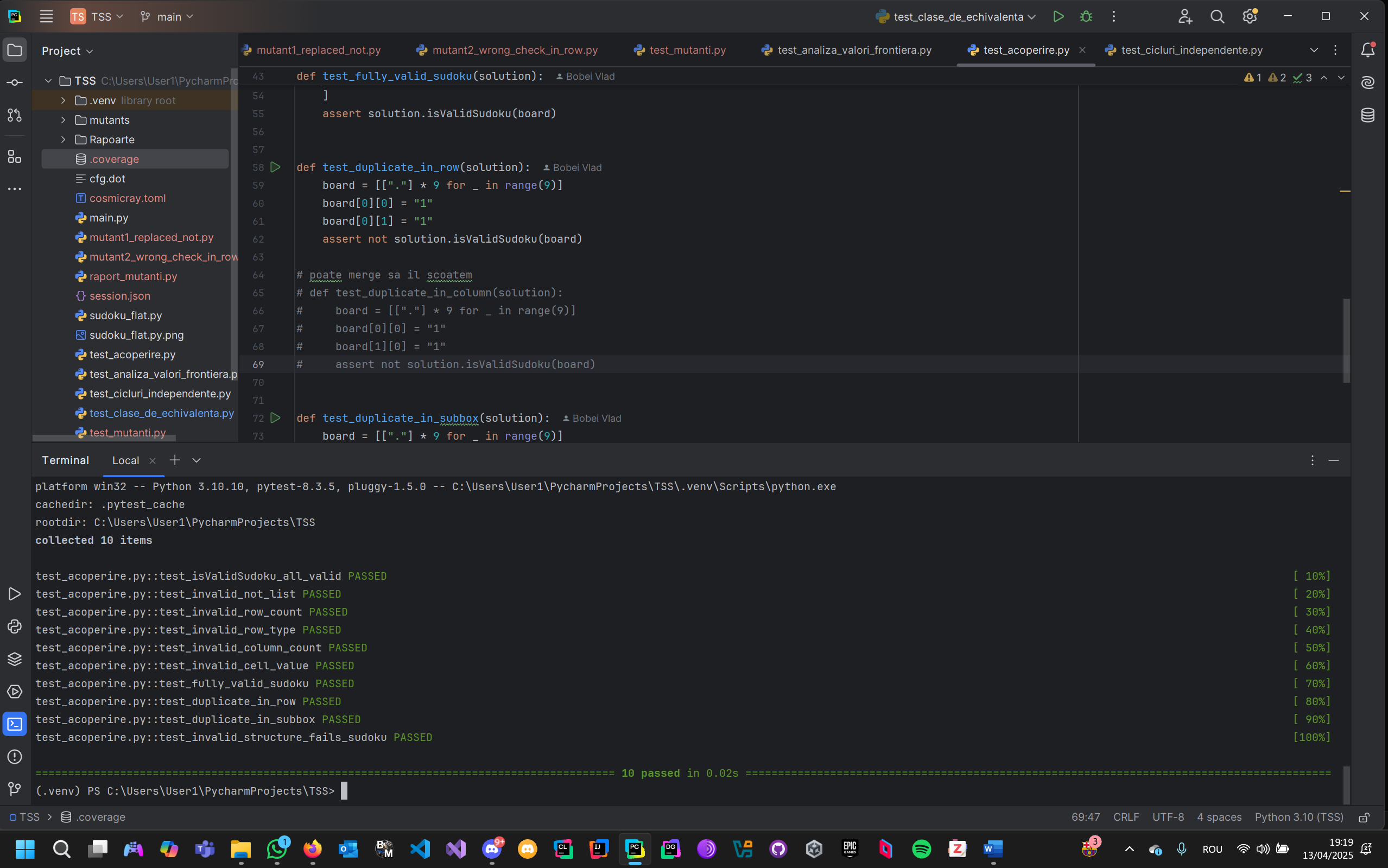
Condiții:

* Structură corectă
* Sudoku invalid (duplicate în box)

def test\_invalid\_not\_list(solution):  
 board = "not a list"  
 assert solution.valid\_board\_structure(board) == False  
  
  
def test\_invalid\_row\_count(solution):  
 board = [["."] \* 9 for \_ in range(8)]  
 assert solution.valid\_board\_structure(board) == False

### Rularea tuturor testelor referitoare la testare a acoperirii [3]





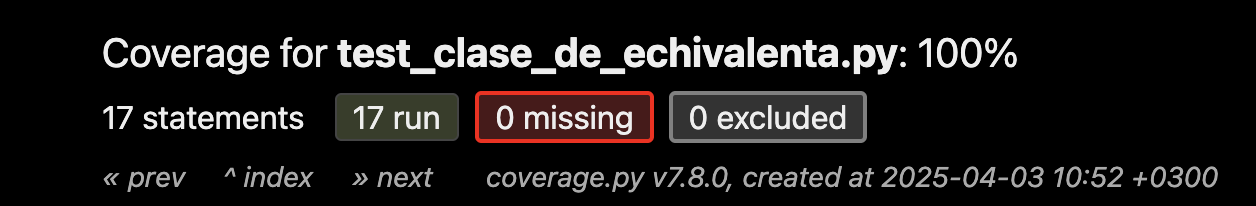
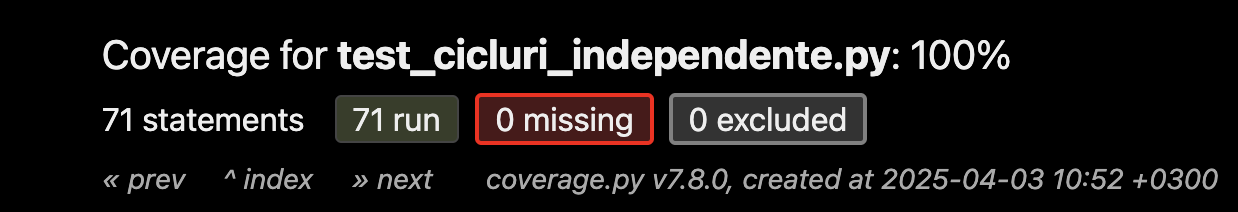
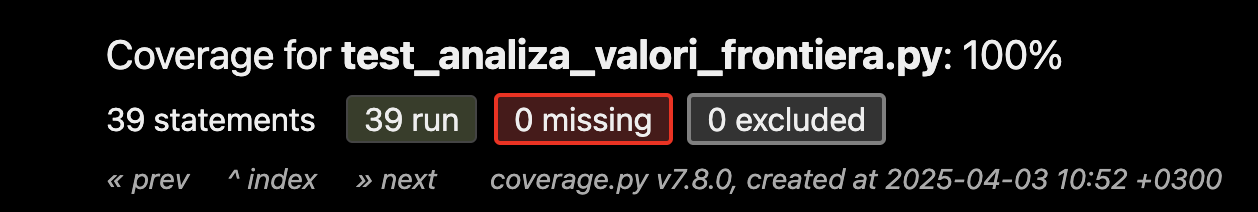
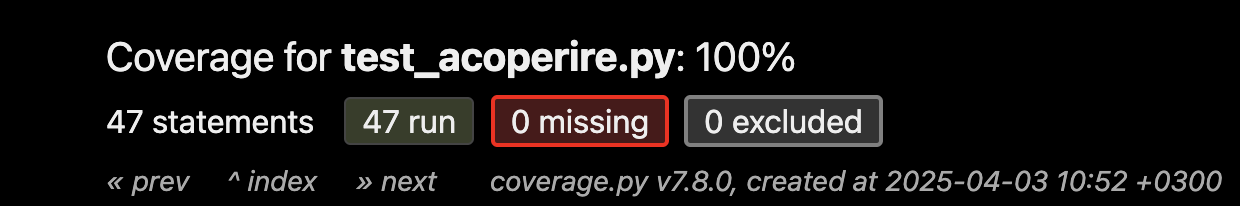
Comenzile:

* 1. python -m coverage run --branch -m pytest test\_acoperire.py

python -m coverage report -m

* 1. pytest -v .\test\_acoperire.py

### Rapoarte HTML pentru acoperire [5]



### Exemple de testare a ciclurilor independente [3]

# 1. valid\_board\_structure 1st if FALSE(isinstance)  
def test\_not\_a\_list(solution):  
 board = "not\_a\_list"  
 assert solution.isValidSudoku(board) == False  
  
# 2. valid\_board\_structure 1st if FALSE(len)  
def test\_list\_len\_not\_9(solution):  
 board = [["."] \* 9] \* 8 # only 8 rows  
 assert solution.isValidSudoku(board) == False  
  
# 3. valid\_board\_structure 2nd if FALSE(isinstance)  
def test\_row\_not\_list(solution):  
 board = ["row"] \* 9 # rows are strings, not lists  
 assert solution.isValidSudoku(board) == False

### Rularea tuturor testelor referitoare la ciclurile independente [3]

A screenshot of a computer program

AI-generated content may be incorrect.

Comanda: pytest -v .\test\_cicluri\_independente.py

### 

### Diagrama CFG care ne-a ajutat sa identificam si sa testam ciclurile independente [2]

### Schita dupa care am creat numarul optim de teste pentru ciclurile independente

1. valid\_board\_structure 1st if FALSE(isinstance)

2. valid\_board\_structure 1st if FALSE(len)

3. valid\_board\_structure 2nd if FALSE(isinstance)

4. valid\_board\_structure 2nd if FALSE(len)

5. valid\_board\_structure 3rd if FALSE(cell invalid)

6. valid\_board\_structure -> checkRowsAndColsForUniqueElems continue FALSE(board in rows)

7. valid\_board\_structure -> checkRowsAndColsForUniqueElems continue FALSE(board in columns)

8. valid\_board\_structure -> checkRowsAndColsForUniqueElems NO continue FALSE(board in rows)

9. valid\_board\_structure -> checkRowsAndColsForUniqueElems NO continue FALSE(board in colums)

10. valid\_board\_structure -> checkRowsAndColsForUniqueElems continue -> checkSubBox continue FALSE(board in elems)

11. valid\_board\_structure -> checkRowsAndColsForUniqueElems continue -> checkSubBox NO continue FALSE(board in elems)

12. valid\_board\_structure -> checkRowsAndColsForUniqueElems NO continue -> checkSubBox continue FALSE(board in elems)

13. valid\_board\_structure -> checkRowsAndColsForUniqueElems NO continue -> checkSubBox NO continue FALSE(board in elems)

14. valid\_board\_structure -> checkRowsAndColsForUniqueElems continue -> checkSubBox continue TRUE

15. valid\_board\_structure -> checkRowsAndColsForUniqueElems continue -> checkSubBox NO continue TRUE

16. valid\_board\_structure -> checkRowsAndColsForUniqueElems NO continue -> checkSubBox continue TRUE

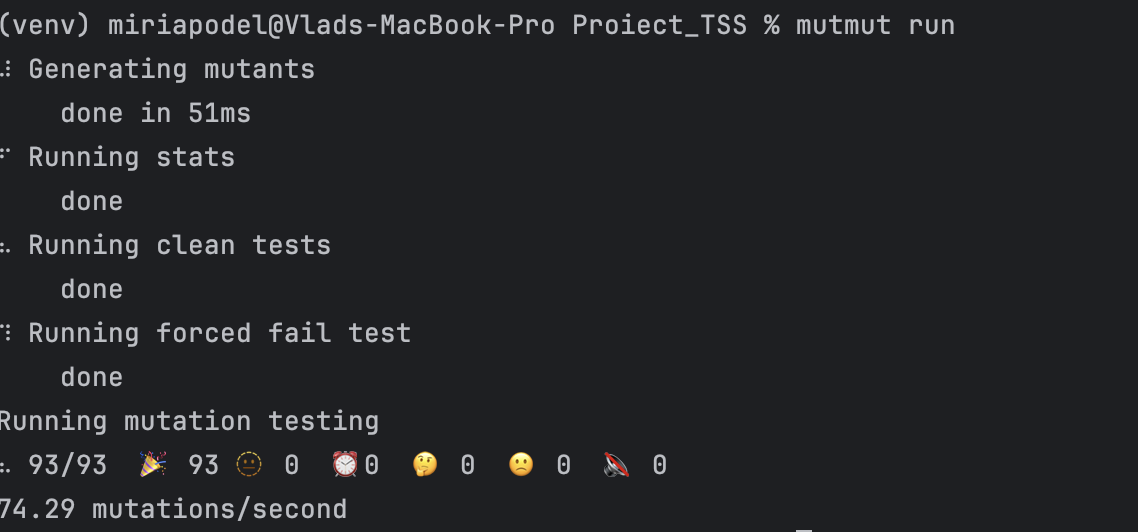
17. valid\_board\_structure -> checkRowsAndColsForUniqueElems NO continue -> checkSubBox NO continue TRUE

Pentru a face această schiță, ne-am uitat pe graficul de mai sus și am determinat toate drumurile unice de la punctul de start al programului la cel de final.

Pentru fiecare astfel de drum am creat un test care să-l valideze.

### Generarea mutantilor [4]

Pentru a genera mutanții ne-am folosit de MutMut. Ca să rulăm am folosit comanda mutmut run.



### Exemple de mutanti generati [4]

def xǁSolutionǁisValidSudoku\_\_mutmut\_20(self, board):  
  
 if not self.valid\_board\_structure(board):  
 return False  
 else:  
 columns = defaultdict(set)  
 rows = defaultdict(set)  
  
 def checkRowsAndColsForUniqueElems():  
 for i in range(9):  
 for j in range(9):  
 if board[i][j] == ".":  
 continue  
  
 if board[i][j] in rows[i] or board[i][None] in columns[j]:  
 return False  
  
 rows[i].add(board[i][j])  
 columns[j].add(board[i][j])  
  
 return True  
  
 def checkSubBox(noOfBox):  
 elems = set()  
 startingRow = (noOfBox // 3) \* 3  
 startingColumn = (noOfBox % 3) \* 3  
  
 for i in range(startingRow, startingRow + 3):  
 for j in range(startingColumn, startingColumn + 3):  
 if board[i][j] == ".":  
 continue  
  
 if board[i][j] in elems:  
 return False  
  
 elems.add(board[i][j])  
  
 return True  
  
 if not checkRowsAndColsForUniqueElems():  
 return False  
  
 for i in range(9):  
 if not checkSubBox(i):  
 return False  
  
 return True

def xǁSolutionǁisValidSudoku\_\_mutmut\_61(self, board):  
 if not self.valid\_board\_structure(board):  
 return False  
 else:  
 columns = defaultdict(set)  
 rows = defaultdict(set)  
  
 def checkRowsAndColsForUniqueElems():  
 for i in range(9):  
 for j in range(9):  
 if board[i][j] == ".":  
 continue  
  
 if board[i][j] in rows[i] or board[i][j] in columns[j]:  
 return False  
  
 rows[i].add(board[i][j])  
 columns[j].add(board[i][j])  
  
 return True  
  
 def checkSubBox(noOfBox):  
 elems = set()  
 startingRow = (noOfBox // 3) \* 3  
 startingColumn = (noOfBox % 3) \* 3  
  
 for i in range(startingRow, startingRow + 3):  
 for j in range(startingColumn, startingColumn + 3):  
 if board[i][j] == ".":  
 continue  
  
 if board[i][j] in elems:  
 return False  
  
 elems.add(board[i][None])  
  
 return True  
  
 if not checkRowsAndColsForUniqueElems():  
 return False  
  
 for i in range(9):  
 if not checkSubBox(i):  
 return False  
  
 return True

### Referinte:

[1] OpenAI, ChatGPT, <https://chatgpt.com/>, Data generarii: 1-2 Aprilie 2025

[2] PyPi, PyCFG, <https://pypi.org/project/pycfg/>, Data accesarii: 2 Aprilie 2025

[3] PyTest, <https://docs.pytest.org/en/stable/>, Data accesarii: 1-2 Aprilie 2025

[4] mutmut, <https://mutmut.readthedocs.io/en/latest/>, Data accesarii: 2 Aprilie 2025

[5] Coverage, <https://coverage.readthedocs.io/en/7.8.0/>, Data accesarii: 1 Aprilie 2025

[6] Predut Sorina, Cursuri TSS, <https://drive.google.com/drive/folders/18CVua5zkJeaY1UsDjJIJgHjMqYxs_d7R>, Data accesarii: 1-2 Aprilie 2025