A Guide to Developing a Sudoku Generator with TDD in Python

This guide will walk you through building a Sudoku generator and solver in Python from scratch using Test-Driven Development (TDD). Each step follows the "Red -> Green -> Refactor" cycle.

Phase 0: Project Setup

Let's begin by setting up a simple Python project.

1. **Create Project Files:** Create two files in your project directory:
   - sudoku.py: This will contain our main application logic.
   - test_sudoku.py: This is the corresponding test file.
2. **Install pytest:** If you don't have it already, install the pytest framework.
   ```
   pip install pytest
   ```

   You will run tests from your terminal using the pytest command.

Phase 1: The Rules Engine

Our first goal is to implement the fundamental rule of Sudoku: verifying if a number placement is valid. This is the most basic, dependency-free piece of logic and the perfect starting point for TDD.

Cycle 1: is_valid() - Validating Number Placement

🎯 **Goal**: Create a method is_valid(row, col, num) to determine if placing a number in a given cell is valid.

🔴 Red: Write a Failing Test

In test_sudoku.py, we'll write tests covering all validation scenarios using simple assert statements.

```
# test_sudoku.py
from sudoku import SudokuBoard

def test_is_valid():
    initial_grid = [
        [5, 3, 0, 0, 7, 0, 0, 0, 0],
        [6, 0, 0, 1, 9, 5, 0, 0, 0],
        [0, 9, 8, 0, 0, 0, 0, 6, 0],
        [8, 0, 0, 0, 6, 0, 0, 0, 3],
```

```
        [4, 0, 0, 8, 0, 3, 0, 0, 1],
        [7, 0, 0, 0, 2, 0, 0, 0, 6],
        [0, 6, 0, 0, 0, 0, 2, 8, 0],
        [0, 0, 0, 4, 1, 9, 0, 0, 5],
        [0, 0, 0, 0, 8, 0, 0, 7, 9]
    ]
    board = SudokuBoard(initial_grid)

    # Test valid placement (placing 4 at (0, 2))
    assert board.is_valid(0, 2, 4) is True, "is_valid(0, 2, 4) should be True"

    # Test row conflict (placing 5 at (0, 2))
    assert board.is_valid(0, 2, 5) is False, "is_valid(0, 2, 5) should be False due to row conflict"

    # Test column conflict (placing 8 at (0, 2))
    assert board.is_valid(0, 2, 8) is False, "is_valid(0, 2, 8) should be False due to column
conflict"

    # Test 3x3 box conflict (placing 9 at (1, 1))
    assert board.is_valid(1, 1, 9) is False, "is_valid(1, 1, 9) should be False due to box conflict"
```

Running pytest now will result in an error because the SudokuBoard class doesn't exist. This is our "Red" light.

🟢 Green: Write Code to Pass the Test

In sudoku.py, create the SudokuBoard class and the is_valid method with just enough logic to make the tests pass.

```
# sudoku.py
import copy

class SudokuBoard:
    def __init__(self, grid):
        # Use deepcopy to ensure the original grid is not modified
        self.grid = copy.deepcopy(grid)
        self.size = 9

    def is_valid(self, row, col, num):
        # Check row
        for i in range(self.size):
            if self.grid[row][i] == num:
                return False
        # Check column
        for i in range(self.size):
            if self.grid[i][col] == num:
                return False

        # Check 3x3 box
        start_row, start_col = 3 * (row // 3), 3 * (col // 3)
        for i in range(3):
            for j in range(3):
                if self.grid[i + start_row][j + start_col] == num:
                    return False

        return True
```

Now, run pytest in your terminal. All tests should pass, turning the light "Green"!

🔵 Refactor: Clean Up the Code

The Python code is clean and idiomatic. The three checks in is_valid are distinct and easy to understand. No refactoring is needed at this stage.

Phase 2: The Solver

With our rules engine validated, we can build the solver using a backtracking algorithm.

Cycle 2: solve() - Implementing the Backtracking Algorithm

🎯 **Goal**: Create a solve() method that can find the solution for any valid Sudoku puzzle.

🔴 Red: Write a Test for the Solver

We'll add a new test to test_sudoku.py to verify that the solver correctly solves a sample puzzle.

```
# test_sudoku.py
# ... (add this test function to the file)

def test_solve():
    puzzle_grid = [
        [5, 3, 0, 0, 7, 0, 0, 0, 0], [6, 0, 0, 1, 9, 5, 0, 0, 0], [0, 9, 8, 0, 0, 0, 0, 6, 0],
        [8, 0, 0, 0, 6, 0, 0, 0, 3], [4, 0, 0, 8, 0, 3, 0, 0, 1], [7, 0, 0, 0, 2, 0, 0, 0, 6],
        [0, 6, 0, 0, 0, 0, 2, 8, 0], [0, 0, 0, 4, 1, 9, 0, 0, 5], [0, 0, 0, 0, 8, 0, 0, 7, 9]
    ]
    solution_grid = [
        [5, 3, 4, 6, 7, 8, 9, 1, 2], [6, 7, 2, 1, 9, 5, 3, 4, 8], [1, 9, 8, 3, 4, 2, 5, 6, 7],
        [8, 5, 9, 7, 6, 1, 4, 2, 3], [4, 2, 6, 8, 5, 3, 7, 9, 1], [7, 1, 3, 9, 2, 4, 8, 5, 6],
        [9, 6, 1, 5, 3, 7, 2, 8, 4], [2, 8, 7, 4, 1, 9, 6, 3, 5], [3, 4, 5, 2, 8, 6, 1, 7, 9]
    ]

    puzzle = SudokuBoard(puzzle_grid)
    assert puzzle.solve() is True, "Solver should return True for a solvable puzzle"
    assert puzzle.grid == solution_grid, "The solved puzzle does not match the expected
solution"
```

This test will fail because the solve method doesn't exist yet.

🟢 Green: Write Just Enough Code to Pass

Implement the backtracking algorithm in sudoku.py. A helper method find_empty is useful here.

```
# sudoku.py
# ... add these methods to the SudokuBoard class

    def find_empty(self):
        for i in range(self.size):
            for j in range(self.size):
                if self.grid[i][j] == 0:
                    return (i, j)  # row, col
        return None

    def solve(self):
        find = self.find_empty()
        if not find:
            return True  # Board is full, puzzle solved
        else:
            row, col = find

        for num in range(1, self.size + 1):
            if self.is_valid(row, col, num):
                self.grid[row][col] = num

                if self.solve():
                    return True

                self.grid[row][col] = 0 # Backtrack
        return False
```

Run pytest. The test_solve function should now pass!

🔵 Refactor: Clean Up

The solver code is a standard recursive backtracking implementation. The logic is clean and well-structured with the find_empty helper. No refactoring is necessary at this point.

Phase 3: The Puzzle Generator

Now that we have a working solver, we can use it to generate new puzzles. The strategy is to generate a full, random solution and then remove digits ("dig holes").

Cycle 3: generate() - Creating the Puzzle

🎯 **Goal**: Create a generate(difficulty) function that returns a puzzle and its solution.

🔴 Red: Write a Test for the Generator

In test_sudoku.py, we'll test the generator function. The test will verify that the puzzle has the correct number of empty cells and that the provided solution is correct.

```
# test_sudoku.py
# ... add this import at the top
from sudoku import generate

def test_generate():
    difficulty = 45  # Number of cells to remove
    puzzle_board, solution_board = generate(difficulty)

    # 1. Validate the number of empty cells
    empty_cells = sum(row.count(0) for row in puzzle_board.grid)
    assert empty_cells == difficulty, f"Puzzle should have {difficulty} empty cells, but has
{empty_cells}"

    # 2. Validate that the solution is correct
    puzzle_copy = SudokuBoard(puzzle_board.grid)
    assert puzzle_copy.solve() is True, "Generated puzzle must be solvable"
    assert puzzle_copy.grid == solution_board.grid, "Solver result must match the provided
solution"
```

This test will fail because the generate function doesn't exist.

🟢 Green: Write Code to Make it Pass

To generate different puzzles, we first need to refactor our solver to introduce randomness.

1. Refactor SudokuBoard.solve() to accept a randomize flag.

```python
# In sudoku.py
# ... add this import
import random

# ... update the solve method in SudokuBoard class
    def solve(self, randomize=False):
        find = self.find_empty()
        if not find:
            return True
        else:
            row, col = find

        numbers = list(range(1, self.size + 1))
        if randomize:
            random.shuffle(numbers)

        for num in numbers:
            if self.is_valid(row, col, num):
                self.grid[row][col] = num
                if self.solve(randomize):
                    return True
                self.grid[row][col] = 0 # Backtrack
        return False
```

2. Create the generate function in sudoku.py.

```python
# sudoku.py
# ... (at the bottom of the file, outside the class)

def generate(difficulty):
    # 1. Create a full, random solution
    empty_grid = [[0] * 9 for _ in range(9)]
    solution_board = SudokuBoard(empty_grid)
    solution_board.solve(randomize=True) # Use the randomized solver
```

```
# 2. Create a puzzle by removing cells
puzzle_board = SudokuBoard(solution_board.grid)

cells_to_remove = difficulty
while cells_to_remove > 0:
    row = random.randint(0, 8)
    col = random.randint(0, 8)

    if puzzle_board.grid[row][col] != 0:
        puzzle_board.grid[row][col] = 0
        cells_to_remove -= 1

return puzzle_board, solution_board
```