

SAT-Solving Sudoku

Exercise sheet 12

Use `sudoku-exercises.ipynb` for the exercises, or write your code in `my_sudoku.py`. Written explanations are in `sudoku.ipynb`; executable solutions are in `sudoku.py`.

Installing PySAT

Use a virtual environment for this session. It keeps these packages separate from your other Python projects and makes it easier to use the same Python installation for both installing packages and running your code. The commands below create an environment called `.venv` and activate it. Activate it again whenever you open a new terminal; use `deactivate` when finished. If Python is missing, install Python 3 from <https://www.python.org/downloads/> first, then reopen the terminal.

Windows (PowerShell)

```
python --version
mkdir sat-sudoku
cd sat-sudoku
python -m venv .venv
.\.venv\Scripts\Activate.ps1
```

macOS (Terminal)

```
python3 --version
mkdir -p sat-sudoku
cd sat-sudoku
python3 -m venv .venv
source .venv/bin/activate
```

Linux (Terminal)

```
python3 --version
mkdir -p sat-sudoku
cd sat-sudoku
python3 -m venv .venv
source .venv/bin/activate
```

If PowerShell blocks activation, open Command Prompt in the `sat-sudoku` folder and run:

```
.venv\Scripts\activate.bat
```

On Debian or Ubuntu, if `python3 -m venv .venv` fails because the `venv` module is missing, run `sudo apt install python3-venv`, then repeat only the environment creation and activation commands, staying in the `sat-sudoku` folder.

On any platform, once the environment is active, install PySAT:

```
python -m pip install --upgrade pip
python -m pip install python-sat
```

Choose how to work

For a notebook, save `sudoku-exercises.ipynb` in the working folder above. Install JupyterLab and run the notebook:

```
python -m pip install jupyterlab ipykernel
jupyter lab sudoku-exercises.ipynb
```

Use Shift+Enter to run a cell. Replace each `raise NotImplementedError(...)` statement with your code, then run its checks.

You can also write a standard Python script. Save your work as `my_sudoku.py` in the same folder and run `python my_sudoku.py`. The code blocks below belong in that file, in the order shown. Keep your function definitions above the calls that use them.

Exercise 12.1. Read a CNF. A clause is an OR of literals, and a CNF is an AND of clauses. PySAT represents literals by signed integers: 1 means x_1 , and -1 means $\neg x_1$. Thus $[1, -2]$ means $x_1 \vee \neg x_2$. Use this helper to call the solver. It returns the true variable numbers, or None for UNSAT. An empty set can be a valid model, so use `model is None` to check for UNSAT.

```
from itertools import combinations, product
from time import perf_counter
from pysat.formula import CNF
from pysat.solvers import Solver

def solve_cnf(clauses, name="g3"):
    """Return the true variable numbers, or None if the CNF is UNSAT."""
    with Solver(name=name, bootstrap_with=clauses) as solver:
        if not solver.solve():
            return None
        return {lit for lit in solver.get_model() if lit > 0}
```

Write the formula represented below. Predict the three truth values and what happens when the extra clause is added. Then run the code and explain which clause forces each value.

```
warmup = [[1, 2], [-1], [-2, 3]]
model = solve_cnf(warmup)
print("True variables:", model)
print("With the extra clause:", solve_cnf(warmup + [[-3]]))
```

Exercise 12.2. Write three clauses. You choose exactly one workshop: robotics (variable 1) or painting (variable 2). Robotics requires reserving a kit (variable 3). Reserving a kit is allowed even if you choose painting.

Define `workshop_clauses()` that returns three clauses: at least one workshop, at most one workshop, and robotics implies a kit. Use PySAT to list all satisfying assignments.

Complete the function, then run the enumeration below. Each printed tuple contains the truth values for robotics, painting, and reserving a kit, in that order. The solver may list them in any order.

```
def workshop_clauses():
    # Return one clause per requirement. Variables: 1, 2, 3.
    raise NotImplementedError("Encode the three workshop requirements")

with Solver(name="g3", bootstrap_with=workshop_clauses()) as solver:
    for model in solver.enum_models():
        print(tuple(i in model for i in (1, 2, 3)))
```

The method `enum_models()` finds a model, excludes that exact assignment, and repeats until no further models remain. In each model, a positive integer means the corresponding variable is true; a negative integer means it is false.

Exercise 12.3. Build an exactly-one helper. Write clauses allowing exactly one of three candidates x, y, z to be true.

Turn your idea into the function `exactly_one(clauses, literals)`, which takes a list of clauses and a list of distinct literals. It appends new clauses to `clauses` expressing the condition “exactly one of `literals` is true”. It changes the supplied list and returns nothing. Use `combinations(literals, 2)`, which gives all distinct pairs.

```
def exactly_one(clauses, literals):
    literals = list(literals)
    # Add an at-least-one clause, then forbid each pair.
    # combinations(literals, 2) gives all distinct pairs.
    raise NotImplementedError("Write the exactly-one helper")
```

How many clauses do nine candidates need? What should happen for zero candidates, or one candidate? Check all assignments for lists of up to three candidates:

```
for n in range(4):
    clauses = []
    exactly_one(clauses, range(1, n + 1))
    for values in product([False, True], repeat=n):
        units = [[i if value else -i]
                  for i, value in enumerate(values, start=1)]
        assert (solve_cnf(clauses + units) is not None) == (sum(values) == 1)
print("Exactly-one checked for zero, one, two, and three candidates.")
```

Exercise 12.4. Encode Sudoku. In a standard 9×9 Sudoku, each cell has one digit, and each digit appears once in every row, column, and 3×3 block. A zero below marks a blank.

```
puzzle = [
    "100007090",
    "030020008",
    "009600500",
    "005300900",
    "010080002",
    "600004000",
    "300000010",
    "040000007",
    "007000300",
]
```

```
def var(r, c, d):
    return 81 * r + 9 * c + d
```

Let $x_{r,c,d}$ mean “row r , column c contains digit d ”. Rows and columns run from 0 to 8; digits run from 1 to 9. As (r, c, d) ranges over all valid triples, the function $v(r, c, d) = 81r + 9c + d$ uses each variable number from 1 to 729 exactly once.

Write these five functions. Each appends clauses to the list it receives.

(a) `add_cells(clauses)`: exactly one digit per cell.

- (b) `add_rows(clauses)`: for each row and digit, exactly one column.
- (c) `add_columns(clauses)`: for each column and digit, exactly one row.
- (d) `add_blocks(clauses)`: for each block and digit, exactly one cell.
- (e) `add_clues(clauses, puzzle)`: one unit clause per nonzero clue.

Reuse `exactly_one`. For a fixed cell, its candidates are `[var(r, c, d) for d in range(1, 10)]`. For a block beginning at `(r0, c0)`, use rows `range(r0, r0 + 3)` and columns `range(c0, c0 + 3)`. Each starting coordinate is 0, 3, or 6.

Use this builder to call your functions:

```
def build_sudoku_cnf(puzzle):
    if (len(puzzle) != 9 or
        any(not isinstance(row, str) or len(row) != 9 or
            any(ch not in "0123456789" for ch in row) for row in puzzle)):
        raise ValueError("Use nine strings of nine digits, with 0 for blanks")
    clauses = []
    add_cells(clauses)
    add_rows(clauses)
    add_columns(clauses)
    add_blocks(clauses)
    add_clues(clauses, puzzle)
    return CNF(from_clauses=clauses)
```

Each of the first four functions must make 81 exactly-one groups. Count the clauses and check that the number is $4 \cdot 81 \cdot 37$ plus the number of clues, counting repeated clauses. Matching the count is useful, but does not prove the encoding correct. Explain both directions: a valid grid satisfies your formula, and a model of your formula describes a valid grid.

Exercise 12.5. Solve and check the grid. Use the supplied decoder to read a model. It reports an error if a cell has zero or several true candidates.

```
def decode(model):
    grid = []
    for r in range(9):
        row = []
        for c in range(9):
            choices = [d for d in range(1, 10) if var(r, c, d) in model]
            if len(choices) != 1:
                raise ValueError(f"Cell ({r}, {c}) has {len(choices)} digits")
            row.append(choices[0])
        grid.append(row)
    return grid
```

The solver checks your clauses; this checker helps detect mistakes in your encoding. Complete a separate checker for rows, columns, blocks, and clues:

```
def valid_solution(grid, puzzle):
    if grid is None or len(grid) != 9 or any(len(row) != 9 for row in grid):
        return False
    digits = set(range(1, 10))
```

```

rows_ok = all(set(row) == digits for row in grid)
# Add analogous checks for columns, blocks, and original clues.
raise NotImplementedError("Finish the independent Sudoku checker")

```

The checker must inspect the grid itself. Then build the formula and solve:

```

cnf = build_sudoku_cnf(puzzle)
model = solve_cnf(cnf.clauses)
if model is None:
    print("UNSAT")
else:
    solution = decode(model)
    for row in solution:
        print(" ".join(map(str, row)))
    assert valid_solution(solution, puzzle)
    broken = [row[:] for row in solution]
    broken[0][1] = broken[0][0]
    assert not valid_solution(broken, puzzle)
    print("The solution passes the Sudoku checks; the broken grid fails.")

```

Why is seeing a SAT answer alone not enough? Which modelling mistakes could pass the clause-count check but fail the grid check?

Exercise 12.6. Explain an UNSAT answer. Make a copy of the puzzle:

```
bad_puzzle = list(puzzle)
```

Change its first row to "110007090". Rebuild the CNF from that copy. Check that the solver returns None. Which two unit clauses conflict with which row constraint? Keep the original puzzle and cnf for the next experiment.

Exercise 12.7. Compare search choices (optional). Use this helper to compare MiniSat (m22) and Glucose (g3) on the original CNF. Each call creates a fresh solver.

```

def measure(cnf, name, phases=None):
    with Solver(name=name, bootstrap_with=cnf.clauses) as solver:
        if phases is not None:
            solver.set_phases(phases)
        start = perf_counter()
        answer = solver.solve()
        seconds = perf_counter() - start
        stats = solver.accum_stats()
        model = ({lit for lit in solver.get_model() if lit > 0}
                  if answer else None)
    return answer, seconds, stats, model

```

Call `measure(cnf, "m22")` and `measure(cnf, "g3")`. Compare the decision, conflict, and propagation counts. Check each decoded model with `valid_solution`.

Then give MiniSat phase preferences for the “digit 1” candidates:

```
preferences = [var(r, c, 1) for r in range(9) for c in range(9)]  
result = measure(cnf, "m22", phases=preferences)
```

A phase is a preferred truth value when the solver makes a choice; it does not add a clause. Does it change which assignments are legal? Does comparing two engines isolate one heuristic? Small-puzzle timings can vary between runs; a single run does not establish which solver is generally faster.