CNF

`main.py` reads the input in (space-seperated) Polish notation as the 1st command-line argument. To run use as follows:

```
$ python3 cnf/main.py "> & - p q & p > r q"
```

Output is printed to stdout.

Implementation details

Files and their purposes:

- **main.py** - Contains the main logic. Parsing of the input into a tree is done here.
- **cnf.py** - Contains CNF converter. *It is implemented exactly like in the textbook/slides.*
- **node.py** - Contains tree data structure for managing formulas. There is base class called `Node` which has atttributes `left` and `right` and is extended by subclasses `Liter` (for literals), `Not`, `And`, `Or`, `Impl`, `RevImpl` and `Equiv`.
- **printer.py** - Contains Polish, infix and a special CNF printer functions.
- **valid.py** - Contains logic for checking validity of a CNF formula. *This, too, is implemented exactly like in the textbook/slides*.

Nonogram

`main()` function inside `nonogram.py` reads input from stdin and writes the result into stdout. To run it use following command format:

```
$ python3 nonogram.py <example.cwd >nonogram.sol
```

`nonogram.py` also uses files `nonogram.dimacs` (to construct the input for `minisat`) and `minisat.out` (to store the output of `minisat`).

NOTE: This program currently uses `minisat` from the current directory. Please change the first line of `NonogramSolver._solve_cnf()` accordingly if needed.

Converting nonogram to CNF clauses

The algorithm I used to build the input for `minisat` from the given nonogram is relatively simple (and would work for nonograms of maximum size of 10 by 10 or so):

- I assign a variable for each cell of the nanogram, starting from 1 to R*C.
- Then for each row, I find all interpretations for the corresponding variables of the row that do NOT comply by the nonogram rules of the row. We get a clause like this for each interpretation:(-i

and i+1 and i+2 and -(i+3) and ... and -(i+C-1))

- Do the same for columns.
- Take the conjunction of all the resulting clauses and finally negate the conjunction to get a formula in CNF.

If found, a satisfying interpretation given by `minisat` will be a solution to the nonogram. That is because any interpretaion that satisfies the negation of all the "non-rules" of the nonogram would also satisfy rules of the nonogram.

NonogramSolver class

The class `NonogramSolver` has one public method `solve()`. It is initialized by passing two lists as arguments that contain row and column information for the nanogram.

```
>>> ns = NonogramSolver(rows, cols)
```

Calling `ns.solve()` will either return `None` if no solution is found or will return a string containing '#' for colored cells and '.' for blank cells:

```
>>> solution = ns.solve()
>>> print(solution, end='')
........
.####...
.######.
.##..##.
.##..##.
.######.
.####...
.##.....
.##.....
.##.....
........
```

Private methods do the heavy work for `solve()`:

- **_build_clauses()** - builds the CNF clauses using `rows` and `cols` attributes.
- **_write_dimacs_file()** - prepares input for `minisat` using the built clauses.
- **_solve_cnf()** - reads the `minisat` output to get the satisfying interpretation and to build the nonogram character grid.