

CLASSIC ALGORITHMS IN COMPUTER SCIENCE

Coding a 1D Cellular Automaton

A walk through wolfram_1D_ca.py: how each function carries out one of the four jobs we planned

```
$ python wolfram_1D_ca.py
```

From plan to program

1 Make the rule
`create_boolean_function()`
Rule number → lookup table

2 Set up the start
`simulate_wolfram()`
A row of zeros with the middle cell set to 1

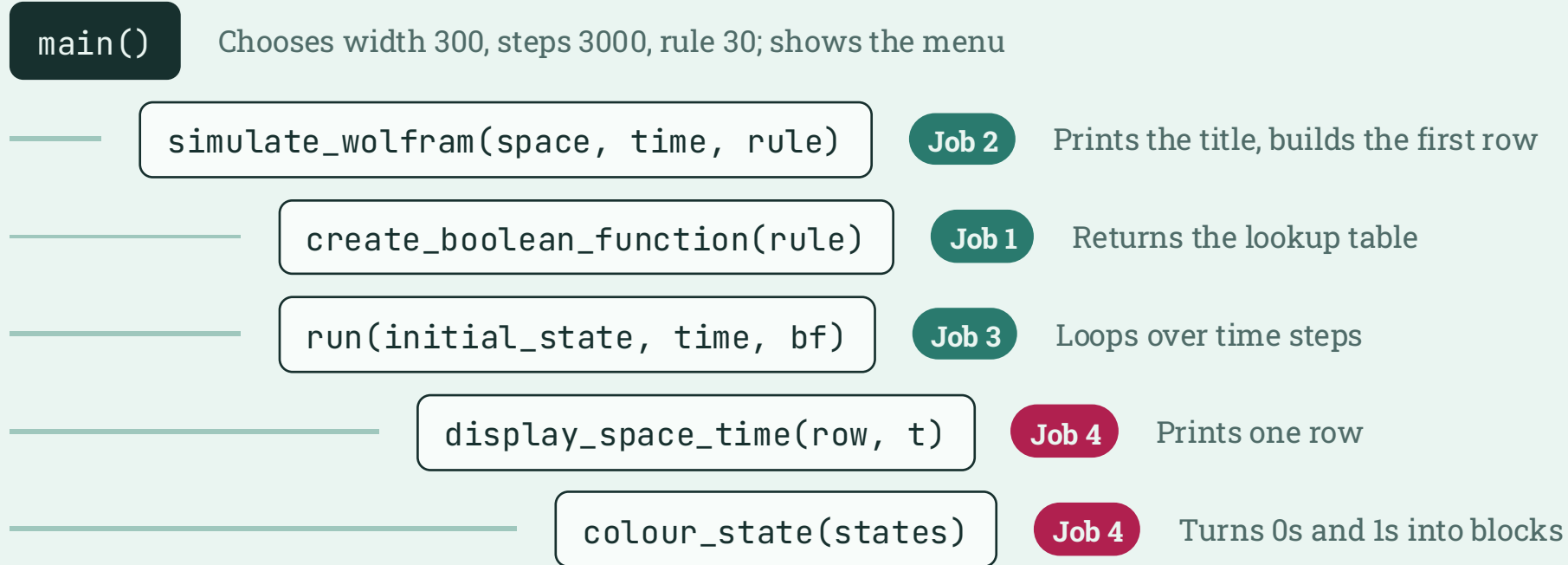
3 Step forward
`run()`
Build each new row from the old one

4 Draw it
`display_space_time() + colour_state()`
One row of coloured blocks per time step

The main program ties the jobs together

`main()` picks the rule, the width (space) and the number of steps (time), then hands them to `simulate_wolfram()`, which runs all four jobs

Which function calls which



Each step to the right is one call deeper. Read top to bottom, this is the order things happen.

Libraries the program borrows

Import	What it is used for
<code>from colorama import init</code>	Makes colour codes work in Windows terminals; called once at start-up
<code>from termcolor import colored</code>	Wraps text in a colour: <code>colored('███', 'black')</code>
<code>from time import sleep</code>	Pauses half a second between rules
<code>import random</code>	Picks a random rule if the user presses q
<code>import sys</code>	Imported but never used, so it can be deleted

colorama and termcolor are not built in: install them once with `pip install colorama termcolor`

Job 1 in code: create_boolean_function

```
def create_boolean_function(rule):  
  
    # 1: rule number -> 8-bit string  
  
    outputs = bin(rule)  
  
    outputs = outputs.replace('0b', '')  
  
    outputs = outputs.zfill(8)  
  
  
    # 2: every 3-cell neighbourhood  
  
    inputs = [(1,1,1),(1,1,0),(1,0,1),(1,0,0),  
              (0,1,1),(0,1,0),(0,0,1),(0,0,0)]  
  
  
    # 3: pair each input with its output  
  
    boolean_function = {}  
  
    for i in range(len(inputs)):  
  
        boolean_function[inputs[i]] = int(outputs[i])  
  
    return boolean_function
```

1 Number to binary

bin() gives the digits with a 0b label; remove it, then pad on the left to 8 digits

2 Wolfram's order

The 8 inputs as tuples, from 111 down to 000

3 A dictionary

Key: a neighbourhood tuple. Value: the new cell, turned from the character '1' into the number 1 by int()

Tracing Rule 30 through Job 1

```
bin(30)
```

```
'0b111110'
```

```
.replace('0b', '')
```

```
'111110'
```

```
.zfill(8)
```

```
'00011110'
```

i	0	1	2	3	4	5	6	7
inputs[i]	111	110	101	100	011	010	001	000
outputs[i]	0	0	0	1	1	1	1	0

Why pad? Rule 4 is '100' without zfill: only 3 digits, so outputs[3] crashes with an IndexError

Job 2 in code: the starting row

```
# inside simulate_wolfram(space, time, rule)
bf = create_boolean_function(rule)
initial_state = [0] * space
initial_state[space // 2] = 1
run(initial_state, time, bf)
```

Example: space = 11, so space // 2 = 5



Job 3 in code: the run loop

```
def run(initial_state, max_time, boolean_function):

    current_state = initial_state

    display_space_time(current_state, 0)

    length = len(current_state)

    for tick in range(1, max_time):

        new_state = []

        for i in range(length):

            left = current_state[(i - 1) % length]

            middle = current_state[i]

            right = current_state[(i + 1) % length]

            output = boolean_function((left, middle, right))

            new_state.append(output)

        current_state = new_state

    display_space_time(current_state, tick)
```

Outer loop = time

One pass per row. Row 0 is printed before the loop starts

Inner loop = space

One pass per cell, reading three neighbours

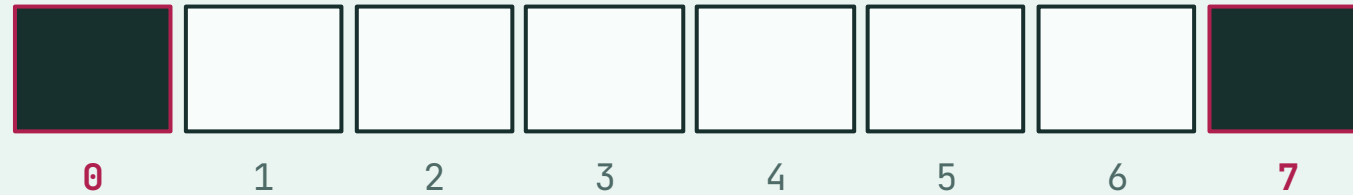
Look it up

The tuple (left, middle, right) is the dictionary key

Swap, then draw

Only a finished row replaces the old one

Wrap-around with the % operator



length = 8: the two end cells are neighbours, as if the row were a ring

Left of the first cell

$$(0 - 1) \% 8 = 7$$

Right of the last cell

$$(7 + 1) \% 8 = 0$$

Everywhere else

$$(3 + 1) \% 8 = 4$$

Why new_state is a separate list

Correct: as in the program

```
new_state = []
for i in range(length):
    # read from current_state
    ...
    new_state.append(output)
current_state = new_state
```

Every cell reads the same old row, so all cells change at the same moment

Bug: overwrite in place

```
for i in range(length):
    # read from current_state
    ...
    current_state[i] = output
```

When cell i reads its left neighbour, cell $i - 1$ already holds its new value, so the output is no longer Rule 30

Job 4 in code: drawing a row

```
def colour_state(states, on="black", off="white",
                 character='█' * 1):
    coloured_state = ''
    for cell in states:
        if cell:
            coloured_state += colored(character, on)
        else:
            coloured_state += colored(character, off)
    return coloured_state
```

```
# in display_space_time(row, t)
edge = colored("|", "green")
print(edge + space + edge + time)
```

1 is on, 0 is off

if cell: is true for 1 and false for 0, so no comparison is needed

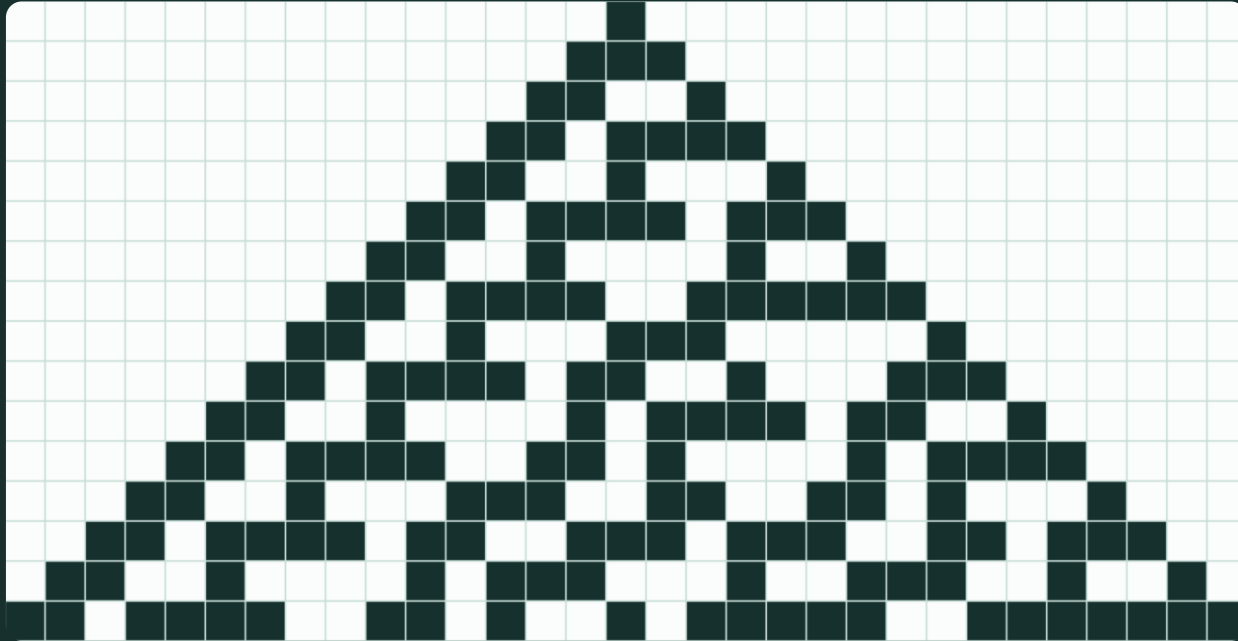
Two blocks per cell

Characters are tall and thin; two side by side look square. Change * 1 to * 2 for wider cells

Build, then print

The whole row becomes one string, framed by green bars, with the time step on the end

What the four jobs produce together



Rule 30 with space = 31 and time = 16

Made by running the program's own
`create_boolean_function` and run, row by row

16 rows: $t = 0$ is the starting row, then `range(1, 16)` adds 15 more

In the terminal each row also has green bars and its time step on the right

main(): settings, then a menu

Settings

```
width = 300  
steps = 3000
```



Show Rule 30

```
simulate_wolfram(  
    width, steps, 30)
```



Ask the user

```
choice = input(...)
```

Anything except q

```
for rule in range(256):  
    simulate_wolfram(width, steps, rule)  
    sleep(0.5)
```

Every rule from 0 to 255, half a second apart

q

```
rule = random.randint(0, 255)  
simulate_wolfram(width, steps, rule)  
print(message)
```

One random rule, then a goodbye message

The file ends with `if __name__ == "__main__": init(), then main(),` so it only runs when started directly

Running it in class

```
# once
pip install colorama termcolor
# every time
python wolfram_1D_ca.py
```

Setting	What happens	For a projector, try
width = 300	Each row is over 600 characters wide and wraps on most screens	width = 60
steps = 3000	3000 rows per rule; 768,000 rows for all 256 rules	steps = 60

Press **Ctrl + C** to stop a run at any time

Your turn: extend the program

1 Choose a rule

Ask the user for a rule number and show just that rule

Hint: `int(input(...))`

2 Random start

Start from a random row, then compare Rules 110 and 184

Hint: `random.randint(0, 1)`

3 Hard edges

Remove the wrap-around: treat cells past either end as 0. Does Rule 30 change?

4 Count the cells

Print how many cells are on in each row, next to the time step

Hint: `sum(row)`

Tidy-up first: delete the unused `import sys`