Principled Programming

Introduction to Coding in Any Imperative Language

Tim Teitelbaum

Emeritus Professor

Department of Computer Science

Cornell University

Cellular Automata

Introduction

We illustrate two-dimensional arrays, and enumerations over them, using the examples of Cellular Automata and the Game of Life.

Cellular Automata model the Universe as a rectangular grid of *cells*, each in a given *state*. Time progresses in discrete steps. On each clock tick, each cell simultaneously decides what state to enter based on its current state and the current states of its neighbors. Each cell makes its decision independently, but all cells follow the same rules.

The Game of Life is a particular Cellular Automaton that models birth and death.

Systematic top-down development of an entire program is illustrated. Deeply-nested **for**-statements in the code arise naturally as a consequence of stepwise refinement, but are readily understood.

Top-level Code Structure

Class Sim models the notion of a Cellular Automaton, and its simulation.

```
class Sim:
    ________
```

☞ **Aggregate the definitions of related variables and methods in a class.**

The simulation as a whole is implemented as method `main`.

```
class Sim:

    @classmethod
    def main(cls) -> None:
        ___________
```

☞ **Aggregate the definitions of related variables and methods in a class.**

main

A *class method* is defined within a class using

- Decorator `@classmethod`
- First parameter **`cls`**

The simulation as a whole is implemented as method `main`.

```
class Sim:

    @classmethod
    def main(cls) -> None:
        ___________
```

☞ **Aggregate the definitions of related variables and methods in a class.**

A *class method* is defined within a class using
- Decorator `@classmethod`
- First parameter **`cls`**

The simulation as a whole is implemented as method `main`.

```
class Sim:

    @classmethod
    def main(cls) -> None:
        __________
```

The simulation will be run by invoking this class method using its qualified name, i.e., `Sim.main()`, which passes the class itself as an implicit first argument matching **`cls`**.

☞ **Aggregate the definitions of related variables and methods in a class.**

The simulation as a whole is implemented as method `main`.

```
class Sim:

    @classmethod
    def main(cls) -> None:
        """Simulate a cellular automaton."""
        ___________
```

☞ **A method header-comment specifies the effect of invoking it, and (if the method has non-None type) the value returned. If the method has parameters, the specification is written in terms of those parameters.**

For the implementation of `main`, adopt the pattern that first initializes, then computes.

```
class Sim:

    @classmethod
    def main(cls) -> None:
        """Simulate a cellular automaton."""
        #.Initialize.
        #.Compute.
```

☞ **Master stylized code patterns, and use them.**

Instantiate placeholders *Initialize* and *Compute* for the problem at hand.

```
class Sim:

    @classmethod
    def main(cls) -> None:
        """Simulate a cellular automaton."""
        #.Create the initial Universe and display it.
        #.Simulate and display LAST_GEN additional generations.
```

☞ **Master stylized code patterns, and use them.**

Refine the specifications.

```
class Sim:

    @classmethod
    def main(cls) -> None:
        """Simulate a cellular automaton."""
        # Create the initial Universe and display it.
        Sim.initialize()
        Sim.display()

        #.Simulate and display LAST_GEN additional generations.
```

☞ **Program top-down, outside-in.**

Refine the specifications.

```
class Sim:

    @classmethod
    def main(cls) -> None:
        """Simulate a cellular automaton."""
        # Create the initial Universe and display it.
        Sim.initialize()
        Sim.display()

        # Simulate and display LAST_GEN additional generations.
        for Sim.generation in range(1, Sim.LAST_GEN + 1):
            Sim.next_generation()
            Sim.display()
```

☞ **Program top-down, outside-in.**

main

initialize, display, and next_generation are other class methods to be defined. Class methods are always invoked using their qualified names, e.g., Sim.display().

Refine the specifications.

```
class Sim:

    @classmethod
    def main(cls) -> None:
        """Simulate a cellular automaton."""
        # Create the initial Universe and display it.
        Sim.initialize()
        Sim.display()

        # Simulate and display LAST_GEN additional generations.
        for Sim.generation in range(1, Sim.LAST_GEN + 1):
            Sim.next_generation()
            Sim.display()
```

☞ **Many short procedures are better than large blocks of code.**

Method specifications use a syntactic construct are known as a “docstring”

Refine the specifications.

```
class Sim:

    @classmethod
    def main(cls) -> None:
        """Simulate a cellular automaton."""
        # Create the initial Universe and display it.
        Sim.initialize()
        Sim.display()

        # Simulate and display LAST_GEN additional generations.
        for Sim.generation in range(1, Sim.LAST_GEN + 1):
            Sim.next_generation()
            Sim.display()
```

Top-level Code Structure

Create stubs for the methods that have been introduced, which you can do mindlessly.

```
class Sim:
    ...

    @classmethod
    def initialize(cls) -> None:
        """Create the initial Universes old. """
        pass
    @classmethod
    def display(cls) -> None:
        """Display the present Universe.""“
        pass
    @classmethod
    def next_generation(cls) -> None:
        """Update Universe to be the next generation.""“
        pass
```

☞ **Defer challenging code for later; do the easy parts first.**

Data Representation

The simulation of a cellular automaton models a finite M-by-N Universe of cells. The states of each generation of next cells is determined from the states of the old cells, where generations are numbered from 0 through LAST_GEN. The state of each cell is modeled as an **int**.

Specify and declare the data representation.

```
class Sim:

    M: int = 5                                 # Height of Universe.
    N: int = 20                                # Width of Universe.
    old: list[list[int]]  = []                 # old Universe.
    next: list[list[int]] = []                 # next Universe.
    LAST_GEN: int = 40                         # Last generation.
    generation: int = 0                        # Generation number.

    ...
```

☞ **Aggregate the definitions of related variables and methods in a class.**

Data Representation

The simulation of a cellular automaton models a finite M-by-N Universe of cells. The states of each generation of next cells is determined from the states of the old cells, where generations are numbered from 0 through LAST_GEN. The state of each cell is modeled as an **int**.

N.B. The term "state" is overloaded. Each cell of the Universe has a "state".

Specify and declare the data representation.

```
class Sim:

    M: int = 5                                  # Height of Universe.
    N: int = 20                                 # Width of Universe.
    old: list[list[int]]  = []                  # old Universe.
    next: list[list[int]] = []                  # next Universe.
    LAST_GEN: int = 40                          # Last generation.
    generation: int = 0                         # Generation number.

    ...
```

☞ **Introduce program variables whose values describe "state".**

Data Representation

The simulation of a cellular automaton models a finite M-by-N Universe of cells. The states of each generation of next cells is determined from the states of the old cells, where generations are numbered from 0 through LAST_GEN. The state of each cell is modeled as an **int**.

N.B. The term "state" is overloaded. Each cell of the Universe has a "state", and the simulation as a whole has a "state".

Specify and declare the data representation.

```
class Sim:

    M: int = 5                                         # Height of Universe.
    N: int = 20                                        # Width of Universe.
    old: list[list[int]]  = []                         # old Universe.
    next: list[list[int]] = []                         # next Universe.
    LAST_GEN: int = 40                                 # Last generation.
    generation: int = 0                                # Generation number.

    ...
```

☞ **Introduce program variables whose values describe "state".**

Specify and declare the data representation.

```
class Sim:

    M: int = 5                                   # Height of Universe.
    N: int = 20                                  # Width of Universe.
    old: list[list[int]]  = []                   # old Universe.
    next: list[list[int]] = []                   # next Universe.
    LAST_GEN: int = 40                           # Last generation.
    generation: int = 0                          # Generation number.

    ...
```

Names of variables intended to be constant throughout program execution are, by convention, all capital letters.

☞ **Minimize use of literal numerals in code; define and use symbolic constants.**

Data Representation

Specify and declare the data representation.

```
class Sim:

    M: int = 5                                # Height of Universe.
    N: int = 20                               # Width of Universe.
    old: list[list[int]]  = []                # old Universe.
    next: list[list[int]] = []                # next Universe.
    LAST_GEN: int = 40                        # Last generation.
    generation: int = 0                       # Generation number.

    ...
```

Variables <u>declared and initialized</u> at the top-level of a class are called *class variables*, and are shared among all methods of the class.

Specify and declare the data representation.

```
class Sim:

    M: int = 5                                  # Height of Universe.
    N: int = 20                                 # Width of Universe.
    old: list[list[int]]  = []                  # old Universe.
    next: list[list[int]] = []                  # next Universe.
    LAST_GEN: int = 40                          # Last generation.
    generation: int = 0                         # Generation number.

    ...
```

The initial value of the 2-D array must be created in method initialize because the needed construct (list comprehension) is not permitted here.

Data Representation

Specify and declare the data representation.

```
class Sim:

    M: int = 5                                # Height of Universe.
    N: int = 20                               # Width of Universe.
    old: list[list[int]]  = []                # old Universe.
    next: list[list[int]] = []                # next Universe.
    LAST_GEN: int = 40                        # Last generation.
    generation: int = 0                       # Generation number.

    ...
```

The initial value of the 2-D array must be created in method initialize because the needed construct (list comprehension) is not permitted here.

Technically, initialization to [] is a violation of the representation invariant because [] is not M-by-N.

initialize

We now turn to implementation of the methods: `initialize`.

```
class Sim:
    ...

    @classmethod
    def initialize(cls) -> None:
        """Create the initial Universe."""
        pass

    ...
```

initialize

We now turn to implementation of the methods: `initialize`.

```
class Sim:
    ...

    @classmethod
    def initialize(cls) -> None:
        """Create the initial Universe."""
        #.Initialize old and new Universes to M-by-N arrays of 0.

    ...
```

initialize

We now turn to implementation of the methods: `initialize`.

```
class Sim:
    ...

    @classmethod
    def initialize(cls) -> None:
        """Create the initial Universe."""
        # Initialize old and new Universes to M-by-N arrays of False.
        Sim.old =  [[0 for _ in range(0, Sim.N)]
                       for _ in range(0, Sim.M)]
        Sim.next = [[0 for _ in range(0, Sim.N)]
                       for _ in range(0, Sim.M)]

    ...
```

initialize

Long lines can be split if between matched parentheses or brackets.

We now turn to implementation of the methods: `initialize`.

```
class Sim:
    ...

    @classmethod
    def initialize(cls) -> None:
        """Create the initial Universe."""
        # Initialize old and new Universes to M-by-N arrays of False.
        Sim.old =  [[0 for _ in range(0, Sim.N)]
                    for _ in range(0, Sim.M)]
        Sim.next = [[0 for _ in range(0, Sim.N)]
                    for _ in range(0, Sim.M)]

    ...
```

initialize

We now turn to implementation of the methods: `initialize`.

```
class Sim:
    ...

    @classmethod
    def initialize(cls) -> None:
        """Create the initial Universe."""
        # Initialize old and new Universes to M-by-N arrays of False.
        Sim.old =  [[0 for _ in range(0, Sim.N)]
                     for _ in range(0, Sim.M)]
        Sim.next = [[0 for _ in range(0, Sim.N)]
                     for _ in range(0, Sim.M)]

    ...
```

The temporary violation of the representation invariant has now been corrected.

Turn to implementing method `display`.

```
class Sim:
    ...

    @classmethod
    def display(cls) -> None:
        """Display the present Universe."""
        pass

    ...
```

display

Turn to implementing method `display`.

```
class Sim:
    ...

    @classmethod
    def display(cls) -> None:
        """Display the present Universe."""
        print( "Generation: ", Sim.generation )
        ___________

    ...
```

display

Use a standard row-major-order traversal.

```
class Sim:
    ...

    @classmethod
    def display(cls) -> None:
        """Display the present Universe."""
        print( "Generation: ", Sim.generation )
        for r in range(0, Sim.M):
            for c in range(0, Sim.N): print(Sim.old[r][c], end='')
            print()

    ...
```

☞ **Master stylized code patterns, and use them.**

display

Use a standard row-major-order traversal, output cells.

```
class Sim:
    ...

    @classmethod
    def display(cls) -> None:
        """Display the present Universe."""
        print( "Generation: ", Sim.generation )
        for r in range(0, Sim.M):
            for c in range(0, Sim.N): print(Sim.old[r][c], end='')
            print()

    ...
```

☞ **Master stylized code patterns, and use them.**

display

Use a standard row-major-order traversal, output cells, and output newlines at row ends.

```
class Sim:
    ...

    @classmethod
    def display(cls) -> None:
        """Display the present Universe."""
        print( "Generation: ", Sim.generation )
        for r in range(0, Sim.M):
            for c in range(0, Sim.N): print(Sim.old[r][c], end='')
            print()

    ...
```

☞ **Master stylized code patterns, and use them.**

display

Use a standard row-major-order traversal, output cells, and output newlines at row ends.

```
class Sim:
    ...

    @classmethod
    def display(cls) -> None:
        """Display the present Universe."""
        print( "Generation: ", Sim.generation )
        for r in range(0, Sim.M):
            for c in range(0, Sim.N): print(Sim.old[r][c], end='')
            print()

    ...
```

Means: Don’t go to the beginning of the next line after printing.

☞ **Master stylized code patterns, and use them.**

display

Use a standard row-major-order traversal, output cells, and output newlines at row ends.

```
class Sim:
    ...

    @classmethod
    def display(cls) -> None:
        """Display the present Universe."""
        print( "Generation: ", Sim.generation )
        for r in range(0, Sim.M):
            for c in range(0, Sim.N): print(Sim.old[r][c], end='')
            print()

    ...
```

Variables `r` and `c` are local variables of method `display`.
Refer to local variables without qualification.

display

Use a standard row-major-order traversal, output cells, and output newlines at row ends.

```
class Sim:
    ...

    @classmethod
    def display(cls) -> None:
        """Display the present Universe."""
        print( "Generation: ", Sim.generation )
        for r in range(0, Sim.M):
            for c in range(0, Sim.N): print(Sim.old[r][c], end='')
            print()

    ...
```

Variables generation, M, N, and old are class variables.
Refer to class variables wth qualified names.

next_generation

A method stub may suffice for a program test.

```
class Sim:
    ...

    @classmethod
    def next_generation(cls) -> None:
        """Update Universe to be the next generation."""
        pass
    ...
```

☞ **Write degenerate program stubs that allow partial programs to execute.**

Testing

Test early and often.

To try it out, invoke `Sim.main()`.

☞ **Test programs incrementally.**

☞ **Never be (very) lost. Don't stray far from a correct (albeit, partial) program.**

Output:

```
Generation: 0
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
Generation: 1
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
Generation: 2
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
Etc.
```

What has been validated?

- Generation counting
- Formatting of output

☞ **Validate output thoroughly.**

We now turn to implementation of the method stub: next_generation.

```
class Sim:
    ...

    @classmethod
    def next_generation(cls) -> None:
        """Update Universe to be the next generation."""
        pass

    ...
```

☞ **Master stylized code patterns, and use them.**

next_generation

Instance of the standard *compute-use* pattern.

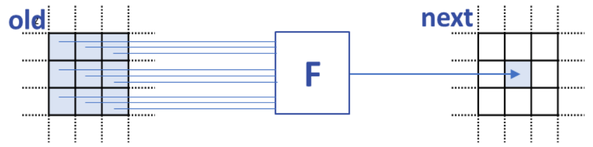
```
class Sim:
    ...

    @classmethod
    def next_generation(cls) -> None:
        """Update Universe to be the next generation."""
        #.Determine the states of next[][] as F(old[][] states).
        #.Swap old[][] and next[][] Universes.
```

☞ **Master stylized code patterns, and use them.**

next_generation

Standard row-major-order traversal for determining new states of each cell of next.

```
class Sim:
    ...

    @classmethod
    def next_generation(cls) -> None:
        """Update Universe to be the next generation."""
        # Determine the states of next[][] as F(old[][] states).
        for r in range(0, Sim.M):
            for c in range(0, Sim.N):
```



```
        #.Swap old[][] and next[][] Universes.

    ...
```

Standard row-major-order traversal for determining new states of each cell of next.

```
class Sim:
    ...

    @classmethod
    def next_generation(cls) -> None:
        """Update Universe to be the next generation."""
        # Determine the states of next[][] as F(old[][] states).
        for r in range(0, Sim.M):
            for c in range(0, Sim.N):
                # Sim.next[r][c] = F(Sim.old[r][c] and its neighbors)

        #.Swap old[][] and next[][] Universes.

    ...
```

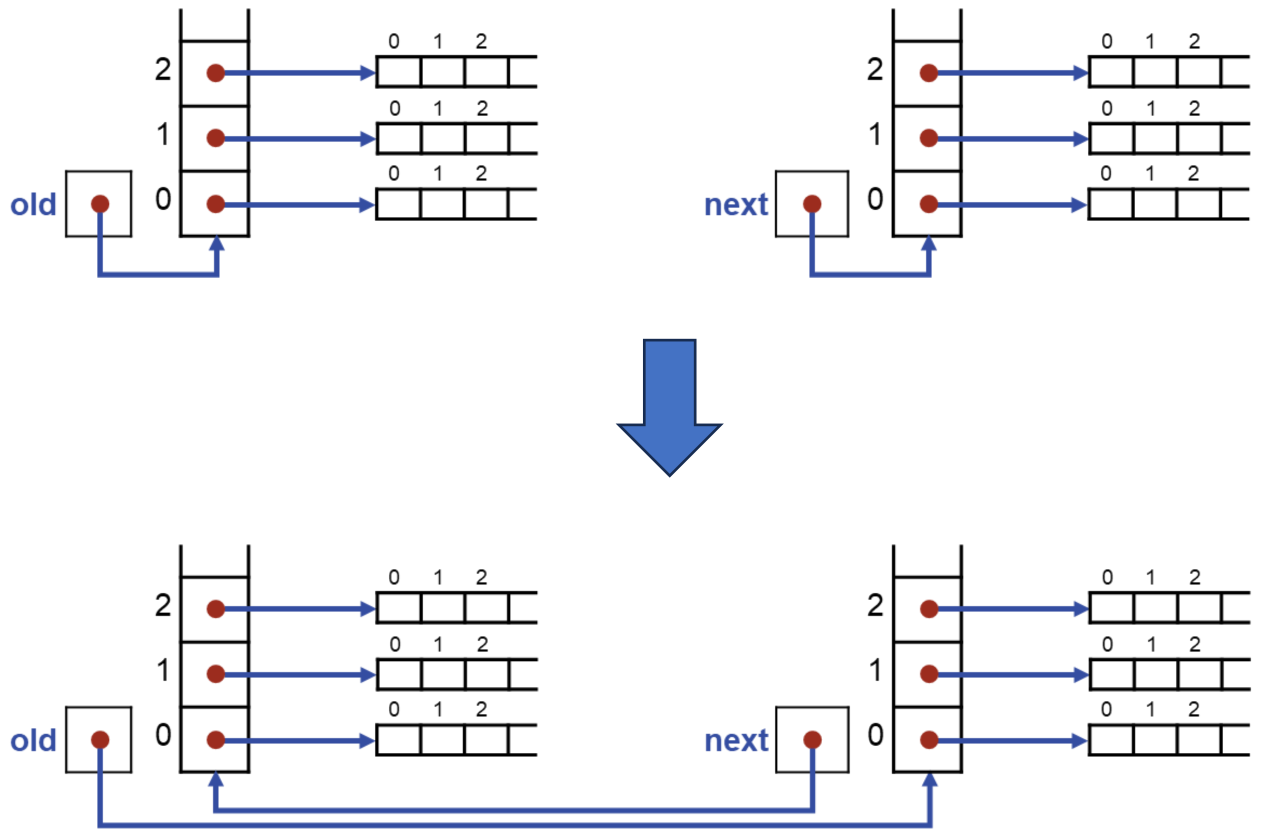
Standard code for swap

```
class Sim:
    ...

    @classmethod
    def next_generation(cls) -> None:
        """Update Universe to be the next generation."""
        # Determine the states of next[][] as F(old[][] states).
        for r in range(0, Sim.M):
            for c in range(0, Sim.N):
                # Sim.next[r][c] = F(Sim.old[r][c] and its neighbors)

        # Swap old[][] and next[][] Universes.
        temp = Sim.old;  Sim.old = Sim.next;  Sim.next = temp

   ...
```

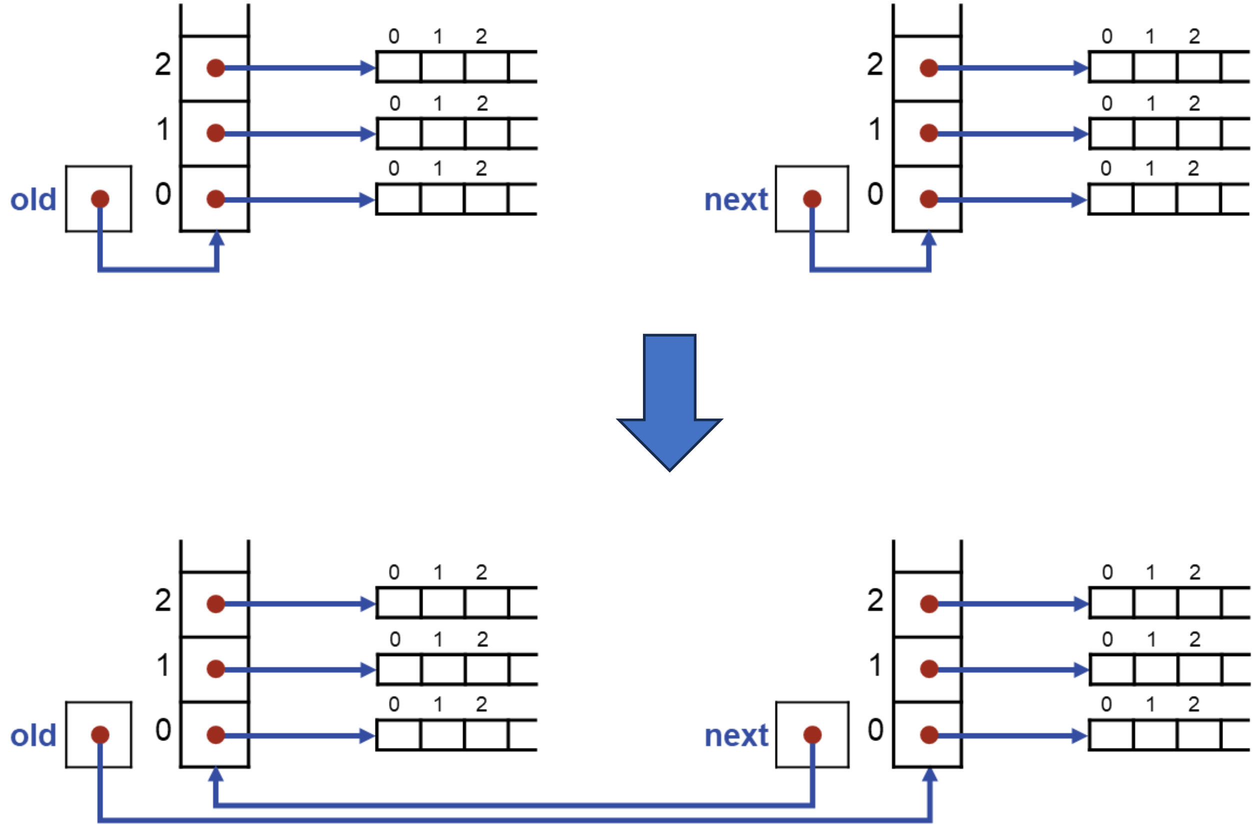
☞ **Master stylized code patterns, and use them.**

next_generation

Notice that swap is a constant-time operation, independent of the size of the Universes.



Notice that swap is a constant-time operation, independent of the size of the Universes.*



*__C/C++__ Constant-time swap is not available for C-style arrays in C/C++. Rather, this can be read as describing one of the alternatives to C-style arrays that are available in C++.

next_generation

Completed `next_generation` for a generic cellular automaton.

```
class Sim:
    ...

    @classmethod
    def next_generation(cls) -> None:
        """Update Universe to be the next generation."""
        # Determine the states of next[][] as F(old[][] states).
        for r in range(0, Sim.M):
            for c in range(0, Sim.N):
                # Sim.next[r][c] = F(Sim.old[r][c] and its neighbors)

        # Swap old[][] and next[][] Universes.
        temp = Sim.old;  Sim.old = Sim.next;  Sim.next = temp

    ...
```

Testing

For easy immediate testing, let each cell increment its state on each generation.

```
class Sim:
    ...

    @classmethod
    def next_generation(cls) -> None:
        """Update Universe to be the next generation."""
        # Determine the states of next[][] as F(old[][] states).
        for r in range(0, Sim.M):
            for c in range(0, Sim.N):
                Sim.next[r][c] = Sim.old[r][c] + 1

        # Swap old[][] and next[][] Universes.
        temp = Sim.old;  Sim.old = Sim.next;  Sim.next = temp

    ...
```

☞ **Test programs incrementally.**

Test early and often.

To try it out again, invoke `Sim.main()`.

☞ **Test programs incrementally.**

☞ **Never be (very) lost. Don’t stray far from a correct (albeit, partial) program.**

Output:

```
Generation: 0
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
Generation: 1
11111111111111111111
11111111111111111111
11111111111111111111
11111111111111111111
11111111111111111111
Generation: 2
22222222222222222222
22222222222222222222
22222222222222222222
22222222222222222222
22222222222222222222
Etc.
```

What has been validated?

- Generation counting
- Formatting of output
- Creation of `next` Universe from `old` Universe
- Swapping of `old` and `next` Universes

☞ **Validate output thoroughly.**

Game of Life

Game of Life. A Cellular Automaton in which each cell is either dead or alive.

In each generation:

- Each live cell with 2 or 3 live neighbors lives on to the next generation (life) otherwise it dies (death).
- Each dead cell with 3 live neighbors comes alive in the next generation (birth) otherwise it remains dead.

Each cell is either dead or alive, so specialize the Universes as Boolean 2-D arrays.

```
class Sim:

    M: int = 5                                # Height of Universe.
    N: int = 20                               # Width of Universe.
    old: list[list[bool]]  = []               # old Universe.
    next: list[list[bool]] = []               # next Universe.
    LAST_GEN: int = 50                        # Last generation.
    generation: int = 0                       # Generation number.
```

☞ **Choose representations that by design don't have nonsensical configurations.**

Revise method initialize similarly.

```
class Sim:
    ...
    @classmethod
    def initialize(cls) -> None:
        """Create the initial Universe."""
        # Initialize old and new Universes to M-by-N arrays of False.
        Sim.old =  [[False for _ in range(0, Sim.N)]
                     for _ in range(0, Sim.M)]
        Sim.next = [[False for _ in range(0, Sim.N)]
                     for _ in range(0, Sim.M)]
    ...
```

☞ **Choose representations that by design don’t have nonsensical configurations.**

display

Revise method `display` to compactly render dead as “_” and alive as “X“.

```
class Sim:
    ...
    @classmethod
    def display(cls) -> None:
        """Display the present Universe."""
        print( "Generation: ", Sim.generation )
        for r in range(0, Sim.M):
            for c in range(0, Sim.N): print(Sim.old[r][c], end='')
            print()
    ...
```

display

Revise method display to compactly render dead as “_” and alive as “X“.

```
class Sim:
    ...
    @classmethod
    def display(cls) -> None:
        """Display the present Universe."""“
        print( "Generation: ", Sim.generation )
        for r in range(0, Sim.M):
            for c in range(0, Sim.N):
                if Sim.old[r][c]: print("X", end='')
                else: print("_", end='')
            print()
    ...
```

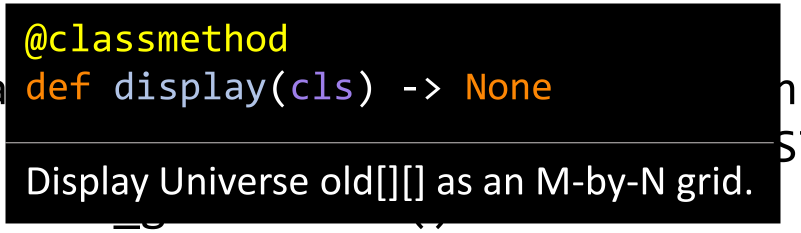
While we are at it, revise the method header to be more informative.

```
class Sim:
    ...
    @classmethod
    def display(cls) -> None:
        """Display Universe old[][] as an M-by-N grid."""
        print( "Generation: ", Sim.generation )
        for r in range(0, Sim.M):
            for c in range(0, Sim.N):
                if Sim.old[r][c]: print("X", end='')
                else: print("_", end='')
            print()
    ...
```

☞ **Repeatedly improve comments by relentless copy editing.**

That way, for example, the tip provided in an IDE editor will be more helpful.

```
class Sim:
    ...
    @classmethod
    def main(cls) -> None:
        """Simulate a cellular automaton."""
        # Create the initial Universe and display it.
        Sim.initialize()
        Sim.display()

        # Simula                                      hing generations.
        for Sim.                                      ST_GEN + 1):
            Sim.
            Sim.display()
```



```
@classmethod
def display(cls) -> None
```

Display Universe old[][] as an M-by-N grid.

☞ **Repeatedly improve comments by relentless copy editing.**

Similarly, turn end-of-line comments of variable declarations into docstrings.

```
class Sim:

    M: int = 5                                       # Height of Universe.
    N: int = 20                                      # Width of Universe.
    old: list[list[bool]]  = []                      # old Universe.
    next: list[list[bool]] = []                      # next Universe.
    LAST_GEN: int = 50                               # Last generation.
    generation: int = 0                              # Generation number.
```

☞ **Repeatedly improve comments by relentless copy editing.**

Documentation

Similarly, you can turn end-of-line comments of variable declarations into docstrings.

```
class Sim:

    M: int = 5
    """M is the height of the Universe."""

    N: int = 20
    """N is the width of the Universe."""

    old: list[list[bool]]  = []
    """old is the present state of the Universe."""

    next: list[list[bool]] = []
    """next is the upcoming generation of the Universe, in preparation."""

    LAST_GEN: int = 40
    """LAST_GEN is last generation to be simulated."""

    generation: int = 0
    """generation is the number of the present generation."""
```

Documentation

The docstring of a variable is the variable's representation invariant.

```
class Sim:
    ...
    @classmethod
    def display(cls) -> None:
        """Display Universe old[][] as an M-by-N grid."""
        print( "Generation: ", Sim.generation )
        for r in range(0, Sim.M):
            for c in range(0,
                if Sim.old[r]
                else: print("
            print()
    ...
```

```
class attribute M of Sim
M: int = 5
```

M is the height of the Universe.

☞ **Repeatedly improve comments by relentless copy editing.**

The docstring of a variable is the variable’s representation invariant.

```
class Sim:
    ...
    @classmethod
    def display(cls) -> None:
        """Display Universe old[][] as an M-by-N grid."""
        print( "Generation: ", Sim.generation )
        for r in range(0, Sim.M):
            for c in range(0, Sim.N):
                if Sim.old[r][c]: print("X", end='')
                else:
            print()
    ...
```

```
class attribute old of Sim
old: list[list[bool]] = []
```

old is the present state of the Universe.

☞ **Repeatedly improve comments by relentless copy editing.**

Implement the Game of Life rules.

```
class Sim:
    ...
    @classmethod
    def next_generation(cls) -> None:
        """Update old[][] to be the next generation of the Universe."""
        # Determine the states of next[][] as F(old[][] states).
        for r in range(0, Sim.M):
            for c in range(0, Sim.N):
                # Set next[r][c] = according to the Game of Life rules.

        # Swap old[][] and next[][] Universes.
        temp = Sim.old;  Sim.old = Sim.next;  Sim.next = temp
    ...
```

Game of Life

Refine the specification using the standard *compute-use* pattern.

```
@classmethod
def next_generation(cls) -> None:
    """Update old[][] to be the next generation of the Universe."""
    # Determine the states of next[][] as F(old[][] states).
    for r in range(0, Sim.M):
        for c in range(0, Sim.N):
            #.Compute.
            #.Use.

    # Swap old[][] and next[][] Universes.
    temp = Sim.old;  Sim.old = Sim.next;  Sim.next = temp
```

☞ **Master stylized code patterns, and use them.**

Instantiate placeholders *Compute* and *Use* for the problem at hand.

```
@classmethod
def next_generation(cls) -> None:
    """Update old[][] to be the next generation of the Universe."""
    # Determine the states of next[][] as F(old[][] states).
    for r in range(0, Sim.M):
        for c in range(0, Sim.N):
            #.Let live_neighbors be number of alive cells around old[r][c].
            #.Set next[r][c] according to the rules re. live_neighbors.

    # Swap old[][] and next[][] Universes.
    temp = Sim.old;  Sim.old = Sim.next;  Sim.next = temp
```

☞ **Master stylized code patterns, and use them.**

Use is a structured four-way case analysis.

```
@classmethod
def next_generation(cls) -> None:
    """Update old[][] to be the next generation of the Universe."""
    # Determine the states of next[][] as F(old[][] states).
    for r in range(0, Sim.M):
        for c in range(0, Sim.N):
            #.Let live_neighbors be number of alive cells around old[r][c].

            # Set next[r][c] according to the rules re. live_neighbors.
            if Sim.old[r][c]:  # Currently live.
                if (live_neighbors == 2) or (live_neighbors == 3):
                     Sim.next[r][c] = True
                else: Sim.next[r][c] = False
            else:  # Currently dead.
                if live_neighbors == 3:
                   Sim.next[r][c] = True
                else: Sim.next[r][c] = False

    # Swap old[][] and next[][] Universes.
    temp = Sim.old;  Sim.old = Sim.next;  Sim.next = temp
```

Compute is a 3x3 row-major-order traversal, counting `live_neighbors` as appropriate.

```
@classmethod
def next_generation(cls) -> None:
    """Update old[][] to be the next generation of the Universe."""
    # Determine the states of next[][] as F(old[][] states).
    for r in range(0, Sim.M):
        for c in range(0, N):
            # Let live_neighbors be number of alive cells around old[r][c].
                live_neighbors = 0
                for dr in range(-1, 2):
                    for dc in range(-1, 2):
                        if not((dr == 0) and (dc == 0)) and (
                            Sim.old[r + dr][c + dc] ):
                                live_neighbors += 1

            # Set next[r][c] following the rules re. live_neighbors.
            ...

    # Swap old[][] and next[][] Universes.
    temp = Sim.old;  Sim.old = Sim.next;  Sim.next = temp;
```

To prevent the subscripts from going out of bounds

```
@classmethod
def next_generation(cls) -> None:
    """Update old[][] to be the next generation of the Universe."""
    # Determine the states of next[][] as F(old[][] states).
    for r in range(0, Sim.M):
        for c in range(0, Sim.N):
            # Let live_neighbors be number of alive cells around old[r][c].
                live_neighbors = 0
                for dr in range(-1, 2):
                    for dc in range(-1, 2):
                        if not((dr == 0) and (dc == 0)) and (
                            Sim.old[r + dr][c + dc] ):
                                live_neighbors += 1

            # Set next[r][c] following the rules re. live_neighbors.
            ...

    # Swap old[][] and next[][] Universes.
    temp = Sim.old;  Sim.old = Sim.next;  Sim.next = temp;
```

To prevent the subscripts from going out of bounds, simulate on a torus.

```
@classmethod
def next_generation(cls) -> None:
    """Update old[][] to be the next generation of the Universe."""
    # Determine the states of next[][] as F(old[][] states).
    for r in range(0, Sim.M):
        for c in range(0, Sim.N):
            # Let live_neighbors be number of alive cells around old[r][c].
                live_neighbors = 0
                for dr in range(-1, 2):
                    for dc in range(-1, 2):
                        if not((dr == 0) and (dc == 0)) and (
                            Sim.old[(r + dr) % Sim.M][(c + dc) % Sim.N] ):
                                live_neighbors += 1

            # Set next[r][c] following the rules re. live_neighbors.
            ...

    # Swap old[][] and next[][] Universes.
    temp = Sim.old;  Sim.old = Sim.next;  Sim.next = temp;
```

Create some life, which will glide diagonally down and to the right.

```
@classmethod
def initialize(cls) -> None:
    """Create the initial Universe.""“
    # Initialize old and new Universes to M-by-N arrays of False.
        ...
    # Glider
        Sim.old[0][1] = True
        Sim.old[1][2] = True
        Sim.old[2][0] = True
        Sim.old[2][1] = True
        Sim.old[2][2] = True
```

old	0	1	2	…
0		T		…
1			T	…
2	T	T	T	…
	…	…	…	…

To let it rip, invoke `Sim.main()` yet again.

And presto …

```
Generation: 0
_X__________________
__X_________________
XXX_________________
____________________
____________________
```

And presto …

```
Generation:  1

____________________
X_X_________________
_XX_________________
_X__________________
____________________
```

And presto …

```
Generation:  2
____________________
__X_________________
X_X_________________
_XX_________________
____________________
```

And presto …

```
Generation:  3
____________________
_X__________________
__XX________________
_XX_________________
____________________
```

And presto …

```
Generation:  4
____________________
__X_________________
___X________________
_XXX________________
____________________
```

Back to the same configuration as Generation 0, but shifted down and right one cell.

And presto …

```
Generation:  5
____________________
____________________
_X_X________________
__XX________________
__X_________________
```

And presto …

```
Generation:  6
____________________
____________________
___X________________
_X_X________________
__XX________________
```

Testing

And presto …

```
Generation:  7
____________________
____________________
__X_________________
___XX_______________
__XX________________
```

And presto …

```
Generation:  8
____________________
____________________
___X________________
____X_______________
__XXX_______________
```

Back to the same configuration as Generation 1, but shifted down and right one cell.

And presto …

```
Generation:  9
___X____________________
________________________
________________________
__X_X___________________
___XX___________________
```

Whoa! What’s going on? Oh, I forgot, we are on a torus.

And presto …

```
Generation:  10
___XX_______________
____________________
____________________
____X_______________
__X_X_______________
Generation:  11
```

And presto …

```
Generation:  11
___XX _______________
____________________
____________________
___X________________
____XX______________
```

Testing

And presto …

```
Generation:  12
___XXX______________
____________________
____________________
____X_______________
_____X______________
```

Testing

And presto …

```
Generation:  13
____XX______________
____X_______________
____________________
____________________
___X_X______________
```

And presto …

```
Generation:  14
___X_X______________
____XX______________
____________________
____________________
_____X______________
```

And presto …

```
Generation:  15
_____XX_____________
____XX______________
____________________
____________________
____X_______________
```

Testing

And presto …

```
Generation:  16
______X_____________
____XXX_____________
____________________
____________________
_____X______________
```

And presto …

```
Generation:  17
____X_X_____________
_____XX_____________
_____X______________
____________________
____________________
```

Testing

And presto …

```
Generation:  18
______X_____________
____X_X_____________
_____XX_____________
____________________
____________________
```

And presto …

```
Generation:  19
_____X______________
______XX____________
_____XX_____________
____________________
____________________
```

And presto …

```
Generation:   20
______X_____________
_______X____________
_____XXX____________
____________________
____________________
```

Back to the same configuration as Generation 0, but shifted right several cells. The glider is coiling around the donut!

Boundary Conditions

What are the boundary conditions for this problem, and did we forget them?

For example, what if the height of the Universe were only 4? To try it out, change N, and invoke `Sim.main()`.

```
Generation: 0
_X__________________
__X_________________
XXX_________________
____________________
```

☞ **Boundary conditions. Dead last, but don’t forget them.**

Boundary Conditions

What are the boundary conditions for this problem, and did we forget them?

For example, what if the height of the Universe were only 4? To try it out, change N, and invoke `Sim.main()`.

```
Generation: 0
_X__________________
__X_________________
XXX_________________
____________________
Generation:  1
____________________
X_X_________________
_XX_________________
X_X_________________
```

There isn't enough "elbowroom" around the glider, and it is interfering with its own propagation. By generation 6, all life is gone!

Should your program be defensive and prevent this, or is this just how life goes?

☞ **Boundary conditions. Dead last, but don't forget them.**

Summary:

The notion of a **class** has been introduced as a means for aggregating variables and methods.

Many standard precepts, patterns, and recommended coding techniques have been illustrated.

Representation invariants for data structures and their components have been emphasized, and their effective use in IDE’s shown.

And the Game of Life itself is fascinating.